NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer’s Manual: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.
Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.
No product or component can be absolutely secure.
Your costs and results may vary.
You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.
All product plans and roadmaps are subject to change without notice.
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.
Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.
Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.
No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.
© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
NOTE: The Intel 64 and IA-32 Architectures Software Developer’s Manual consists of four volumes: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.
Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
### CONTENTS

#### CHAPTER 1

**ABOUT THIS MANUAL**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL</td>
</tr>
<tr>
<td>1.2</td>
<td>OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE</td>
</tr>
<tr>
<td>1.3</td>
<td>NOTATIONAL CONVENTIONS</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Bit and Byte Order</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Reserved Bits and Software Compatibility</td>
</tr>
<tr>
<td>1.3.2.1</td>
<td>Instruction Operands</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Hexadecimal and Binary Numbers</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Segmented Addressing</td>
</tr>
<tr>
<td>1.3.5</td>
<td>A New Syntax for CPUID, CR, and MSR Values</td>
</tr>
<tr>
<td>1.3.6</td>
<td>Exceptions</td>
</tr>
<tr>
<td>1.4</td>
<td>RELATED LITERATURE</td>
</tr>
</tbody>
</table>

#### CHAPTER 2

**INTEL® 64 AND IA-32 ARCHITECTURES**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>BRIEF HISTORY OF INTEL® 64 AND IA-32 ARCHITECTURES</td>
</tr>
<tr>
<td>2.1.1</td>
<td>16-Bit Processors and Segmentation (1978)</td>
</tr>
<tr>
<td>2.1.2</td>
<td>The Intel® 286 Processor (1982)</td>
</tr>
<tr>
<td>2.1.3</td>
<td>The Intel®386™ Processor (1985)</td>
</tr>
<tr>
<td>2.1.4</td>
<td>The Intel®486™ Processor (1989)</td>
</tr>
<tr>
<td>2.1.5</td>
<td>The Intel® Pentium® Processor (1993)</td>
</tr>
<tr>
<td>2.1.7</td>
<td>The Intel® Pentium® Processor Family (2000—2006)</td>
</tr>
<tr>
<td>2.1.8</td>
<td>The Intel® Xeon® Processor (2001—2007)</td>
</tr>
<tr>
<td>2.1.9</td>
<td>The Intel® Pentium® M Processor (2003—2006)</td>
</tr>
<tr>
<td>2.1.10</td>
<td>The Intel® Pentium® Processor Extreme Edition (2005)</td>
</tr>
<tr>
<td>2.1.11</td>
<td>The Intel® Core™ Processor Family (2006—2007)</td>
</tr>
<tr>
<td>2.1.12</td>
<td>The Intel® Xeon® Processor 5100, 5300 Series, and Intel® Core™ 2 Processor Family (2006)</td>
</tr>
<tr>
<td>2.1.13</td>
<td>The Intel® Xeon® Processor 5200, 5400, 7400 Series, and Intel® Core™ 2 Processor Family (2007)</td>
</tr>
<tr>
<td>2.1.15</td>
<td>The Intel® Atom™ Processor Family Based on Silvermont Microarchitecture (2013)</td>
</tr>
<tr>
<td>2.1.16</td>
<td>The Intel® Core™ i7 Processor Family (2008)</td>
</tr>
<tr>
<td>2.1.17</td>
<td>The Intel® Xeon® Processor 7500 Series (2010)</td>
</tr>
<tr>
<td>2.1.18</td>
<td>2010 Intel® Core™ Processor Family (2010)</td>
</tr>
<tr>
<td>2.1.19</td>
<td>The Intel® Xeon® Processor 5600 Series (2010)</td>
</tr>
<tr>
<td>2.1.20</td>
<td>The Second Generation Intel® Core™ Processor Family (2011)</td>
</tr>
<tr>
<td>2.1.21</td>
<td>The Third Generation Intel® Core™ Processor Family (2012)</td>
</tr>
<tr>
<td>2.1.22</td>
<td>The Fourth Generation Intel® Core™ Processor Family (2013)</td>
</tr>
<tr>
<td>2.2</td>
<td>MORE ON SPECIFIC ADVANCES</td>
</tr>
<tr>
<td>2.2.1</td>
<td>P6 Family Microarchitecture</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Intel NetBurst® Microarchitecture</td>
</tr>
<tr>
<td>2.2.2.1</td>
<td>The Front End Pipeline</td>
</tr>
<tr>
<td>2.2.2.2</td>
<td>Out-Of-Order Execution Core</td>
</tr>
<tr>
<td>2.2.2.3</td>
<td>Retirement Unit</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Intel® Core™ Microarchitecture</td>
</tr>
<tr>
<td>2.2.3.1</td>
<td>The Front End</td>
</tr>
<tr>
<td>2.2.3.2</td>
<td>Execution Core</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Intel Atom® Microarchitecture</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Nehalem Microarchitecture</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Sandy Bridge Microarchitecture</td>
</tr>
<tr>
<td>2.2.7</td>
<td>SIMD Instructions</td>
</tr>
<tr>
<td>2.2.8</td>
<td>Intel® Hyper-Threading Technology</td>
</tr>
<tr>
<td>2.2.8.1</td>
<td>Some Implementation Notes</td>
</tr>
<tr>
<td>2.2.9</td>
<td>Multi-Core Technology</td>
</tr>
</tbody>
</table>
CHAPTER 6
PROCEDURE CALLS, INTERRUPTS, AND EXCEPTIONS

6.1 PROCEDURE CALL TYPES ........................................................................... 6-1
6.2 STACKS ........................................................................................................... 6-1
6.2.1 Setting Up a Stack .................................................................................. 6-2
6.2.2 Stack Alignment ....................................................................................... 6-2
6.2.3 Address-Size Attributes for Stack Accesses ............................................. 6-3
6.2.4 Procedure Linking Information ................................................................. 6-3
6.2.4.1 Stack-Frame Base Pointer ............................................................... 6-3
6.2.4.2 Return Instruction Pointer ............................................................... 6-3
6.2.5 Stack Behavior in 64-Bit Mode ................................................................. 6-4
6.3 SHADOW STACKS ......................................................................................... 6-4
6.4 CALLING PROCEDURES USING CALL AND RET .................................... 6-4
6.4.1 Near CALL and RET Operation ............................................................... 6-4
6.4.2 Far CALL and RET Operation ................................................................. 6-5
6.4.3 Parameter Passing ................................................................................... 6-7
6.4.3.1 Passing Parameters Through the General-Purpose Registers .............. 6-7
6.4.3.2 Passing Parameters on the Stack ....................................................... 6-7
6.4.3.3 Passing Parameters in an Argument List ......................................... 6-7
6.4.4 Saving Procedure State Information ...................................................... 6-7
6.4.5 Calls to Other Privilege Levels ............................................................... 6-7
6.4.6 CALL and RET Operation Between Privilege Levels ............................ 6-8
6.4.7 Branch Functions in 64-Bit Mode ............................................................ 6-12
6.5 INTERRUPTS AND EXCEPTIONS ............................................................. 6-12
6.5.1 Call and Return Operation for Interrupt or Exception Handling Procedures ........................................................................ 6-13
6.5.2 Calls to Interrupt or Exception Handler Tasks ........................................ 6-18
6.5.3 Interrupt and Exception Handling in Real-Address Mode ...................... 6-18
6.5.4 INT n, INTO, INT3, INT1, and BOUND Instructions .............................. 6-18
6.5.5 Handling Floating-Point Exceptions ...................................................... 6-19
6.5.6 Interrupt and Exception Behavior in 64-Bit Mode ................................ 6-19
6.6 PROCEDURE CALLS FOR BLOCK-STRUCTURED LANGUAGES .............. 6-20
6.6.1 ENTER Instruction ................................................................................. 6-20

Vol. 1 vii
## CONTENTS

### 6.6.2 LEAVE Instruction ................................................................. 6-24

### CHAPTER 7
**PROGRAMMING WITH GENERAL-PURPOSE INSTRUCTIONS**

7.1  PROGRAMMING ENVIRONMENT FOR GP INSTRUCTIONS .................................................. 7-1
7.2  PROGRAMMING ENVIRONMENT FOR GP INSTRUCTIONS IN 64-BIT MODE ................................. 7-1
7.3  SUMMARY OF GP INSTRUCTIONS .................................................................................... 7-2
7.3.1 Data Transfer Instructions ......................................................................................... 7-2
7.3.1.1 General Data Movement Instructions ................................................................. 7-3
7.3.1.2 Exchange Instructions ......................................................................................... 7-4
7.3.1.3 Exchange Instructions in 64-Bit Mode ............................................................... 7-5
7.3.1.4 Stack Manipulation Instructions ......................................................................... 7-5
7.3.1.5 Stack Manipulation Instructions in 64-Bit Mode ............................................... 7-7
7.3.1.6 Type Conversion Instructions ............................................................................. 7-7
7.3.1.7 Type Conversion Instructions in 64-Bit Mode .................................................... 7-8
7.3.2 Binary Arithmetic Instructions ................................................................................. 7-8
7.3.2.1 Addition and Subtraction Instructions ................................................................. 7-8
7.3.2.2 Increment and Decrement Instructions .............................................................. 7-8
7.3.2.3 Increment and Decrement Instructions in 64-Bit Mode ...................................... 7-8
7.3.2.4 Comparison and Sign Change Instructions ......................................................... 7-8
7.3.2.5 Multiplication and Division Instructions ............................................................ 7-9
7.3.3 Decimal Arithmetic Instructions .............................................................................. 7-9
7.3.3.1 Packed BCD Adjustment Instructions ................................................................. 7-9
7.3.3.2 Unpacked BCD Adjustment Instructions ............................................................ 7-9
7.3.4 Decimal Arithmetic Instructions in 64-Bit Mode ....................................................... 7-10
7.3.5 Logical Instructions ................................................................................................. 7-10
7.3.5.1 Shift and Rotate Instructions ............................................................................ 7-10
7.3.5.2 Double-Shift Instructions .................................................................................. 7-12
7.3.5.3 Rotate Instructions ........................................................................................... 7-13
7.3.6 Bit and Byte Instructions ......................................................................................... 7-13
7.3.6.1 Bit Test and Modify Instructions ...................................................................... 7-14
7.3.6.2 Bit Scan Instructions ....................................................................................... 7-14
7.3.6.3 Byte Set on Condition Instructions ................................................................... 7-14
7.3.7 Test Instruction ....................................................................................................... 7-14
7.3.8 Control Transfer Instructions .................................................................................. 7-14
7.3.8.1 Unconditional Transfer Instructions ................................................................. 7-14
7.3.8.2 Conditional Transfer Instructions ................................................................... 7-15
7.3.8.3 Control Transfer Instructions in 64-Bit Mode .................................................... 7-17
7.3.8.4 Software Interrupt Instructions ........................................................................ 7-17
7.3.8.5 Software Interrupt Instructions in 64-Bit Mode and Compatibility Mode .......... 7-18
7.3.9 String Operations ................................................................................................... 7-18
7.3.9.1 String Instructions .......................................................................................... 7-18
7.3.9.2 Repeated String Operations ............................................................................ 7-19
7.3.9.3 Fast-String Operation ...................................................................................... 7-19
7.3.9.4 String Operations in 64-Bit Mode ................................................................... 7-20
7.3.10 I/O Instructions ..................................................................................................... 7-20
7.3.11 I/O Instructions in 64-Bit Mode ................................................................................ 7-20
7.3.12 Enter and Leave Instructions .................................................................................. 7-21
7.3.13 Flag Control (EFLAG) Instructions ....................................................................... 7-21
7.3.13.1 Carry and Direction Flag Instructions .............................................................. 7-21
7.3.13.2 EFLAGS Transfer Instructions ....................................................................... 7-21
7.3.13.3 Interrupt Flag Instructions ............................................................................... 7-22
7.3.14 Flag Control (RFLAG) Instructions in 64-Bit Mode ................................................ 7-22
7.3.15 Segment Register Instructions ................................................................................ 7-22
7.3.15.1 Segment-Register Load and Store Instructions .................................................. 7-22
7.3.15.2 Far Control Transfer Instructions .................................................................... 7-22
7.3.15.3 Software Interrupt Instructions ...................................................................... 7-23
7.3.15.4 Load Far Pointer Instructions ........................................................................... 7-23

viii Vol. 1
# PROGRAMMING WITH THE X87 FPU

## CHAPTER 8

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>X87 FPU EXECUTION ENVIRONMENT</td>
<td>8-1</td>
</tr>
<tr>
<td>x87 FPU in 64-Bit Mode and Compatibility Mode</td>
<td>8-1</td>
</tr>
<tr>
<td>x87 FPU Data Registers</td>
<td>8-1</td>
</tr>
<tr>
<td>Parameter Passing With the x87 FPU Register Stack</td>
<td>8-3</td>
</tr>
<tr>
<td>x87 FPU Status Register</td>
<td>8-4</td>
</tr>
<tr>
<td>Top of Stack (TOP) Pointer</td>
<td>8-4</td>
</tr>
<tr>
<td>Condition Code Flags</td>
<td>8-4</td>
</tr>
<tr>
<td>x87 FPU Floating-Point Exception Flags</td>
<td>8-5</td>
</tr>
<tr>
<td>Stack Fault Flag</td>
<td>8-6</td>
</tr>
<tr>
<td>Branching and Conditional Moves on Condition Codes</td>
<td>8-6</td>
</tr>
<tr>
<td>x87 FPU Control Word</td>
<td>8-7</td>
</tr>
<tr>
<td>x87 FPU Floating-Point Exception Mask Bits</td>
<td>8-7</td>
</tr>
<tr>
<td>Precision Control Field</td>
<td>8-7</td>
</tr>
<tr>
<td>Rounding Control Field</td>
<td>8-8</td>
</tr>
<tr>
<td>Infinity Control Flag</td>
<td>8-8</td>
</tr>
<tr>
<td>x87 FPU Tag Word</td>
<td>8-8</td>
</tr>
<tr>
<td>x87 FPU Instruction and Data (Operand) Pointers</td>
<td>8-9</td>
</tr>
<tr>
<td>Last Instruction Opcode</td>
<td>8-10</td>
</tr>
<tr>
<td>Fopcode Compatibility Sub-mode</td>
<td>8-10</td>
</tr>
<tr>
<td>Saving the x87 FPU State with FSTENV/FNSTENV and FSAVE/FNSAVE</td>
<td>8-11</td>
</tr>
<tr>
<td>Saving the x87 FPU State with FXSAVE</td>
<td>8-12</td>
</tr>
<tr>
<td>X87 FPU DATA TYPES</td>
<td>8-13</td>
</tr>
<tr>
<td>Indefinites</td>
<td>8-14</td>
</tr>
<tr>
<td>Unsupported Double Extended Precision Floating-Point Encodings and Pseudo-Denormals</td>
<td>8-14</td>
</tr>
<tr>
<td>X87 FPU INSTRUCTION SET</td>
<td>8-15</td>
</tr>
<tr>
<td>Escape (ESC) Instructions</td>
<td>8-15</td>
</tr>
<tr>
<td>x87 FPU Instruction Operands</td>
<td>8-15</td>
</tr>
<tr>
<td>Data Transfer Instructions</td>
<td>8-15</td>
</tr>
<tr>
<td>Load Constant Instructions</td>
<td>8-17</td>
</tr>
<tr>
<td>Basic Arithmetic Instructions</td>
<td>8-17</td>
</tr>
<tr>
<td>Comparison and Classification Instructions</td>
<td>8-18</td>
</tr>
<tr>
<td>Branching on the x87 FPU Condition Codes</td>
<td>8-20</td>
</tr>
<tr>
<td>Trigonometric Instructions</td>
<td>8-20</td>
</tr>
<tr>
<td>Approximation of Pi</td>
<td>8-21</td>
</tr>
<tr>
<td>Logarithmic, Exponential, and Scale</td>
<td>8-21</td>
</tr>
<tr>
<td>Transcendental Instruction Accuracy</td>
<td>8-21</td>
</tr>
<tr>
<td>x87 FPU Control Instructions</td>
<td>8-23</td>
</tr>
<tr>
<td>Waiting vs. Non-waiting Instructions</td>
<td>8-24</td>
</tr>
<tr>
<td>Unsupported x87 FPU Instructions</td>
<td>8-24</td>
</tr>
<tr>
<td>X87 FPU FLOATING-POINT EXCEPTION HANDLING</td>
<td>8-24</td>
</tr>
<tr>
<td>Arithmetic vs. Non-arithmetic Instructions</td>
<td>8-24</td>
</tr>
<tr>
<td>X87 FPU FLOATING-POINT EXCEPTION CONDITIONS</td>
<td>8-26</td>
</tr>
<tr>
<td>Invalid Operation Exception</td>
<td>8-26</td>
</tr>
<tr>
<td>Stack Overflow or Underflow Exception (#IS)</td>
<td>8-26</td>
</tr>
<tr>
<td>Invalid Arithmetic Operand Exception (#IA)</td>
<td>8-27</td>
</tr>
<tr>
<td>Denormal Operand Exception (#D)</td>
<td>8-28</td>
</tr>
<tr>
<td>Divide-By-Zero Exception (#Z)</td>
<td>8-28</td>
</tr>
<tr>
<td>Numeric Overflow Exception (#O)</td>
<td>8-28</td>
</tr>
</tbody>
</table>
CHAPTER 11
PROGRAMMING WITH INTEL® STREAMING SIMD EXTENSIONS 2 (INTEL® SSE2)

11.1 OVERVIEW OF INTEL® SSE2 ......................................................... 11-1
11.2 INTEL® SSE2 PROGRAMMING ENVIRONMENT .................................... 11-2
11.2.1 Intel® SSE2 in 64-Bit Mode and Compatibility Mode ............................ 11-3
11.2.2 Compatibility of Intel® SSE2 with Intel® SSE, MMX Technology, and x87 FPU Programming Environment .......................................................... 11-3
11.2.3 Denormals-Are-Zeros Flag ............................................................ 11-3
11.3 INTEL® SSE2 DATA TYPES ............................................................11-3
11.4 INTEL® SSE2 INSTRUCTIONS .......................................................11-4
11.4.1 Packed and Scalar Double Precision Floating-Point Instructions ............. 11-4
11.4.1.1 Data Movement Instructions .................................................... 11-5
11.4.1.2 Intel® SSE2 Arithmetic Instructions ......................................... 11-6
11.4.1.3 Intel® SSE2 Logical Instructions .............................................. 11-7
11.4.1.4 Intel® SSE2 Comparison Instructions ....................................... 11-7
11.4.1.5 Intel® SSE2 Shuffle and Unpack Instructions ................................. 11-7
11.4.1.6 Intel® SSE2 Conversion Instructions ........................................ 11-9
11.4.2 Intel® SSE2 64-Bit and 128-Bit SIMD Integer Instructions .................... 11-11
11.4.3 128-Bit SIMD Integer Instruction Extensions .................................... 11-11
11.4.4 Cacheability Control and Memory Ordering Instructions ....................... 11-12
11.4.4.1 FLUSH Cache Line ............................................................... 11-12
11.4.4.2 Cacheability Control Instructions .......................................... 11-12
11.4.4.3 Memory Ordering Instructions .............................................. 11-12
11.4.4.4 Pause .............................................................................. 11-12
11.4.5 Branch Hints .................................................................. 11-13
11.5 INTEL® SSE, SSE2, AND SSE3 EXCEPTIONS ...................................... 11-13
11.5.1 SIMD Floating-Point Exceptions ................................................... 11-13
11.5.2 SIMD Floating-Point Exception Conditions ....................................... 11-14
11.5.2.1 Invalid Operation Exception (#I) ........................................... 11-14
11.5.2.2 Denormal-Operand Exception (#D) ......................................... 11-15
11.5.2.3 Divide-By-Zero Exception (#Z) .............................................. 11-15
11.5.2.4 Numeric Overflow Exception (#O) ........................................ 11-15
11.5.2.5 Numeric Underflow Exception (#U) ....................................... 11-16
11.5.2.6 Inexact-Result (Precision) Exception (#P) ................................ 11-16
11.5.3 Generating SIMD Floating-Point Exceptions ..................................... 11-16
11.5.3.1 Handling Masked Exceptions ................................................ 11-17
11.5.3.2 Handling Unmasked Exceptions ............................................ 11-18
11.5.3.3 Handling Combinations of Masked and Unmasked Exceptions ......... 11-18
11.5.4 Handling SIMD Floating-Point Exceptions in Software ....................... 11-18
11.5.5 Interaction of SIMD and x87 FPU Floating-Point Exceptions ............... 11-18
11.6 WRITING APPLICATIONS WITH INTEL® SSE AND SSE2 ............... 11-19
11.6.1 General Guidelines for Using Intel® SSE and SSE2 ............................. 11-19
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.6.2</td>
<td>11-19</td>
</tr>
<tr>
<td>11.6.3</td>
<td>11-20</td>
</tr>
<tr>
<td>11.6.4</td>
<td>11-20</td>
</tr>
<tr>
<td>11.6.5</td>
<td>11-21</td>
</tr>
<tr>
<td>11.6.6</td>
<td>11-21</td>
</tr>
<tr>
<td>11.6.7</td>
<td>11-22</td>
</tr>
<tr>
<td>11.6.8</td>
<td>11-22</td>
</tr>
<tr>
<td>11.6.9</td>
<td>11-22</td>
</tr>
<tr>
<td>11.6.10</td>
<td>11-23</td>
</tr>
<tr>
<td>11.6.10.1</td>
<td>11-23</td>
</tr>
<tr>
<td>11.6.10.2</td>
<td>11-23</td>
</tr>
<tr>
<td>11.6.10.3</td>
<td>11-24</td>
</tr>
<tr>
<td>11.6.11</td>
<td>11-24</td>
</tr>
<tr>
<td>11.6.12</td>
<td>11-24</td>
</tr>
<tr>
<td>11.6.13</td>
<td>11-25</td>
</tr>
<tr>
<td>11.6.14</td>
<td>11-25</td>
</tr>
</tbody>
</table>

### PROGRAMMING WITH INTEL® SSE3, SSSE3, INTEL® SSE4, AND INTEL® AES-NI

12.1 PROGRAMMING ENVIRONMENT AND DATA TYPES
12.1.1 Intel® SSE3, SSSE3, and Intel® SSE4 in 64-Bit Mode and Compatibility Mode
12.1.2 Compatibility of Intel® SSE3 and SSSE3 with MMX Technology, the x87 FPU Environment, and Intel® SSE and SSE2
12.1.3 Horizontal and Asymmetric Processing.
12.2 OVERVIEW OF INTEL® SSE3 INSTRUCTIONS
12.3 INTEL® SSE3 INSTRUCTIONS
12.3.1 x87 FPU Instruction for Integer Conversion
12.3.2 SIMD Integer Instruction for Specialized 128-Bit Unaligned Data Load
12.3.3 SIMD Floating-Point Instructions That Enhance LOAD/MOVE/DUPLICATE Performance.
12.3.4 SIMD Floating-Point Instructions Provide Packed Addition/Subtraction
12.3.5 SIMD Floating-Point Instructions Provide Horizontal Addition/Subtraction
12.3.6 Two Thread Synchronization Instructions
12.4 WRITING APPLICATIONS WITH INTEL® SSE3
12.4.1 Guidelines for Using Intel® SSE3
12.4.2 Checking for Intel® SSE3 Support
12.4.3 Enable FTZ and DAZ for SIMD Floating-Point Computation
12.4.4 Programming Intel® SSE3 with Intel® SSE and SSE2
12.5 OVERVIEW OF SSSE3 INSTRUCTIONS
12.6 SSSE3 INSTRUCTIONS
12.6.1 Horizontal Addition/Subtraction
12.6.2 Packed Absolute Values
12.6.3 Multiply and Add Packed Signed and Unsigned Bytes
12.6.4 Packed Multiply High with Round and Scale
12.6.5 Packed Shuffle Bytes
12.6.6 Packed Sign
12.6.7 Packed Align Right
12.7 WRITING APPLICATIONS WITH SSSE3 EXTENSIONS
12.7.1 Guidelines for Using SSSE3
12.7.2 Checking for SSSE3 Support
12.8 INTEL® SSE3, SSSE3, AND INTEL® SSE4 EXCEPTIONS
12.8.1 Device Not Available (DNA) Exceptions.
12.8.2 Numeric Error Flag and IGNNE#.
12.8.3 Emulation
12.8.4 IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions
12.9 INTEL® SSE4 OVERVIEW
12.10 INTEL® SSE4.1 INSTRUCTION SET
12.10.1 Dword Multiply Instructions
12.10.2 Floating-Point Dot Product Instructions
12.10.3 Streaming Load Hint Instruction
12.10.4 Packed Blending Instructions
CHAPTER 13
MANAGING STATE USING THE XSAVE FEATURE SET
13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS .................................. 13-1
13.2 ENUMERATION OF CPU SUPPORT FOR XSAVE INSTRUCTIONS AND XSAVE-SUPPORTED FEATURES ........................................ 13-3
13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES ................ 13-5
13.4 XSAVE AREA ......................................................................................................................... 13-7
13.4.1 Legacy Region of an XSAVE Area ...................................................................................... 13-8
13.4.2 XSAVE Header ................................................................................................................. 13-9
13.4.3 Extended Region of an XSAVE Area ................................................................................. 13-9
13.5 XSAVE-MANAGED STATE ................................................................................................. 13-10
13.5.1 x87 State ......................................................................................................................... 13-10
13.5.2 SSE State ......................................................................................................................... 13-11
13.5.3 AVX State ......................................................................................................................... 13-11
13.5.4 MPX State ......................................................................................................................... 13-11
13.5.5 AVX-512 State ................................................................................................................ 13-12
13.5.6 PT State .......................................................................................................................... 13-13
13.5.7 PKRU State ...................................................................................................................... 13-13
13.5.8 PASID State ...................................................................................................................... 13-14
13.5.9 CET State ........................................................................................................................ 13-14
13.5.10 HDC State ....................................................................................................................... 13-14
13.5.11 UINTR State ................................................................................................................... 13-15
13.5.12 LBR State ....................................................................................................................... 13-16
13.5.13 HwP State ...................................................................................................................... 13-17
13.5.14 AMX State ...................................................................................................................... 13-17
13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE ...................................................... 13-18
13.7 OPERATION OF XSAVE ...................................................................................................... 13-19
13.8 OPERATION OF XRSTOR ..................................................................................................... 13-20
13.8.1 Standard Form of XRSTOR .............................................................................................. 13-20
13.8.2 Compacted Form of XRSTOR ......................................................................................... 13-21
13.8.3 XRSTOR and the Init and Modified Optimizations .......................................................... 13-21
13.9 OPERATION OF XSAVEOPT ............................................................................................... 13-22
13.10 OPERATION OF XSAVEC .................................................................................................. 13-23
13.11 OPERATION OF XSAVE..................................................................................................... 13-24
13.12 OPERATION OF XRSTORS ................................................................................................. 13-25
13.13 MEMORY ACCESSES BY THE XSAVE FEATURE SET ...................................................... 13-27
CONTENTS

CHAPTER 14
PROGRAMMING WITH INTEL® AVX, FMA, AND INTEL® AVX2
14.1 INTEL® AVX OVERVIEW ................................................................. 14-1
14.1.1 256-Bit Wide SIMD Register Support ....................................... 14-1
14.1.2 Instruction Syntax Enhancements .............................................. 14-2
14.1.3 VEX Prefix Instruction Encoding Support ................................... 14-2
14.2 FUNCTIONAL OVERVIEW .............................................................. 14-3
14.2.1 256-Bit Floating-Point Arithmetic Processing Enhancements ......... 14-9
14.2.2 256-Bit Non-Arithmetic Instruction Enhancements ....................... 14-9
14.2.3 Arithmetic Primitives for 128-Bit Vector and Scalar processing .... 14-11
14.2.4 Non-Arithmetic Primitives for 128-Bit Vector and Scalar Processing 14-13
14.3 DETECTION OF INTEL® AVX INSTRUCTIONS .............................. 14-15
14.3.1 Detection of VEX-Encoded AES and VPCLMULQDQ ....................... 14-17
14.4 HALF PRECISION FLOATING-POINT CONVERSION ......................... 14-18
14.4.1 Detection of F16C Instructions ................................................ 14-20
14.5 FUSED-MULTIPLY-ADD (FMA) EXTENSIONS .................................. 14-21
14.5.1 FMA Instruction Operand Order and Arithmetic Behavior ............ 14-22
14.5.2 Fused-Multiply-ADD (FMA) Numeric Behavior .......................... 14-22
14.5.3 Detection of FMA ................................................................. 14-24
14.6 OVERVIEW OF INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2) 14-25
14.6.1 Intel® AVX2 and 256-Bit Vector Integer Processing ....................... 14-25
14.7 PROMOTED VECTOR INTEGER INSTRUCTIONS IN INTEL® AVX2 ........ 14-26
14.7.1 Detection of Intel® AVX2 ...................................................... 14-31
14.8 ACCESSING YMM REGISTERS ........................................................ 14-32
14.9 MEMORY ALIGNMENT .................................................................. 14-32
14.10 SIMD FLOATING-POINT EXCEPTIONS ......................................... 14-34
14.11 EMULATION .............................................................................. 14-34
14.12 WRITING INTEL® AVX FLOATING-POINT EXCEPTION HANDLERS .... 14-34
14.13 GENERAL PURPOSE INSTRUCTION SET ENHANCEMENTS ............ 14-35

CHAPTER 15
PROGRAMMING WITH INTEL® AVX-512
15.1 OVERVIEW .................................................................................. 15-1
15.1.1 512-Bit Wide SIMD Register Support ....................................... 15-1
15.1.2 32 SIMD Register Support ...................................................... 15-1
15.1.3 Eight Opmask Register Support ............................................... 15-1
15.1.4 Instruction Syntax Enhancement .............................................. 15-2
15.1.5 EVEX Instruction Encoding Support ........................................ 15-2
15.2 DETECTION OF AVX-512 FOUNDATION INSTRUCTIONS ............ 15-3
15.2.1 Additional 512-Bit Instruction Extensions of the Intel® AVX-512 Family 15-4
15.2.2 Detection of AVX512-FP16 Instructions ...................................... 15-5
15.3 DETECTION OF 512-BIT INSTRUCTION GROUPS OF THE INTEL® AVX-512 FAMILY .................................................. 15-6
15.4 DETECTION OF INTEL® AVX-512 INSTRUCTION GROUPS OPERATING AT 256 AND 128-BIT VECTOR LENGTHS ...................... 15-7
15.5 ACCESSING XMM, YMM, AND ZMM REGISTERS .......................... 15-8
15.6 ENHANCED VECTOR PROGRAMMING ENVIRONMENT USING EVEX ENCODING ................................................................. 15-8
15.6.1 OPMASK Register to Predicate Vector Data Processing .............. 15-9
15.6.1.1 Opmask Register K0 ...................................................... 15-9
15.6.1.2 Example of Opmask Usages ............................................ 15-10
15.6.2 OpMask Instructions ............................................................. 15-11
15.6.3 Broadcast .............................................................................. 15-11
15.6.4 Static Rounding Mode and Suppress All Exceptions .................. 15-12
15.6.5 Compressed Disp8*N Encoding .............................................. 15-13
15.7 MEMORY ALIGNMENT ............................................................... 15-13
15.8 SIMD FLOATING-POINT EXCEPTIONS .......................................... 15-14
15.9 INSTRUCTION EXCEPTION SPECIFICATION ............................... 15-15
15.10 EMULATION .............................................................................. 15-15
15.11 WRITING FLOATING-POINT EXCEPTION HANDLERS ................. 15-15

xlv Vol. 1
CHAPTER 16
PROGRAMMING WITH INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

16.1 OVERVIEW ................................................................. 16-1
16.2 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS ................................................. 16-1
  16.2.1 HLE Software Interface .................................................. 16-2
  16.2.2 RTM Software Interface ............................................... 16-3
16.3 INTEL® TSX APPLICATION PROGRAMMING MODEL .................................................. 16-3
  16.3.1 Detection of Transactional Synchronization Support .................. 16-3
    16.3.1.1 Detection of HLE Support ........................................... 16-3
    16.3.1.2 Detection of RTM Support .......................................... 16-3
    16.3.1.3 Detection of XTEST Instruction ................................. 16-4
    16.3.1.4 Detection of Intel® TSX Suspend Load Address Tracking .......... 16-4
  16.3.2 Querying Transactional Execution Status ................................ 16-4
  16.3.3 Requirements for HLE Locks ........................................ 16-4
    16.3.4 Transactional Nesting ................................................ 16-4
    16.3.4.1 HLE Nesting and Elision .......................................... 16-4
    16.3.4.2 RTM Nesting .......................................................... 16-5
    16.3.4.3 Nesting HLE and RTM .............................................. 16-5
  16.3.5 RTM Abort Status Definition ........................................ 16-5
  16.3.6 RTM Memory Ordering ................................................ 16-6
  16.3.7 RTM-Enabled Debugger Support ...................................... 16-6
  16.3.8 Intel® TSX Suspend/Resume Load Address Tracking Support .......... 16-6
  16.3.9 Programming Considerations .......................................... 16-6
    16.3.9.1 Instruction Based Considerations ............................... 16-7
    16.3.9.2 Runtime Considerations ........................................... 16-8
  16.3.10 RTM Nesting .......................................................... 16-9
  16.3.11 RTM Memory Ordering ................................................ 16-9
  16.3.12 RTM-Enabled Debugger Support ..................................... 16-9
  16.3.13 Detection of Intel® TSX Suspend Load Address Tracking .......... 16-9
  16.3.14 Detection of Intel® TSX Suspend Load Address Tracking .......... 16-9
  16.3.15 Programming Considerations .......................................... 16-9

CHAPTER 17
CONTROL-FLOW ENFORCEMENT TECHNOLOGY (CET)

17.1 INTRODUCTION ............................................................... 17-1
  17.1.1 Shadow Stack .......................................................... 17-1
  17.1.2 Indirect Branch Tracking ............................................. 17-1
  17.1.3 Speculative Behavior when CET is Enabled .......................... 17-2
  17.2 SHADOW STACKS .......................................................... 17-2
    17.2.1 Shadow Stack Pointer and its Operand and Address Size Attributes .... 17-2
    17.2.2 Terminology .......................................................... 17-2
    17.2.3 Supervisor Shadow Stack Token .................................... 17-3
    17.2.4 Shadow Stack Usage on Task Switch ............................... 17-5
    17.2.5 Switching Shadow Stacks ............................................ 17-5
    17.2.6 Constraining Execution at Targets of RET ........................ 17-7
  17.3 INDIRECT BRANCH TRACKING ............................................. 17-7
    17.3.1 No-track Prefix for Near Indirect CALL/JMP ........................ 17-8
    17.3.2 Terminology .......................................................... 17-9
    17.3.3 Indirect Branch Tracking ............................................ 17-9
      17.3.3.1 Control Transfers between CPL 3 and CPL < 3 ..................... 17-10
      17.3.3.2 Control Transfers within CPL 3 or CPL < 3 ....................... 17-10
    17.3.4 Indirect Branch Tracking State Machine ........................... 17-11
    17.3.5 INT3 Treatment ..................................................... 17-12
    17.3.6 Legacy Compatibility Treatment .................................... 17-12
      17.3.6.1 Legacy Code Page Bitmap Format ................................ 17-13
    17.3.7 Other Considerations ................................................ 17-13
      17.3.7.1 Intel® Transactional Synchronization Extensions (Intel® TSX) Interactions . 17-13
      17.3.7.2 #CP(ENDBRANCH) Priority w.r.t #NM and #UD .................. 17-13
      17.3.7.3 #CP(ENDBRANCH) Priority w.r.t #BP and #DB ................... 17-13
      17.3.8 Constraining Speculation after Missing ENDBRANCH .............. 17-14
  17.4 INTEL® TSX APPLICATION PROGRAMMING MODEL .................................................. 17-14

CHAPTER 18
PROGRAMMING WITH INTEL® ADVANCED MATRIX EXTENSIONS

18.1 INTRODUCTION ............................................................... 18-1
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1.1</td>
</tr>
<tr>
<td>18.1.2</td>
</tr>
<tr>
<td>18.1.3</td>
</tr>
<tr>
<td>18.1.4</td>
</tr>
<tr>
<td>18.2</td>
</tr>
<tr>
<td>18.3</td>
</tr>
<tr>
<td>18.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT/OUTPUT</td>
</tr>
<tr>
<td>19.1</td>
</tr>
<tr>
<td>19.2</td>
</tr>
<tr>
<td>19.3</td>
</tr>
<tr>
<td>19.3.1</td>
</tr>
<tr>
<td>19.4</td>
</tr>
<tr>
<td>19.5</td>
</tr>
<tr>
<td>19.5.1</td>
</tr>
<tr>
<td>19.5.2</td>
</tr>
<tr>
<td>19.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESSOR IDENTIFICATION AND FEATURE DETERMINATION</td>
</tr>
<tr>
<td>20.1</td>
</tr>
<tr>
<td>20.1.1</td>
</tr>
<tr>
<td>20.1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDIX A</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS CROSS-REFERENCE</td>
</tr>
<tr>
<td>A.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDIX B</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS CONDITION CODES</td>
</tr>
<tr>
<td>B.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDIX C</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOATING-POINT EXCEPTIONS SUMMARY</td>
</tr>
<tr>
<td>C.1</td>
</tr>
<tr>
<td>C.2</td>
</tr>
<tr>
<td>C.3</td>
</tr>
<tr>
<td>C.4</td>
</tr>
<tr>
<td>C.5</td>
</tr>
<tr>
<td>C.6</td>
</tr>
<tr>
<td>C.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDIX D</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS</td>
</tr>
<tr>
<td>D.1</td>
</tr>
<tr>
<td>D.2</td>
</tr>
<tr>
<td>D.3</td>
</tr>
<tr>
<td>D.4</td>
</tr>
<tr>
<td>D.4.1</td>
</tr>
<tr>
<td>D.4.2</td>
</tr>
<tr>
<td>D.4.2.1</td>
</tr>
<tr>
<td>D.4.2.2</td>
</tr>
<tr>
<td>D.4.2.3</td>
</tr>
<tr>
<td>D.4.3</td>
</tr>
</tbody>
</table>
## APPENDIX E
### INTEL® MEMORY PROTECTION EXTENSIONS

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1</td>
</tr>
<tr>
<td>E.2</td>
</tr>
<tr>
<td>E.3</td>
</tr>
<tr>
<td>E.3.1</td>
</tr>
<tr>
<td>E.3.2</td>
</tr>
<tr>
<td>E.3.3</td>
</tr>
<tr>
<td>E.3.4</td>
</tr>
<tr>
<td>E.4</td>
</tr>
<tr>
<td>E.4.1</td>
</tr>
<tr>
<td>E.4.2</td>
</tr>
<tr>
<td>E.4.3</td>
</tr>
<tr>
<td>E.4.3.1</td>
</tr>
<tr>
<td>E.4.3.2</td>
</tr>
<tr>
<td>E.5</td>
</tr>
<tr>
<td>E.5.1</td>
</tr>
<tr>
<td>E.5.2</td>
</tr>
<tr>
<td>E.5.3</td>
</tr>
<tr>
<td>E.5.4</td>
</tr>
<tr>
<td>E.5.5</td>
</tr>
<tr>
<td>E.5.6</td>
</tr>
<tr>
<td>E.5.7</td>
</tr>
<tr>
<td>E.5.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-1</td>
</tr>
<tr>
<td>E-1</td>
</tr>
<tr>
<td>E-2</td>
</tr>
<tr>
<td>E-2</td>
</tr>
<tr>
<td>E-2</td>
</tr>
<tr>
<td>E-3</td>
</tr>
<tr>
<td>E-4</td>
</tr>
<tr>
<td>E-4</td>
</tr>
<tr>
<td>E-5</td>
</tr>
<tr>
<td>E-5</td>
</tr>
<tr>
<td>E-6</td>
</tr>
<tr>
<td>E-7</td>
</tr>
<tr>
<td>E-8</td>
</tr>
<tr>
<td>E-9</td>
</tr>
<tr>
<td>E-9</td>
</tr>
<tr>
<td>E-10</td>
</tr>
<tr>
<td>E-10</td>
</tr>
<tr>
<td>E-11</td>
</tr>
<tr>
<td>E-11</td>
</tr>
<tr>
<td>E-11</td>
</tr>
<tr>
<td>E-11</td>
</tr>
<tr>
<td>E-12</td>
</tr>
</tbody>
</table>
Figure 1-1. Syntax for CPUID, CR, and MSR Data Presentation .......................... 1-6
Figure 1-2. Bit and Byte Order ............................................................................. 1-8
Figure 2-1. The P6 Processor Microarchitecture with Advanced Transfer Cache Enhancement .......................................................... 2-7
Figure 2-2. The Intel NetBurst® Microarchitecture .............................................. 2-9
Figure 2-3. The Intel® Core™ Microarchitecture Pipeline Functionality ............ 2-11
Figure 2-4. SIMD Extensions, Register Layouts, and Data Types .................. 2-16
Figure 2-5. Comparison of an IA-32 Processor Supporting Intel® Hyper-Threading Technology and a Traditional Dual Processor System .................................................................................................................. 2-17
Figure 2-6. Intel 64 and IA-32 Processors that Support Dual-Core ................... 2-18
Figure 2-7. Intel® 64 Processors that Support Quad-Core ................................. 2-19
Figure 2-8. Intel® Core ™ i7 Processor ................................................................. 2-19
Figure 3-1. IA-32 Basic Execution Environment for Non-64-Bit Modes ......... 3-3
Figure 3-2. 64-Bit Mode Execution Environment .............................................. 3-6
Figure 3-3. Three Memory Management Models ............................................. 3-8
Figure 3-4. General System and Application Programming Registers ............. 3-11
Figure 3-5. Alternate General-Purpose Register Names ................................ 3-12
Figure 3-6. Use of Segment Registers for Flat Memory Model ......................... 3-14
Figure 3-7. Use of Segment Registers in Segmented Memory Model ............. 3-14
Figure 3-8. EFLAGS Register ............................................................................. 3-16
Figure 3-9. Memory Operand Address ............................................................. 3-21
Figure 3-10. Memory Operand Address in 64-Bit Mode ................................. 3-21
Figure 3-11. Offset (or Effective Address) Computation ................................ 3-23
Figure 4-1. Fundamental Data Types ............................................................... 4-1
Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory .......................................................... 4-2
Figure 4-3. Numeric Data Types ...................................................................... 4-3
Figure 4-4. Pointer Data Types ........................................................................ 4-6
Figure 4-5. Pointers in 64-Bit Mode ................................................................. 4-7
Figure 4-6. Bit Field Data Type ........................................................................ 4-7
Figure 4-7. 64-Bit Packed SIMD Data Types ................................................... 4-8
Figure 4-8. 128-Bit Packed SIMD Data Types ................................................ 4-9
Figure 4-9. BCD Data Types ........................................................................... 4-10
Figure 4-10. Binary Real Number System ..................................................... 4-12
Figure 4-11. Binary Floating-Point Format .................................................... 4-12
Figure 4-12. Real Numbers and NaNs ............................................................. 4-14
Figure 6-1. Stack Structure .............................................................................. 6-2
Figure 6-2. Stack on Near and Far Calls ......................................................... 6-6
Figure 6-3. Shadow Stack on Near and Far Calls .......................................... 6-6
Figure 6-4. Protection Rings ........................................................................... 6-8
Figure 6-5. Stack Switch on a Call to a Different Privilege Level ..................... 6-9
Figure 6-6. Shadow Stack Switch on a Call to a Different Privilege Level ........ 6-10
Figure 6-7. Stack Usage on Transfers to Interrupt and Exception Handling Routines ................................................................................................................. 6-15
Figure 6-8. Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines ................................................................................................................. 6-16
Figure 6-9. Nested Procedures .................................................................... 6-21
Figure 7-1. Operation of the PUSH Instruction .............................................. 7-5
Figure 7-2. Operation of the PUSHA Instruction ............................................. 7-6
Figure 7-3. Operation of the POP Instruction ................................................ 7-6
Figure 7-4. Operation of the POPA Instruction .............................................. 7-7
Figure 7-5. Sign Extension ............................................................................. 7-7
Figure 7-6. SHL/SAH Instruction Operation .................................................... 7-11
Figure 7-7. SHR Instruction Operation ........................................................... 7-11
Figure 7-8. SAR Instruction Operation ........................................................... 7-12
Figure 7-9. SHLD and SHRD Instruction Operations .................................... 7-12
Figure 7-10. ROL, ROR, RCL, and RCR Instruction Operations ................... 7-13
Figure E-1. Layout of the Bounds Registers BND0-BND3 ................................................................. E-3
Figure E-2. Common Layout of the Bound Configuration Registers BNDCFGU and BNDCFGS .......................... E-3
Figure E-3. Layout of the Bound Status Registers BNDSTATUS .................................................... E-4
Figure E-4. Bound Paging Structure and Address Translation in 64-Bit Mode ........................................ E-7
Figure E-5. Bound Paging Structure and Address Translation Outside 64-Bit Mode ................................. E-9
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Key Features of Most Recent IA-32 Processors</td>
<td>2-21</td>
</tr>
<tr>
<td>2-2</td>
<td>Key Features of Most Recent Intel® 64 Processors</td>
<td>2-21</td>
</tr>
<tr>
<td>2-3</td>
<td>Key Features of Previous Generations of IA-32 Processors</td>
<td>2-27</td>
</tr>
<tr>
<td>2-4</td>
<td>Planned Intel® ISA and Features Removal List</td>
<td>2-28</td>
</tr>
<tr>
<td>2-5</td>
<td>Intel® ISA and Features Removal List</td>
<td>2-28</td>
</tr>
<tr>
<td>3-1</td>
<td>Instruction Pointer Sizes</td>
<td>3-10</td>
</tr>
<tr>
<td>3-2</td>
<td>Addressable General Purpose Registers</td>
<td>3-13</td>
</tr>
<tr>
<td>3-3</td>
<td>Effective Operand- and Address-Size Attributes</td>
<td>3-19</td>
</tr>
<tr>
<td>3-4</td>
<td>Effective Operand- and Address-Size Attributes in 64-Bit Mode</td>
<td>3-19</td>
</tr>
<tr>
<td>3-5</td>
<td>Default Segment Selection Rules</td>
<td>3-22</td>
</tr>
<tr>
<td>4-1</td>
<td>Signed Integer Encodings</td>
<td>4-4</td>
</tr>
<tr>
<td>4-2</td>
<td>Length, Precision, and Range of Floating-Point Data Types</td>
<td>4-5</td>
</tr>
<tr>
<td>4-3</td>
<td>Floating-Point Number and NaN Encodings</td>
<td>4-5</td>
</tr>
<tr>
<td>4-4</td>
<td>Packed Decimal Integer Encodings</td>
<td>4-10</td>
</tr>
<tr>
<td>4-5</td>
<td>Real and Floating-Point Number Notation</td>
<td>4-12</td>
</tr>
<tr>
<td>4-6</td>
<td>Denormalization Process</td>
<td>4-15</td>
</tr>
<tr>
<td>4-7</td>
<td>Rules for Handling NaNs</td>
<td>4-16</td>
</tr>
<tr>
<td>4-8</td>
<td>Rounding Modes and Encoding of Rounding Control (RC) Field</td>
<td>4-18</td>
</tr>
<tr>
<td>4-9</td>
<td>Numeric Overflow Thresholds</td>
<td>4-21</td>
</tr>
<tr>
<td>4-10</td>
<td>Masked Responses to Numeric Overflow</td>
<td>4-21</td>
</tr>
<tr>
<td>4-11</td>
<td>Numeric Underflow (Normalized) Thresholds</td>
<td>4-22</td>
</tr>
<tr>
<td>5-1</td>
<td>Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors</td>
<td>5-2</td>
</tr>
<tr>
<td>5-2</td>
<td>Instruction Sets in Intel® 64 and IA-32 Processors</td>
<td>5-2</td>
</tr>
<tr>
<td>5-3</td>
<td>Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX</td>
<td>5-39</td>
</tr>
<tr>
<td>6-1</td>
<td>Exceptions and Interrupts</td>
<td>6-13</td>
</tr>
<tr>
<td>7-1</td>
<td>Move Instruction Operations</td>
<td>7-3</td>
</tr>
<tr>
<td>7-2</td>
<td>Conditional Move Instructions</td>
<td>7-4</td>
</tr>
<tr>
<td>7-3</td>
<td>Bit Test and Modify Instructions</td>
<td>7-14</td>
</tr>
<tr>
<td>7-4</td>
<td>Conditional Jump Instructions</td>
<td>7-16</td>
</tr>
<tr>
<td>8-1</td>
<td>Condition Code Interpretation</td>
<td>8-5</td>
</tr>
<tr>
<td>8-2</td>
<td>Precision Control Field (PC)</td>
<td>8-8</td>
</tr>
<tr>
<td>8-3</td>
<td>Unsupported Double Extended Precision Floating-Point Encodings and Pseudo-Denormals</td>
<td>8-14</td>
</tr>
<tr>
<td>8-4</td>
<td>Data Transfer Instructions</td>
<td>8-16</td>
</tr>
<tr>
<td>8-5</td>
<td>Floating-Point Conditional Move Instructions</td>
<td>8-16</td>
</tr>
<tr>
<td>8-6</td>
<td>Setting of x87 FPU Condition Code Flags for Floating-Point Number Comparisons</td>
<td>8-19</td>
</tr>
<tr>
<td>8-7</td>
<td>Setting of EFLAGS Status Flags for Floating-Point Number Comparisons</td>
<td>8-19</td>
</tr>
<tr>
<td>8-8</td>
<td>TEST Instruction Constants for Conditional Branching</td>
<td>8-20</td>
</tr>
<tr>
<td>8-9</td>
<td>Arithmetic and Non-arithmetic Instructions</td>
<td>8-25</td>
</tr>
<tr>
<td>8-10</td>
<td>Invalid Arithmetic Operations and the Masked Responses to Them</td>
<td>8-27</td>
</tr>
<tr>
<td>8-11</td>
<td>Divide-By-Zero Conditions and the Masked Responses to Them</td>
<td>8-28</td>
</tr>
<tr>
<td>9-1</td>
<td>Data Range Limits for Saturation</td>
<td>9-5</td>
</tr>
<tr>
<td>9-2</td>
<td>MMX Instruction Set Summary</td>
<td>9-6</td>
</tr>
<tr>
<td>9-3</td>
<td>Effect of Prefixes on MMX Instructions</td>
<td>9-11</td>
</tr>
<tr>
<td>10-1</td>
<td>PREFETCHh Instructions Caching Hints</td>
<td>10-13</td>
</tr>
<tr>
<td>10-2</td>
<td>Format of an FXSAVE Area</td>
<td>10-15</td>
</tr>
<tr>
<td>11-1</td>
<td>Masked Responses of Intel® SSE, SSE2, and SSE3 Instructions to Invalid Arithmetic Operations</td>
<td>11-14</td>
</tr>
<tr>
<td>11-2</td>
<td>SSE and SSE2 State Following a Power-up/Reset or INIT</td>
<td>11-20</td>
</tr>
<tr>
<td>11-3</td>
<td>Effect of Prefixes on the Intel® SSE, SSE2, and SSE3 Instructions</td>
<td>11-26</td>
</tr>
<tr>
<td>12-1</td>
<td>SIMD Numeric Exceptions Signaled by SSE4.1</td>
<td>12-10</td>
</tr>
<tr>
<td>12-2</td>
<td>Enhanced 32-Bit SIMD Multiply Supported by Intel® SSE4.1</td>
<td>12-11</td>
</tr>
<tr>
<td>12-3</td>
<td>Blend Field Size and Control Modes Supported by Intel® SSE4.1</td>
<td>12-14</td>
</tr>
<tr>
<td>12-4</td>
<td>Enhanced SIMD Integer MIN/MAX Instructions Supported by Intel® SSE4.1</td>
<td>12-14</td>
</tr>
<tr>
<td>12-5</td>
<td>New SIMD Integer Conversions Supported by Intel® SSE4.1</td>
<td>12-15</td>
</tr>
<tr>
<td>12-6</td>
<td>New SIMD Integer Conversions Supported by Intel® SSE4.1</td>
<td>12-16</td>
</tr>
<tr>
<td>12-7</td>
<td>Enhanced SIMD Pack Support by Intel® SSE4.1</td>
<td>12-17</td>
</tr>
<tr>
<td>12-8</td>
<td>Byte and 32-Bit Word Representation of a 128-Bit State</td>
<td>12-20</td>
</tr>
<tr>
<td>12-9</td>
<td>Matrix Representation of a 128-Bit State</td>
<td>12-20</td>
</tr>
</tbody>
</table>
Table 15-3. Instruction Mnemonics That Do Not Support EVEX.128 Encoding.  
Table 14-21. Alignment Faulting Conditions when Memory Access is Not Aligned.  
Table 14-20. New Primitive in Intel® AVX2 Instructions.  
Table 14-19. VEX-Only SIMD Instructions in Intel® AVX and AVX2.  
Table 14-18. Promoted Vector Integer SIMD Instructions in Intel® AVX2.  
Table 14-17. FMA Numeric Behavior.  
Table 14-16. Rounding Behavior of Zero Result in FMA Operation.  
Table 14-15. 128-Bit Intel® AVX Instruction Enhancement.  
Table 14-14. 256-Bit Intel® AVX Instruction Enhancements.  
Table 14-13. Promoted 256-Bit and 128-Bit Arithmetic Intel® AVX Instructions.  
Table 14-12. Promoted SSSE3 and Intel® SSE, SSE2, SSE3, and SSE4 Instructions.  
Table 14-10. Invalid Operation for VCVTPH2PS and VCVTPS2PH.  
Table 14-9. Non-Numerical Behavior for VCVTPH2PS and VCVTPS2PH.  
Table 14-7. Promotion of Legacy SIMD ISA to 128-Bit Non-Arithmetic Intel® AVX instruction.  
Table 14-6. 128-Bit Intel® AVX Instruction Enhancement.  
Table 14-5. Promotion of Legacy SIMD ISA to 128-Bit Arithmetic Intel® AVX Instructions.  
Table 14-4. 256-Bit Intel® AVX Instruction Enhancements.  
Table 14-3. Promoted 256-Bit and 128-Bit Data Movement Intel® AVX Instructions.  
Table 14-2. Promoted Vector Integer SIMD Instructions in Intel® AVX2.  
Table 12-12. The ShiftRows Transformation.  
Table 12-11. Little Endian Representation of a 4x4 Byte Matrix.  
Table 12-10. Little Endian Representation of a 128-Bit State.  
Table 12-14. The InvShiftRows Transformation.  
Table 12-13. Look-up Table Associated with S-Box Transformation.  
Table 12-12. The ShiftRows Transformation.  
Table 12-11. Format of the Legacy Region of an XSAVE Area.  
Table 12-10. Promoted SSSE3 and Intel® SSE, SSE2, SSE3, and SSE4 Instructions.  
Table 12-9. Promoted 256-Bit and 128-Bit Arithmetic Intel® AVX Instructions.  
Table 12-8. Promoted Vector Integer SIMD Instructions in Intel® AVX2.  
Table 12-7. VEX-Only SIMD Instructions in Intel® AVX and AVX2.  
Table 12-6. New Primitive in Intel® AVX2 Instructions.  
Table 12-5. Instructions Not Requiring Explicit Memory Alignment.  
Table 12-4. Instructions Requiring Explicitly Aligned Memory.  
Table 12-3. Instructions Not Requiring Explicit Memory Alignment.  
Table 12-2. 512-Bit Instruction Groups in the Intel® AVX-512 Family.  
Table 12-1. Feature Flag Collection Required of 256/128 Bit Vector Lengths for Each Instruction Group.  
Table 19-1. I/O Instruction Serialization.  
Table 17-1. Indirect Branch Tracking State Machine.  
Table 16-1. RTM Abort Status Definition.  
Table 15-9. Static Rounding Mode.  
Table 15-8. Characteristics of Three Rounding Control Interfaces.  
Table 15-7. Instructions Not Requiring Explicit Memory Alignment.  
Table 15-6. SIMD Instructions Requiring Explicitly Aligned Memory.  
Table 15-5. Static Rounding Mode.  
Table 15-4. Instruction Mnemonics That Do Not Support EVEX.128 Encoding.  
Table 15-3. Instruction Mnemonics That Do Not Support EVEX.128 Encoding.  
Table 15-2. Alignment Faulting Conditions when Memory Access is Not Aligned.  
Table 15-1. Little Endian Representation of a 128-Bit State.  
Table 14-11. Denormal Condition Summary.  
Table 14-10. Invalid Operation for VCVTPH2PS and VCVTPS2PH.  
Table 14-9. Non-Numerical Behavior for VCVTPH2PS and VCVTPS2PH.  
Table 14-8. Immediate Byte Encoding for 16-Bit Floating-Point Conversion Instructions.  
Table 14-7. Promotion of Legacy SIMD ISA to 128-Bit Non-Arithmetic Intel® AVX instruction.  
Table 14-6. 128-Bit Intel® AVX Instruction Enhancement.  
Table 14-5. Promotion of Legacy SIMD ISA to 128-Bit Arithmetic Intel® AVX Instructions.  
Table 14-4. 256-Bit Intel® AVX Instruction Enhancements.  
Table 14-3. Promoted 256-Bit and 128-Bit Data Movement Intel® AVX Instructions.  
Table 14-2. Promoted Vector Integer SIMD Instructions in Intel® AVX2.  
Table 14-1. Promoted SSSE3 and Intel® SSE, SSE2, SSE3, and SSE4 Instructions.
| Table D-9. | CVTTPS2DQ, CVTPD2DQ, and CVTTPD2DQ | D-7 |
| Table D-10. | MAXPS, MAXSS, MINPS, MINSS, MAXPD, MAXSD, MINPD, and MINSD | D-8 |
| Table D-11. | SQRTPS, SQRTSS, SQRTPD, and SQRTSD | D-8 |
| Table D-12. | CVTPS2PD and CVTSS2SD | D-8 |
| Table D-13. | CVTPD2PS and CVTSD2SS | D-8 |
| Table D-14. | #I - Invalid Operations | D-9 |
| Table D-15. | #D - Denormal Operand | D-11 |
| Table D-16. | #Z - Divide-by-Zero | D-12 |
| Table D-17. | #O - Numeric Overflow | D-13 |
| Table D-18. | #U - Numeric Underflow | D-14 |
| Table E-1. | Error Code Definition of BNDSTATUS | E-4 |
| Table E-2. | Intel MPX Instruction Summary | E-5 |
| Table E-3. | Effective Address Size of Intel® MPX Instructions with 67H Prefix | E-10 |
| Table E-4. | Bounds Register INIT Behavior Due to BND Prefix with Branch Instructions | E-11 |
The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture (order number 253665) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4: Model-Specific Registers (order number 335592).

The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C, & 3D, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, addresses the programming environment for classes of software that host operating systems. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, describes the model-specific registers of Intel 64 and IA-32 processors.

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™ 2 Duo processor
- Intel® Core™ 2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™ 2 Extreme processor X7000 and X6800 series
- Intel® Core™ 2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
• Intel® Pentium® Dual-Core processor
• Intel® Xeon® processor 7200, 7300 series
• Intel® Xeon® processor 5200, 5400, 7400 series
• Intel® Core™ 2 Extreme processor QX9000 and X9000 series
• Intel® Core™ 2 Quad processor Q9000 series
• Intel® Core™ 2 Duo processor E8000, T9000 series
• Intel Atom® processor family
• Intel Atom® processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
• Intel® Core™ i7 processor
• Intel® Core™ i5 processor
• Intel® Xeon® processor E7-8800/4800/2800 product families
• Intel® Core™ i7-39xx Processor Extreme Edition
• Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
• Intel® Xeon® processor E3-1200 product family
• Intel® Xeon® processor E5-2400/1400 product family
• Intel® Xeon® processor E5-4600/2600/1600 product family
• Intel® Xeon® processor E3-1200 v2 product family
• Intel® Xeon® processor E5-2400/1400 v2 product families
• Intel® Xeon® processor E5-4600/2600/1600 v2 product families
• Intel® Xeon® processor E7-8800/4800/2800 v2 product families
• 3rd generation Intel® Core™ processors
• Intel® Xeon® processor E3-1200 v3 product family
• Intel® Xeon® processor E5-2600/1600 v3 product families
• 4th generation Intel® Core™ processors
• Intel® Xeon® processor E5 v4 family
• The Intel® Core™ M processor family
• Intel® Core™ i7-59xx Processor Extreme Edition
• Intel® Core™ i7-49xx Processor Extreme Edition
• Intel® Xeon® processor E3-1200 v3 product family
• Intel® Xeon® processor E5-2600/1600 v3 product families
• 5th generation Intel® Core™ processors
• Intel® Xeon® processor D-1500 product family
• Intel® Xeon® processor E5 v4 family
• Intel Atom® processor X7-Z8000 and X5-Z8000 series
• Intel Atom® processor Z3400 series
• Intel Atom® processor Z3500 series
• 6th generation Intel® Core™ processors
• Intel® Xeon® processor E3-1500m v5 product family
• 7th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
• Intel® Xeon® Scalable Processor Family
• 8th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
• Intel® Xeon® E processors
• 9th generation Intel® Core™ processors
• 2nd generation Intel® Xeon® Scalable Processor Family
• 10th generation Intel® Core™ processors
• 11th generation Intel® Core™ processors
• 3rd generation Intel® Xeon® Scalable Processor Family
• 12th generation Intel® Core™ processors
• 13th generation Intel® Core™ processors
• 4th generation Intel® Xeon® Scalable Processor Family
• 5th generation Intel® Xeon® Scalable Processor Family
• Intel® Core™ Ultra 7 processors

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium®, Pentium®, and Pentium® II, Pentium® III, and Pentium® III Xeon® processors.


The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™ 2 Duo, Intel® Core™ 2 Quad, and Intel® Core™ 2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™ 2 Quad processor Q9000 series, and Intel® Core™ 2 Extreme processors QX9000, X9000 series, Intel® Core™ 2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.


P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™ 2 Duo, Intel® Core™ 2 Extreme, Intel® Core™ 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-4x9xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor Z8000 series is based on the Airmont microarchitecture.
The Intel Atom® processor Z3400 series and the Intel Atom® processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Scalable Processor Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel Atom® processor C series, the Intel Atom® processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Scalable Processor Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Scalable Processor Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

The 12th generation Intel® Core™ processors are based on the Alder Lake performance hybrid architecture and support Intel 64 architecture.

The 13th generation Intel® Core™ processors are based on the Raptor Lake performance hybrid architecture and support Intel 64 architecture.

The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture and supports Intel 64 architecture.

The 5th generation Intel® Xeon® Scalable Processor Family is based on Emerald Rapids microarchitecture and supports Intel 64 architecture.

The Intel® Core™ Ultra 7 processor is based on Meteor Lake hybrid architecture and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

### 1.2 OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE

A description of this manual’s content follows:

**Chapter 1 — About This Manual.** Gives an overview of all volumes of the Intel® 64 and IA-32 Architectures Software Developer’s Manual. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.
Chapter 2 — Intel® 64 and IA-32 Architectures. Introduces the Intel 64 and IA-32 architectures along with the families of Intel processors that are based on these architectures. It also gives an overview of the common features found in these processors and brief history of the Intel 64 and IA-32 architectures.

Chapter 3 — Basic Execution Environment. Introduces the models of memory organization and describes the register set used by applications.

Chapter 4 — Data Types. Describes the data types and addressing modes recognized by the processor; provides an overview of real numbers and floating-point formats and of floating-point exceptions.

Chapter 5 — Instruction Set Summary. Lists all Intel 64 and IA-32 instructions, divided into technology groups.

Chapter 6 — Procedure Calls, Interrupts, and Exceptions. Describes the procedure stack and mechanisms provided for making procedure calls and for servicing interrupts and exceptions.

Chapter 7 — Programming with General-Purpose Instructions. Describes basic load and store, program control, arithmetic, and string instructions that operate on basic data types, general-purpose and segment registers; also describes system instructions that are executed in protected mode.

Chapter 8 — Programming with the x87 FPU. Describes the x87 floating-point unit (FPU), including floating-point registers and data types; gives an overview of the floating-point instruction set and describes the processor’s floating-point exception conditions.

Chapter 9 — Programming with Intel® MMX™ Technology. Describes Intel MMX technology, including MMX registers and data types; also provides an overview of the MMX instruction set.

Chapter 10 — Programming with Intel® Streaming SIMD Extensions (Intel® SSE). Describes SSE extensions, including XMM registers, the MXCSR register, and packed single precision floating-point data types; provides an overview of the SSE instruction set and gives guidelines for writing code that accesses the SSE extensions.

Chapter 11 — Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2). Describes SSE2 extensions, including XMM registers and packed double precision floating-point data types; provides an overview of the SSE2 instruction set and gives guidelines for writing code that accesses SSE2 extensions. This chapter also describes SIMD floating-point exceptions that can be generated with SSE and SSE2 instructions. It also provides general guidelines for incorporating support for SSE and SSE2 extensions into operating system and applications code.

Chapter 12 — Programming with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® Streaming SIMD Extensions 4 (Intel® SSE4) and Intel® AES New Instructions (Intel® AES-NI). Provides an overview of the SSE3 instruction set, Supplemental SSE3, SSE4, AESNI instructions, and guidelines for writing code that access these extensions.

Chapter 13 — Managing State Using the XSAVE Feature Set. Describes the XSAVE feature set instructions and explains how software can enable the XSAVE feature set and XSAVE-enabled features.

Chapter 14 — Programming with Intel® AVX, FMA, and Intel® AVX2. Provides an overview of the Intel® AVX instruction set, FMA, and Intel® AVX2 extensions and gives guidelines for writing code that access these extensions.

Chapter 15 — Programming with Intel® AVX-512. Provides an overview of the Intel® AVX-512 instruction set extensions and gives guidelines for writing code that access these extensions.

Chapter 16 — Programming with Intel® Transactional Synchronization Extensions. Describes the instruction extensions that support lock elision techniques to improve the performance of multi-threaded software with contended locks.

Chapter 17 — Control-flow Enforcement Technology. Provides an overview of the Control-flow Enforcement Technology (CET) and gives guidelines for writing code that access these extensions.

Chapter 18 — Programming with Intel® Advanced Matrix Extensions. Provides an overview of the Intel® Advanced Matrix Extensions and gives guidelines for writing code that access these extensions.

Chapter 19 — Input/Output. Describes the processor’s I/O mechanism, including I/O port addressing, I/O instructions, and I/O protection mechanisms.

Chapter 20 — Processor Identification and Feature Determination. Describes how to determine the CPU type and features available in the processor.
Appendix A — EFLAGS Cross-Reference. Summarizes how the IA-32 instructions affect the flags in the EFLAGS register.

Appendix B — EFLAGS Condition Codes. Summarizes how conditional jump, move, and ‘byte set on condition code’ instructions use condition code flags (OF, CF, ZF, SF, and PF) in the EFLAGS register.

Appendix C — Floating-Point Exceptions Summary. Summarizes exceptions raised by the x87 FPU floating-point and SSE/SSE2/SSE3 floating-point instructions.

Appendix D — Guidelines for Writing SIMD Floating-Point Exception Handlers. Gives guidelines for writing exception handlers for exceptions generated by SSE/SSE2/SSE3 floating-point instructions.

Appendix E — Intel® Memory Protection Extensions. Provides an overview of the Intel® Memory Protection Extensions, a feature that has been deprecated and will not be available on future processors.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. This notation is described below.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. See Figure 1-1.

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as reserved. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable.

Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
• When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE
Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

1.3.2.1 Instruction Operands
When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```plaintext
label: mnemonic argument1, argument2, argument3
```

where:
• A **label** is an identifier which is followed by a colon.
• A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
• The operands argument1, argument2, and argument3 are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```plaintext
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.3 Hexadecimal and Binary Numbers
Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, 0F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The “B” designation is only used in situations where confusion as to the type of number might arise.

1.3.4 Segmented Addressing
The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```plaintext
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:
DS:FF79H
The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

1.3.5 A New Syntax for CPUID, CR, and MSR Values
Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

![Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation](image)

1.3.6 Exceptions
An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other condi-
Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions that produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:


See also:

- The latest security information on Intel® products:
- Software developer resources, guidance, and insights for security advisories:
  https://software.intel.com/security-software-guidance/
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
- Intel® Fortran Compiler documentation and online help:
- Intel® Software Development Tools:
- Intel® 64 and IA-32 Architectures Software Developer’s Manual (in one, four or ten volumes):
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:
- Intel® Software Guard Extensions (Intel® SGX) Information:
- Developing Multi-threaded Applications: A Platform Consistent Approach:
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide:
  http://software.intel.com/file/30388

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference:

More relevant links are:

- Intel® Developer Zone:
  https://software.intel.com/en-us
ABOUT THIS MANUAL

- Developer centers:
- Processor support general link:
  http://www.intel.com/support/processors/
- Intel® Hyper-Threading Technology (Intel® HT Technology):
2.1 BRIEF HISTORY OF INTEL® 64 AND IA-32 ARCHITECTURES

The following sections provide a summary of the major technical evolutions from IA-32 to Intel 64 architecture: starting from the Intel 8086 processor to the latest Intel® Core® 2 Duo, Core 2 Quad and Intel Xeon processor 5300 and 7300 series. Object code created for processors released as early as 1978 still executes on the latest processors in the Intel 64 and IA-32 architecture families.

2.1.1 16-Bit Processors and Segmentation (1978)

The IA-32 architecture family was preceded by 16-bit processors, the 8086 and 8088. The 8086 has 16-bit registers and a 16-bit external data bus, with 20-bit addressing giving a 1-MByte address space. The 8088 is similar to the 8086 except it has an 8-bit external data bus.

The 8086/8088 introduced segmentation to the IA-32 architecture. With segmentation, a 16-bit segment register contains a pointer to a memory segment of up to 64 KBytes. Using four segment registers at a time, 8086/8088 processors are able to address up to 256 KBytes without switching between segments. The 20-bit addresses that can be formed using a segment register and an additional 16-bit pointer provide a total address range of 1 MByte.

2.1.2 The Intel® 286 Processor (1982)

The Intel 286 processor introduced protected mode operation into the IA-32 architecture. Protected mode uses the segment register content as selectors or pointers into descriptor tables. Descriptors provide 24-bit base addresses with a physical memory size of up to 16 MBytes, support for virtual memory management on a segment swapping basis, and a number of protection mechanisms. These mechanisms include:

• Segment limit checking.
• Read-only and execute-only segment options.
• Four privilege levels.

2.1.3 The Intel386™ Processor (1985)

The Intel386 processor was the first 32-bit processor in the IA-32 architecture family. It introduced 32-bit registers for use both to hold operands and for addressing. The lower half of each 32-bit Intel386 register retains the properties of the 16-bit registers of earlier generations, permitting backward compatibility. The processor also provides a virtual-8086 mode that allows for even greater efficiency when executing programs created for 8086/8088 processors.

In addition, the Intel386 processor has support for:

• A 32-bit address bus that supports up to 4-GBytes of physical memory.
• A segmented-memory model and a flat memory model.
• Paging, with a fixed 4-KByte page size providing a method for virtual memory management.
• Support for parallel stages.

2.1.4 The Intel486™ Processor (1989)

The Intel486™ processor added more parallel execution capability by expanding the Intel386 processor’s instruction decode and execution units into five pipelined stages. Each stage operates in parallel with the others on up to five instructions in different stages of execution.
In addition, the processor added:

- An 8-KByte on-chip first-level cache that increased the percent of instructions that could execute at the scalar rate of one per clock.
- An integrated x87 FPU.
- Power saving and system management capabilities.

### 2.1.5 The Intel® Pentium® Processor (1993)

The introduction of the Intel Pentium processor added a second execution pipeline to achieve superscalar performance (two pipelines, known as u and v, together can execute two instructions per clock). The on-chip first-level cache doubled, with 8 KBytes devoted to code and another 8 KBytes devoted to data. The data cache uses the MESI protocol to support more efficient write-back cache in addition to the write-through cache previously used by the Intel486 processor. Branch prediction with an on-chip branch table was added to increase performance in looping constructs.

In addition, the processor added:

- Extensions to make the virtual-8086 mode more efficient and allow for 4-MByte as well as 4-KByte pages.
- Internal data paths of 128 and 256 bits add speed to internal data transfers.
- Burstable external data bus was increased to 64 bits.
- An APIC to support systems with multiple processors.
- A dual processor mode to support glueless two processor systems.

A subsequent stepping of the Pentium family introduced Intel MMX technology (the Pentium Processor with MMX technology). Intel MMX technology uses the single-instruction, multiple-data (SIMD) execution model to perform parallel computations on packed integer data contained in 64-bit registers.

See Section 2.2.7, “SIMD Instructions.”

### 2.1.6 The P6 Family of Processors (1995—1999)

The P6 family of processors was based on a superscalar microarchitecture that set new performance standards; see also Section 2.2.1, “P6 Family Microarchitecture.” One of the goals in the design of the P6 family microarchitecture was to exceed the performance of the Pentium processor significantly while using the same 0.6-micrometer, four-layer, metal BICMOS manufacturing process. Members of this family include the following:

- The **Intel Pentium Pro** processor is three-way superscalar. Using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. The Pentium Pro introduced the dynamic execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a superscalar implementation. The processor was further enhanced by its caches. It has the same two on-chip 8-KByte 1st-Level caches as the Pentium processor and an additional 256-KByte Level 2 cache in the same package as the processor.

- The **Intel Pentium II** processor added Intel MMX technology to the P6 family processors along with new packaging and several hardware enhancements. The processor core is packaged in the single edge contact cartridge (SECC). The Level I data and instruction caches were enlarged to 16 KBytes each, and Level 2 cache sizes of 256 KBytes, 512 KBytes, and 1 MBytes are supported. A half-frequency backside bus connects the Level 2 cache to the processor. Multiple low-power states such as AutoHALT, Stop-Grant, Sleep, and Deep Sleep are supported to conserve power when idling.

- The **Pentium II Xeon** processor combined the premium characteristics of previous generations of Intel processors. This includes: 4-way, 8-way (and up) scalability and a 2 MBytes 2nd-Level cache running on a full-frequency backside bus.

- The **Intel Celeron** processor family focused on the value PC market segment. Its introduction offers an integrated 128 KBytes of Level 2 cache and a plastic pin grid array (P.P.G.A.) form factor to lower system design cost.

- The **Intel Pentium III** processor introduced the Streaming SIMD Extensions (SSE) to the IA-32 architecture. SSE extensions expand the SIMD execution model introduced with the Intel MMX technology by providing a
new set of 128-bit registers and the ability to perform SIMD operations on packed single precision floating-point values. See Section 2.2.7, “SIMD Instructions.”

• The Pentium III Xeon processor extended the performance levels of the IA-32 processors with the enhancement of a full-speed, on-die, and Advanced Transfer Cache.

2.1.7 The Intel® Pentium® 4 Processor Family (2000—2006)

The Intel Pentium 4 processor family is based on Intel NetBurst microarchitecture; see Section 2.2.2, “Intel NetBurst® Microarchitecture.”

The Intel Pentium 4 processor introduced Streaming SIMD Extensions 2 (SSE2); see Section 2.2.7, “SIMD Instructions.” The Intel Pentium 4 processor 3.40 GHz, supporting Hyper-Threading Technology introduced Streaming SIMD Extensions 3 (SSE3); see Section 2.2.7, “SIMD Instructions.”

Intel 64 architecture was introduced in the Intel Pentium 4 Processor Extreme Edition supporting Hyper-Threading Technology and in the Intel Pentium 4 Processor 6xx and 5xx sequences.

Intel® Virtualization Technology (Intel® VT) was introduced in the Intel Pentium 4 processor 672 and 662.

2.1.8 The Intel® Xeon® Processor (2001—2007)

Intel Xeon processors (with exception for dual-core Intel Xeon processor LV, Intel Xeon processor 5100 series) are based on the Intel NetBurst microarchitecture; see Section 2.2.2, “Intel NetBurst® Microarchitecture.” As a family, this group of IA-32 processors (more recently Intel 64 processors) is designed for use in multi-processor server systems and high-performance workstations.

The Intel Xeon processor MP introduced support for Intel® Hyper-Threading Technology; see Section 2.2.8, “Intel® Hyper-Threading Technology.”

The 64-bit Intel Xeon processor 3.60 GHz (with an 800 MHz System Bus) was used to introduce Intel 64 architecture. The Dual-Core Intel Xeon processor includes dual core technology. The Intel Xeon processor 70xx series includes Intel Virtualization Technology.

The Intel Xeon processor 5100 series introduces power-efficient, high performance Intel Core microarchitecture. This processor is based on Intel 64 architecture; it includes Intel Virtualization Technology and dual-core technology. The Intel Xeon processor 3000 series are also based on Intel Core microarchitecture. The Intel Xeon processor 5300 series introduces four processor cores in a physical package, they are also based on Intel Core microarchitecture.

2.1.9 The Intel® Pentium® M Processor (2003—2006)

The Intel Pentium M processor family is a high performance, low power mobile processor family with microarchitectural enhancements over previous generations of IA-32 Intel mobile processors. This family is designed for extending battery life and seamless integration with platform innovations that enable new usage models (such as extended mobility, ultra thin form-factors, and integrated wireless networking).

Its enhanced microarchitecture includes:

• Support for Intel Architecture with Dynamic Execution.
• A high performance, low-power core manufactured using Intel’s advanced process technology with copper interconnect.
• On-die, primary 32-KByte instruction cache and 32-KByte write-back data cache.
• On-die, second-level cache (up to 2 MByte) with Advanced Transfer Cache Architecture.
• Advanced Branch Prediction and Data Prefetch Logic.
• Support for MMX technology, Streaming SIMD instructions, and the SSE2 instruction set.
• A 400 or 533 MHz, Source-Synchronous Processor System Bus.
• Advanced power management using Enhanced Intel SpeedStep® technology.
2.1.10 **The Intel® Pentium® Processor Extreme Edition (2005)**

The Intel Pentium processor Extreme Edition introduced dual-core technology. This technology provides advanced hardware multi-threading support. The processor is based on Intel NetBurst microarchitecture and supports Intel SSE, SSE2, SSE3, Intel Hyper-Threading Technology, and Intel 64 architecture.

See also:
- Section 2.2.2, “Intel NetBurst® Microarchitecture.”
- Section 2.2.3, “Intel® Core™ Microarchitecture.”
- Section 2.2.7, “SIMD Instructions.”
- Section 2.2.8, “Intel® Hyper-Threading Technology.”
- Section 2.2.9, “Multi-Core Technology.”
- Section 2.2.10, “Intel® 64 Architecture.”

2.1.11 **The Intel® Core™ Duo and Intel® Core™ Solo Processors (2006—2007)**

The Intel Core Duo processor offers power-efficient, dual-core performance with a low-power design that extends battery life. This family and the single-core Intel Core Solo processor offer microarchitectural enhancements over Pentium M processor family.

Its enhanced microarchitecture includes:
- Intel® Smart Cache which allows for efficient data sharing between two processor cores.
- Improved decoding and SIMD execution.
- Intel® Dynamic Power Coordination and Enhanced Intel® Deeper Sleep to reduce power consumption.
- Intel® Advanced Thermal Manager which features digital thermal sensor interfaces.
- Support for power-optimized 667 MHz bus.

The dual-core Intel Xeon processor LV is based on the same microarchitecture as Intel Core Duo processor, and supports IA-32 architecture.

2.1.12 **The Intel® Xeon® Processor 5100, 5300 Series, and Intel® Core™ 2 Processor Family (2006)**

The Intel Xeon processor 3000, 3200, 5100, 5300, and 7300 series, Intel Pentium Dual-Core, Intel Core 2 Extreme, Intel Core 2 Quad processors, and the Intel Core 2 Duo processor family support Intel 64 architecture; they are based on the high-performance, power-efficient Intel® Core microarchitecture built on 65 nm process technology. The Intel Core microarchitecture includes the following innovative features:
- Intel® Wide Dynamic Execution to increase performance and execution throughput.
- Intel® Intelligent Power Capability to reduce power consumption.
- Intel® Advanced Smart Cache which allows for efficient data sharing between two processor cores.
- Intel® Smart Memory Access to increase data bandwidth and hide latency of memory accesses.
- Intel® Advanced Digital Media Boost which improves application performance using multiple generations of Streaming SIMD extensions.

The Intel Xeon processor 5300 series, Intel Core 2 Extreme processor QX6800 series, and Intel Core 2 Quad processors support Intel quad-core technology.

2.1.13 **The Intel® Xeon® Processor 5200, 5400, 7400 Series, and Intel® Core™ 2 Processor Family (2007)**

The Intel Xeon processor 5200, 5400, and 7400 series, Intel Core 2 Quad processor Q9000 Series, Intel Core 2 Duo processor E8000 series support Intel 64 architecture; they are based on the Enhanced Intel® Core microarchitec-
The Enhanced Intel Core microarchitecture provides the following improved features:

- A radix-16 divider, faster OS primitives further increases the performance of Intel® Wide Dynamic Execution.
- Improves Intel® Advanced Smart Cache with up to 50% larger level-two cache and up to 50% increase in way-set associativity.
- A 128-bit shuffler engine significantly improves the performance of Intel® Advanced Digital Media Boost and SSE4.

The Intel Xeon processor 5400 series and the Intel Core 2 Quad processor Q9000 Series support Intel quad-core technology. The Intel Xeon processor 7400 series offers up to six processor cores and an L3 cache up to 16 MBytes.

### 2.1.14 The Intel Atom® Processor Family (2008)

The first generation of Intel Atom® processors are built on 45 nm process technology. They are based on a new microarchitecture, Intel Atom® microarchitecture, which is optimized for ultra low power devices. The Intel Atom® microarchitecture features two in-order execution pipelines that minimize power consumption, increase battery life, and enable ultra-small form factors. The initial Intel Atom Processor family and subsequent generations including Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series provide the following features:

- Enhanced Intel® SpeedStep® Technology.
- Intel® Hyper-Threading Technology.
- Support for instruction set extensions up to and including Supplemental Streaming SIMD Extensions 3 (SSSE3).
- Support for Intel® Virtualization Technology.
- Support for Intel® 64 Architecture (excluding Intel Atom processor Z5xx Series).

### 2.1.15 The Intel Atom® Processor Family Based on Silvermont Microarchitecture (2013)

Intel Atom Processor C2xxx, E3xxx, S1xxx series are based on the Silvermont microarchitecture. Processors based on the Silvermont microarchitecture support instruction set extensions up to and including SSE4.2, AESNI, and PCLMULQDQ.

### 2.1.16 The Intel® Core™ i7 Processor Family (2008)

The Intel Core i7 processor 900 series supports Intel 64 architecture, and is based on Nehalem microarchitecture using 45 nm process technology. The Intel Core i7 processor and Intel Xeon processor 5500 series include the following features:

- Intel® Turbo Boost Technology converts thermal headroom into higher performance.
- Intel® HyperThreading Technology in conjunction with Quadcore to provide four cores and eight threads.
- Dedicated power control unit to reduce active and idle power consumption.
- Integrated memory controller on the processor supporting three channels of DDR3 memory.
- 8 MB inclusive Intel® Smart Cache.
- Intel® QuickPath interconnect (QPI) providing point-to-point link to chipset.
- Support for SSE4.2 and SSE4.1 instruction sets.
- Second generation Intel Virtualization Technology.

### 2.1.17 The Intel® Xeon® Processor 7500 Series (2010)

The Intel Xeon processor 7500 and 6500 series are based on Nehalem microarchitecture using 45 nm process technology. These processors support the same features described in Section 2.1.16, plus the following features:
• Up to eight cores per physical processor package.
• Up to 24 MB inclusive Intel® Smart Cache.
• Provides Intel® Scalable Memory Interconnect (Intel® SMI) channels with Intel® 7500 Scalable Memory Buffer to connect to system memory.
• Advanced RAS supporting software recoverable machine check architecture.

2.1.18 2010 Intel® Core™ Processor Family (2010)
The 2010 Intel Core processor family spans Intel Core i7, i5, and i3 processors. These processors are based on Westmere microarchitecture using 32 nm process technology. The features can include:
• Deliver smart performance using Intel Hyper-Threading Technology plus Intel Turbo Boost Technology.
• Enhanced Intel Smart Cache and integrated memory controller.
• Intelligent power gating.
• Repartitioned platform with on-die integration of 45 nm integrated graphics.
• Range of instruction set support up to AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

2.1.19 The Intel® Xeon® Processor 5600 Series (2010)
The Intel Xeon processor 5600 series are based on Westmere microarchitecture using 32 nm process technology. They support the same features described in Section 2.1.16, plus the following features:
• Up to six cores per physical processor package.
• Up to 12 MB enhanced Intel® Smart Cache.
• Support for AESNI, PCLMULQDQ, SSE4.2 and SSE4.1 instruction sets.
• Flexible Intel Virtualization Technologies across processor and I/O.

2.1.20 The Second Generation Intel® Core™ Processor Family (2011)
The Second Generation Intel Core processor family spans Intel Core i7, i5, and i3 processors based on the Sandy Bridge microarchitecture. These processors are built from 32 nm process technology and have features including:
• Intel Turbo Boost Technology for Intel Core i5 and i7 processors.
• Intel Hyper-Threading Technology.
• Enhanced Intel Smart Cache and integrated memory controller.
• Processor graphics and built-in visual features like Intel® Quick Sync Video, Intel® Insider™, etc.
• Range of instruction set support up to AVX, AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

The Intel Xeon processor E3-1200 product family is also based on the Sandy Bridge microarchitecture.
The Intel Xeon processor E5-2400/1400 product families are based on the Sandy Bridge-EP microarchitecture.
The Intel Xeon processor E5-4600/2600/1600 product families are based on the Sandy Bridge-EP microarchitecture and provide support for multiple sockets.

2.1.21 The Third Generation Intel® Core™ Processor Family (2012)
The Third Generation Intel Core processor family spans Intel Core i7, i5, and i3 processors based on the Ivy Bridge microarchitecture. The Intel Xeon processor E7-8800/4800/2800 v2 product families and Intel Xeon processor E3-1200 v2 product family are also based on the Ivy Bridge microarchitecture.
The Intel Xeon processor E5-2400/1400 v2 product families are based on the Ivy Bridge-EP microarchitecture.
The Intel Xeon processor E5-4600/2600/1600 v2 product families are based on the Ivy Bridge-EP microarchitecture and provide support for multiple sockets.
2.1.22 The Fourth Generation Intel® Core™ Processor Family (2013)

The Fourth Generation Intel Core processor family spans Intel Core i7, i5, and i3 processors based on the Haswell microarchitecture. Intel Xeon processor E3-1200 v3 product family is also based on the Haswell microarchitecture.

2.2 MORE ON SPECIFIC ADVANCES

The following sections provide more information on major innovations.

2.2.1 P6 Family Microarchitecture

The Pentium Pro processor introduced a new microarchitecture commonly referred to as P6 processor microarchitecture. The P6 processor microarchitecture was later enhanced with an on-die, Level 2 cache, called Advanced Transfer Cache.

The microarchitecture is a three-way superscalar, pipelined architecture. Three-way superscalar means that by using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. To handle this level of instruction throughput, the P6 processor family uses a decoupled, 12-stage superpipeline that supports out-of-order instruction execution.

Figure 2-1 shows a conceptual view of the P6 processor microarchitecture pipeline with the Advanced Transfer Cache enhancement.

![Figure 2-1. The P6 Processor Microarchitecture with Advanced Transfer Cache Enhancement](image)

To ensure a steady supply of instructions and data for the instruction execution pipeline, the P6 processor microarchitecture incorporates two cache levels. The Level 1 cache provides an 8-KByte instruction cache and an 8-KByte data cache, both closely coupled to the pipeline. The Level 2 cache provides 256-KByte, 512-KByte, or 1-MByte static RAM that is coupled to the core processor through a full clock-speed 64-bit cache bus.

The centerpiece of the P6 processor microarchitecture is an out-of-order execution mechanism called dynamic execution. Dynamic execution incorporates three data-processing concepts:
• **Deep branch prediction** allows the processor to decode instructions beyond branches to keep the instruction pipeline full. The P6 processor family implements highly optimized branch prediction algorithms to predict the direction of the instruction.

• **Dynamic data flow analysis** requires real-time analysis of the flow of data through the processor to determine dependencies and to detect opportunities for out-of-order instruction execution. The out-of-order execution core can monitor many instructions and execute these instructions in the order that best optimizes the use of the processor’s multiple execution units, while maintaining the data integrity.

• **Speculative execution** refers to the processor’s ability to execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream. To make speculative execution possible, the P6 processor microarchitecture decouples the dispatch and execution of instructions from the commitment of results. The processor’s out-of-order execution core uses data-flow analysis to execute all available instructions in the instruction pool and store the results in temporary registers. The retirement unit then linearly searches the instruction pool for completed instructions that no longer have data dependencies with other instructions or unresolved branch predictions. When completed instructions are found, the retirement unit commits the results of these instructions to memory and/or the IA-32 registers (the processor’s eight general-purpose registers and eight x87 FPU data registers) in the order they were originally issued and retires the instructions from the instruction pool.

### 2.2.2 Intel NetBurst® Microarchitecture

The Intel NetBurst microarchitecture provides:

• The Rapid Execution Engine.
  — Arithmetic Logic Units (ALUs) run at twice the processor frequency.
  — Basic integer operations can dispatch in 1/2 processor clock tick.

• Hyper-Pipelined Technology.
  — Deep pipeline to enable industry-leading clock rates for desktop PCs and servers.
  — Frequency headroom and scalability to continue leadership into the future.

• Advanced Dynamic Execution.
    • Up to 126 instructions in flight.
    • Up to 48 loads and 24 stores in pipeline\(^1\).
  — Enhanced branch prediction capability.
    • Reduces the misprediction penalty associated with deeper pipelines.
    • Advanced branch prediction algorithm.
    • 4K-entry branch target array.

• New cache subsystem.
  — First level caches.
    • Advanced Execution Trace Cache stores decoded instructions.
    • Execution Trace Cache removes decoder latency from main execution loops.
    • Execution Trace Cache integrates path of program execution flow into a single line.
    • Low latency data cache.
  — Second level cache.
    • Full-speed, unified 8-way Level 2 on-die Advance Transfer Cache.
    • Bandwidth and performance increases with processor frequency.

---

1. Intel 64 and IA-32 processors based on the Intel NetBurst microarchitecture at 90 nm process can handle more than 24 stores in flight.
• High-performance, quad-pumped bus interface to the Intel NetBurst microarchitecture system bus.
  — Supports quad-pumped, scalable bus clock to achieve up to 4X effective speed.
  — Capable of delivering up to 8.5 GBytes of bandwidth per second.
• Superscalar issue to enable parallelism.
• Expanded hardware registers with renaming to avoid register name space limitations.
• 64-byte cache line size (transfers data up to two lines per sector).

Figure 2-2 is an overview of the Intel NetBurst microarchitecture. This microarchitecture pipeline is made up of three sections: (1) the front end pipeline, (2) the out-of-order execution core, and (3) the retirement unit.

![Figure 2-2. The Intel NetBurst® Microarchitecture](image_url)

### 2.2.2.1 The Front End Pipeline

The front end supplies instructions in program order to the out-of-order execution core. It performs a number of functions:

• Prefetches instructions that are likely to be executed.
• Fetches instructions that have not already been prefetched.
• Decodes instructions into micro-operations.
• Generates microcode for complex instructions and special-purpose code.
• Delivers decoded instructions from the execution trace cache.
• Predicts branches using highly advanced algorithm.

The pipeline is designed to address common problems in high-speed, pipelined microprocessors. Two of these problems contribute to major sources of delays:

• Time to decode instructions fetched from the target.
• Wasted decode bandwidth due to branches or branch target in the middle of cache lines.

The operation of the pipeline’s trace cache addresses these issues. Instructions are constantly being fetched and decoded by the translation engine (part of the fetch/decode logic) and built into sequences of micro-ops called traces. At any time, multiple traces (representing prefetched branches) are being stored in the trace cache. The trace cache is searched for the instruction that follows the active branch. If the instruction also appears as the first instruction in a pre-fetched branch, the fetch and decode of instructions from the memory hierarchy ceases and the pre-fetched branch becomes the new source of instructions (see Figure 2-2).

The trace cache and the translation engine have cooperating branch prediction hardware. Branch targets are predicted based on their linear addresses using branch target buffers (BTBs) and fetched as soon as possible.

2.2.2.2 Out-Of-Order Execution Core

The out-of-order execution core’s ability to execute instructions out of order is a key factor in enabling parallelism. This feature enables the processor to reorder instructions so that if one micro-op is delayed, other micro-ops may proceed around it. The processor employs several buffers to smooth the flow of micro-ops.

The core is designed to facilitate parallel execution. It can dispatch up to six micro-ops per cycle (this exceeds trace cache and retirement micro-op bandwidth). Most pipelines can start executing a new micro-op every cycle, so several instructions can be in flight at a time for each pipeline. A number of arithmetic logical unit (ALU) instructions can start at two per cycle; many floating-point instructions can start once every two cycles.

2.2.2.3 Retirement Unit

The retirement unit receives the results of the executed micro-ops from the out-of-order execution core and processes the results so that the architectural state updates according to the original program order.

When a micro-op completes and writes its result, it is retired. Up to three micro-ops may be retired per cycle. The Reorder Buffer (ROB) is the unit in the processor which buffers completed micro-ops, updates the architectural state in order, and manages the ordering of exceptions. The retirement section also keeps track of branches and sends updated branch target information to the BTB. The BTB then purges pre-fetched traces that are no longer needed.

2.2.3 Intel® Core™ Microarchitecture

Intel Core microarchitecture introduces the following features that enable high performance and power-efficient performance for single-threaded as well as multi-threaded workloads:

• **Intel® Wide Dynamic Execution** enable each processor core to fetch, dispatch, execute in high bandwidths to support retirement of up to four instructions per cycle.
  — Fourteen-stage efficient pipeline.
  — Three arithmetic logical units.
  — Four decoders to decode up to five instruction per cycle.
  — Macro-fusion and micro-fusion to improve front-end throughput.
  — Peak issue rate of dispatching up to six micro-ops per cycle.
  — Peak retirement bandwidth of up to 4 micro-ops per cycle.
  — Advanced branch prediction.
  — Stack pointer tracker to improve efficiency of executing function/procedure entries and exits.

• **Intel® Advanced Smart Cache** delivers higher bandwidth from the second level cache to the core, and optimal performance and flexibility for single-threaded and multi-threaded applications.
  — Large second level cache up to 4 MB and 16-way associativity.
  — Optimized for multicore and single-threaded execution environments.
  — 256-bit internal data path to improve bandwidth from L2 to first-level data cache.
• **Intel® Smart Memory Access** prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution.
  — Hardware prefetchers to reduce effective latency of second-level cache misses.
  — Hardware prefetchers to reduce effective latency of first-level data cache misses.
  — Memory disambiguation to improve efficiency of speculative execution engine.

• **Intel® Advanced Digital Media Boost** improves most 128-bit SIMD instructions with single-cycle throughput and floating-point operations.
  — Single-cycle throughput of most 128-bit SIMD instructions.
  — Up to eight floating-point operations per cycle.
  — Three issue ports available to dispatching SIMD instructions for execution.

Intel Core 2 Extreme, Intel Core 2 Duo processors and Intel Xeon processor 5100 series implement two processor cores based on the Intel Core microarchitecture, the functionality of the subsystems in each core are depicted in Figure 2-3.

![Figure 2-3. The Intel® Core™ Microarchitecture Pipeline Functionality](image)

### 2.2.3.1 The Front End

The front end of Intel Core microarchitecture provides several enhancements to feed the Intel Wide Dynamic Execution engine:

• Instruction fetch unit prefetches instructions into an instruction queue to maintain steady supply of instruction to the decode units.
• Four-wide decode unit can decode 4 instructions per cycle or 5 instructions per cycle with Macrofusion.
• Macrofusion fuses common sequence of two instructions as one decoded instruction (micro-ops) to increase decoding throughput.
• Microfusion fuses common sequence of two micro-ops as one micro-ops to improve retirement throughput.
• Instruction queue provides caching of short loops to improve efficiency.
• Stack pointer tracker improves efficiency of executing procedure/function entries and exits.
• Branch prediction unit employs dedicated hardware to handle different types of branches for improved branch prediction.
• Advanced branch prediction algorithm directs instruction fetch unit to fetch instructions likely in the architectural code path for decoding.

2.2.3.2 Execution Core
The execution core of the Intel Core microarchitecture is superscalar and can process instructions out of order to increase the overall rate of instructions executed per cycle (IPC). The execution core employs the following feature to improve execution throughput and efficiency:
• Up to six micro-ops can be dispatched to execute per cycle.
• Up to four instructions can be retired per cycle.
• Three full arithmetic logical units.
• SIMD instructions can be dispatched through three issue ports.
• Most SIMD instructions have 1-cycle throughput (including 128-bit SIMD instructions).
• Up to eight floating-point operation per cycle.
• Many long-latency computation operation are pipelined in hardware to increase overall throughput.
• Reduced exposure to data access delays using Intel Smart Memory Access.

2.2.4 Intel Atom® Microarchitecture
Intel Atom microarchitecture maximizes power-efficient performance for single-threaded and multi-threaded workloads by providing:
• **Advanced Micro-Ops Execution**
  — Single-micro-op instruction execution from decode to retirement, including instructions with register-only, load, and store semantics.
  — Sixteen-stage, in-order pipeline optimized for throughput and reduced power consumption.
  — Dual pipelines to enable decode, issue, execution, and retirement of two instructions per cycle.
  — Advanced stack pointer to improve efficiency of executing function entry/returns.
• **Intel® Smart Cache**
  — Second level cache is 512 KB and 8-way associativity.
  — Optimized for multi-threaded and single-threaded execution environments
  — 256-bit internal data path between L2 and L1 data caches improves high bandwidth.
• **Efficient Memory Access**
  — Efficient hardware prefetchers to L1 and L2, speculatively loading data likely to be requested by processor to reduce cache miss impact.
• **Intel® Digital Media Boost**
  — Two issue ports for dispatching SIMD instructions to execution units.
  — Single-cycle throughput for most 128-bit integer SIMD instructions.
  — Up to six floating-point operations per cycle.
  — Up to two 128-bit SIMD integer operations per cycle.
  — Safe Instruction Recognition (SIR) to allow long-latency floating-point operations to retire out of order with respect to integer instructions.
2.2.5 Nehalem Microarchitecture

Nehalem microarchitecture provides the foundation for many features of Intel Core i7 processors. It builds on the success of 45 nm Intel Core microarchitecture and provides the following feature enhancements:

- **Enhanced processor core**
  - Improved branch prediction and recovery from misprediction.
  - Enhanced loop streaming to improve front end performance and reduce power consumption.
  - Deeper buffering in out-of-order engine to extract parallelism.
  - Enhanced execution units to provide acceleration in CRC, string/text processing and data shuffling.

- **Smart Memory Access**
  - Integrated memory controller provides low-latency access to system memory and scalable memory bandwidth.
  - New cache hierarchy organization with shared, inclusive L3 to reduce snoop traffic.
  - Two level TLBs and increased TLB size.
  - Fast unaligned memory access.

- **HyperThreading Technology**
  - Provides two hardware threads (logical processors) per core.
  - Takes advantage of 4-wide execution engine, large L3, and massive memory bandwidth.

- **Dedicated Power management Innovations**
  - Integrated microcontroller with optimized embedded firmware to manage power consumption.
  - Embedded real-time sensors for temperature, current, and power.
  - Integrated power gate to turn off/on per-core power consumption.
  - Versatility to reduce power consumption of memory, link subsystems.

2.2.6 Sandy Bridge Microarchitecture

Sandy Bridge microarchitecture builds on the successes of Intel® Core™ microarchitecture and Nehalem microarchitecture. It offers the following features:

- Intel Advanced Vector Extensions (Intel AVX).
  - 256-bit floating-point instruction set extensions to the 128-bit Intel Streaming SIMD Extensions, providing up to 2X performance benefits relative to 128-bit code.
  - Non-destructive destination encoding offers more flexible coding techniques.
  - Supports flexible migration and co-existence between 256-bit AVX code, 128-bit AVX code and legacy 128-bit SSE code.

- Enhanced front-end and execution engine.
  - New decoded Icache component that improves front-end bandwidth and reduces branch misprediction penalty.
  - Advanced branch prediction.
  - Additional macro-fusion support.
  - Larger dynamic execution window.
  - Multi-precision integer arithmetic enhancements (ADC/SBB, MUL/IMUL).
  - LEA bandwidth improvement.
  - Reduction of general execution stalls (read ports, writeback conflicts, bypass latency, partial stalls).
  - Fast floating-point exception handling.
- XSAVE/XRSTORE performance improvements and XSAVEOPT new instruction.
- Cache hierarchy improvements for wider data path.
  - Doubling of bandwidth enabled by two symmetric ports for memory operation.
  - Simultaneous handling of more in-flight loads and stores enabled by increased buffers.
  - Internal bandwidth of two loads and one store each cycle.
  - Improved prefetching.
  - High bandwidth low latency LLC architecture.
  - High bandwidth ring architecture of on-die interconnect.

For additional information on Intel® Advanced Vector Extensions (AVX), see Section 5.13, "Intel® Advanced Vector Extensions (Intel® AVX)" and Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2" in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

### 2.2.7 SIMD Instructions

Beginning with the Pentium II and Pentium with Intel MMX technology processor families, six extensions have been introduced into the Intel 64 and IA-32 architectures to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, SSE3 extensions, Supplemental Streaming SIMD Extensions 3, and SSE4. Each of these extensions provides a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements.

SIMD integer operations can use the 64-bit MMX or the 128-bit XMM registers. SIMD floating-point operations use 128-bit XMM registers. Figure 2-4 shows a summary of the various SIMD extensions (MMX technology, Intel SSE, Intel SSE2, Intel SSE3, SSSE3, and Intel SSE4), the data types they operate on, and how the data types are packed into MMX and XMM registers.

The Intel MMX technology was introduced in the Pentium II and Pentium with MMX technology processor families. MMX instructions perform SIMD operations on packed byte, word, or doubleword integers located in MMX registers. These instructions are useful in applications that operate on integer arrays and streams of integer data that lend themselves to SIMD processing.

Intel SSE was introduced in the Pentium III processor family. Intel SSE instructions operate on packed single precision floating-point values contained in XMM registers and on packed integers contained in MMX registers. Several Intel SSE instructions provide state management, cache control, and memory ordering operations. Other Intel SSE instructions are targeted at applications that operate on arrays of single precision floating-point data elements (3-D geometry, 3-D rendering, and video encoding and decoding applications).

Intel SSE2 was introduced in the Pentium 4 and Intel Xeon processors. Intel SSE2 instructions operate on packed double precision floating-point values contained in XMM registers and on packed integers contained in MMX and XMM registers. Intel SSE2 integer instructions extend IA-32 SIMD operations by adding new 128-bit SIMD integer operations and by expanding existing 64-bit SIMD integer operations to 128-bit XMM capability. Intel SSE2 instructions also provide new cache control and memory ordering operations.

Intel SSE3 was introduced with the Pentium 4 processor supporting Hyper-Threading Technology (built on 90 nm process technology). Intel SSE3 offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities.

SSSE3 was introduced with the Intel Xeon processor 5100 series and Intel Core 2 processor family. SSSE3 offers 32 instructions to accelerate processing of SIMD integer data.

Intel SSE4 offers 54 instructions. 47 of them are referred to as Intel SSE4.1 instructions. Intel SSE4.1 was introduced with the Intel Xeon processor 5400 series and Intel Core 2 Extreme processor QX9650. The other seven Intel SSE4 instructions are referred to as Intel SSE4.2 instructions.

Intel AES-NI and PCLMULQDQ introduced seven new instructions. Six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, and are referred to as Intel AES-NI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.
Intel 64 architecture allows four generations of 128-bit SIMD extensions to access up to 16 XMM registers. IA-32 architecture provides eight XMM registers.

Intel® Advanced Vector Extensions offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three operand syntax and to simplify compiler vectorization of high-level language expressions.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

In addition to performance considerations, programmers should also be cognizant of the implications of VEX-encoded AVX instructions with the expectations of system software components that manage the processor state components enabled by XCR0. For additional information see Section 2.3.10.1, “Vector Length Transition and Programming Considerations” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

See also:

- Section 5.4, “MMX Instructions,” and Chapter 9, “Programming with Intel® MMX™ Technology.”
- Section 5.5, “Intel® SSE Instructions,” and Chapter 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE).”
- Section 5.6, “Intel® SSE2 Instructions,” and Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2).”
Intel Hyper-Threading Technology (Intel HT Technology) was developed to improve the performance of IA-32 processors when executing multi-threaded operating system and application code or single-threaded applications under multi-tasking environments. The technology enables a single physical processor to execute two or more separate code streams (threads) concurrently using shared execution resources.

Intel HT Technology is one form of hardware multi-threading capability in IA-32 processor families. It differs from multi-processor capability using separate physically distinct packages with each physical processor package mated with a physical socket. Intel HT Technology provides hardware multi-threading capability with a single physical package by using shared execution resources in a processor core.

Architecturally, an IA-32 processor that supports Intel HT Technology consists of two or more logical processors, each of which has its own IA-32 architectural state. Each logical processor consists of a full set of IA-32 data registers, segment registers, control registers, debug registers, and most of the MSRs. Each also has its own advanced programmable interrupt controller (APIC).

Figure 2-5 shows a comparison of a processor that supports Intel HT Technology (implemented with two logical processors) and a traditional dual processor system.
Unlike a traditional MP system configuration that uses two or more separate physical IA-32 processors, the logical processors in an IA-32 processor supporting Intel HT Technology share the core resources of the physical processor. This includes the execution engine and the system bus interface. After power up and initialization, each logical processor can be independently directed to execute a specified thread, interrupted, or halted.

Intel HT Technology leverages the process and thread-level parallelism found in contemporary operating systems and high-performance applications by providing two or more logical processors on a single chip. This configuration allows two or more threads\(^1\) to be executed simultaneously on each a physical processor. Each logical processor executes instructions from an application thread using the resources in the processor core. The core executes these threads concurrently, using out-of-order instruction scheduling to maximize the use of execution units during each clock cycle.

### 2.2.8.1 Some Implementation Notes

All Intel HT Technology configurations require:

- A processor that supports Intel HT Technology.
- A chipset and BIOS that utilize the technology.
- Operating system optimizations.


At the firmware (BIOS) level, the basic procedures to initialize the logical processors in a processor supporting Intel HT Technology are the same as those for a traditional DP or MP platform. The mechanisms that are described in the Multiprocessor Specification, Version 1.4, to power-up and initialize physical processors in an MP system also apply to logical processors in a processor that supports Intel HT Technology.

An operating system designed to run on a traditional DP or MP platform may use CPUID to determine the presence of hardware multi-threading support feature and the number of logical processors they provide.

Although existing operating system and application code should run correctly on a processor that supports Intel HT Technology, some code modifications are recommended to get the optimum benefit. These modifications are discussed in Chapter 7, “Multiple-Processor Management,” Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

---

1. In the remainder of this document, the term "thread" will be used as a general term for the terms "process" and "thread."
2.2.9 Multi-Core Technology

Multi-core technology is another form of hardware multi-threading capability in IA-32 processor families. Multi-core technology enhances hardware multi-threading capability by providing two or more execution cores in a physical package.

The Intel Pentium processor Extreme Edition is the first member in the IA-32 processor family to introduce multi-core technology. The processor provides hardware multi-threading support with both two processor cores and Intel Hyper-Threading Technology. This means that the Intel Pentium processor Extreme Edition provides four logical processors in a physical package (two logical processors for each processor core). The Dual-Core Intel Xeon processor features multi-core, Intel Hyper-Threading Technology and supports multi-processor platforms.

The Intel Pentium D processor also features multi-core technology. This processor provides hardware multi-threading support with two processor cores but does not offer Intel Hyper-Threading Technology. This means that the Intel Pentium D processor provides two logical processors in a physical package, with each logical processor owning the complete execution resources of a processor core.

The Intel Core 2 processor family, Intel Xeon processor 3000 series, Intel Xeon processor 5100 series, and Intel Core Duo processor offer power-efficient multi-core technology. The processor contains two cores that share a smart second level cache. The Level 2 cache enables efficient data sharing between two cores to reduce memory traffic to the system bus.

The Pentium® dual-core processor is based on the same technology as the Intel Core 2 Duo processor family.

The Intel Xeon processor 7300, 5300, and 3200 series, Intel Core 2 Extreme Quad-Core processor, and Intel Core 2 Quad processors support Intel quad-core technology. The Quad-core Intel Xeon processors and the Quad-Core Intel Core 2 processor family are also in Figure 2-7.
Intel Core i7 processors support Intel quad-core technology, Intel HyperThreading Technology, provides Intel QuickPath interconnect link to the chipset and have integrated memory controller supporting three channels to DDR3 memory.

**Figure 2-7. Intel® 64 Processors that Support Quad-Core**

**Figure 2-8. Intel® Core™ i7 Processor**
2.2.10 Intel® 64 Architecture

Intel 64 architecture increases the linear address space for software to 64 bits and supports physical address space up to 52 bits. The technology also introduces a new operating mode referred to as IA-32e mode.

IA-32e mode operates in one of two sub-modes: (1) compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified, (2) 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit address space.

In the 64-bit mode, applications may access:

- 64-bit flat linear addressing.
- 8 additional general-purpose registers (GPRs).
- 8 additional registers for streaming SIMD extensions (Intel SSE, SSE2, and SSE3, and SSSE3).
- 64-bit-wide GPRs and instruction pointers.
- Uniform byte-register addressing.
- Fast interrupt-prioritization mechanism.
- A new instruction-pointer relative-addressing mode.

An Intel 64 architecture processor supports existing IA-32 software because it is able to run all non-64-bit legacy modes supported by IA-32 architecture. Most existing IA-32 applications also run in compatibility mode.

2.2.11 Intel® Virtualization Technology (Intel® VT)

Intel® Virtualization Technology for Intel 64 and IA-32 architectures provide extensions that support virtualization. The extensions are referred to as Virtual Machine Extensions (VMX). An Intel 64 or IA-32 platform with VMX can function as multiple virtual systems (or virtual machines). Each virtual machine can run operating systems and applications in separate partitions.

VMX also provides programming interface for a new layer of system software (called the Virtual Machine Monitor (VMM)) used to manage the operation of virtual machines. Information on VMX and on the programming of VMMs is in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

Intel Core i7 processor provides the following enhancements to Intel Virtualization Technology:

- Virtual processor ID (VPID) to reduce the cost of VMM managing transitions.
- Extended page table (EPT) to reduce the number of transitions for VMM to manage memory virtualization.
- Reduced latency of VM transitions.

2.3 INTEL® 64 AND IA-32 PROCESSOR GENERATIONS

In the mid-1960s, Intel co-founder and Chairman Emeritus Gordon Moore had this observation: "... the number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years.” Over the past three and half decades, this prediction known as “Moore's Law” has continued to hold true.

The computing power and the complexity (or roughly, the number of transistors per processor) of Intel architecture processors has grown in close relation to Moore's law. By taking advantage of new process technology and new microarchitecture designs, each new generation of IA-32 processors has demonstrated frequency-scaling headroom and new performance levels over the previous generation processors.
The key features of the Intel Pentium 4 processor, Intel Xeon processor, Intel Xeon processor MP, Pentium III processor, and Pentium III Xeon processor with advanced transfer cache are shown in Table 2-1. Older generation IA-32 processors, which do not employ on-die Level 2 cache, are shown in Table 2-2.

### Table 2-1. Key Features of Most Recent IA-32 Processors

<table>
<thead>
<tr>
<th>Intel Processor</th>
<th>Date Introduced</th>
<th>Microarchitecture</th>
<th>Top-Bin Clock Frequency at Introduction</th>
<th>Transistors</th>
<th>Register Sizes</th>
<th>System Bus Bandwidth</th>
<th>Max. Extern. Addr. Space</th>
<th>On-Die Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium M Processor 755³</td>
<td>2004</td>
<td>Intel Pentium M Processor</td>
<td>2.00 GHz</td>
<td>140 M</td>
<td>GP: 32 FPU: 80 MMX: 64 XMM: 128</td>
<td>3.2 GB/s</td>
<td>4 GB</td>
<td>L1: 64 KB L2: 2 MB</td>
</tr>
<tr>
<td>Intel Core Duo Processor T2600³</td>
<td>2006</td>
<td>Improved Intel Pentium M Processor Microarchitecture; Dual Core; Intel Smart Cache, Advanced Thermal Manager</td>
<td>2.16 GHz</td>
<td>152 M</td>
<td>GP: 32 FPU: 80 MMX: 64 XMM: 128</td>
<td>5.3 GB/s</td>
<td>4 GB</td>
<td>L1: 64 KB L2: 2 MB (2 MB Total)</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The register size and external data bus size are given in bits.
2. First level cache is denoted using the abbreviation L1, 2nd level cache is denoted as L2. The size of L1 includes the first-level data cache and the instruction cache where applicable, but does not include the trace cache.
3. Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families.
4. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.

### Table 2-2. Key Features of Most Recent Intel 64 Processors

<table>
<thead>
<tr>
<th>Intel Processor</th>
<th>Date Introduced</th>
<th>Microarchitecture</th>
<th>Highest Processor Base Frequency at Introduction</th>
<th>Transistors</th>
<th>Register Sizes</th>
<th>System Bus/QPI Link Speed</th>
<th>Max. Extern. Addr. Space</th>
<th>On-Die Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Intel Xeon Processor with 800 MHz System Bus</td>
<td>2004</td>
<td>Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture</td>
<td>3.60 GHz</td>
<td>125 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>6.4 GB/s</td>
<td>64 GB</td>
<td>12K µop Execution Trace Cache; 16 KB L1; 1 MB L2</td>
</tr>
<tr>
<td>64-bit Intel Xeon Processor MP with 8MB L3</td>
<td>2005</td>
<td>Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture</td>
<td>3.33 GHz</td>
<td>675 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>5.3 GB/s</td>
<td>1024 GB (1 TB)</td>
<td>12K µop Execution Trace Cache; 16 KB L1; 1 MB L2; 8 MB L3</td>
</tr>
<tr>
<td>Intel Processor</td>
<td>Date Introduced</td>
<td>Micro-architecture</td>
<td>Highest Processor Base Frequency at Introduction</td>
<td>Transistors</td>
<td>Register Sizes</td>
<td>System Bus/QPI Link Speed</td>
<td>Max. Extern. Addr. Space</td>
<td>On-Die Caches</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>--------------------</td>
<td>-----------------------------------------------</td>
<td>-------------</td>
<td>----------------</td>
<td>---------------------------</td>
<td>----------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology</td>
<td>2005</td>
<td>Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture</td>
<td>3.73 GHz</td>
<td>164 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>8.5 GB/s</td>
<td>64 GB</td>
<td>12K mop Execution Trace Cache; 16 KB L1; 2 MB L2</td>
</tr>
<tr>
<td>Intel Pentium Processor Extreme Edition 840</td>
<td>2005</td>
<td>Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core 2</td>
<td>3.20 GHz</td>
<td>230 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>6.4 GB/s</td>
<td>64 GB</td>
<td>12K mop Execution Trace Cache; 16 KB L1; 1 MB L2 (2 MB Total)</td>
</tr>
<tr>
<td>Dual-Core Intel Xeon Processor 7041</td>
<td>2005</td>
<td>Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core 3</td>
<td>3.00 GHz</td>
<td>321 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>6.4 GB/s</td>
<td>64 GB</td>
<td>12K mop Execution Trace Cache; 16 KB L1; 2 MB L2 (4 MB Total)</td>
</tr>
<tr>
<td>Intel Pentium 4 Processor 672</td>
<td>2005</td>
<td>Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.80 GHz</td>
<td>164 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>6.4 GB/s</td>
<td>64 GB</td>
<td>12K mop Execution Trace Cache; 16 KB L1; 2 MB L2</td>
</tr>
<tr>
<td>Intel Pentium Processor Extreme Edition 955</td>
<td>2006</td>
<td>Intel NetBurst Microarchitecture; Intel 64 Architecture; Dual Core; Intel Virtualization Technology.</td>
<td>3.46 GHz</td>
<td>376 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>8.5 GB/s</td>
<td>64 GB</td>
<td>12K mop Execution Trace Cache; 16 KB L1; 2 MB L2 (4 MB Total)</td>
</tr>
<tr>
<td>Intel Core 2 Extreme Processor X6800</td>
<td>2006</td>
<td>Intel Core Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.93 GHz</td>
<td>291 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>8.5 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB L2: 4 MB (4 MB Total)</td>
</tr>
<tr>
<td>Intel Processor</td>
<td>Date Introduced</td>
<td>Micro-architecture</td>
<td>Highest Processor Base Frequency at Introduction</td>
<td>Transistors</td>
<td>Register Sizes</td>
<td>System Bus/QPI Link Speed</td>
<td>Max. Extern. Addr. Space</td>
<td>On-Die Caches</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>-----------------------------------------------</td>
<td>-------------</td>
<td>---------------</td>
<td>--------------------------</td>
<td>------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Intel Xeon Processor 5160</td>
<td>2006</td>
<td>Intel Core Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.00 GHz</td>
<td>291 M</td>
<td>GP: 32, FPU: 64 MMX: 64 XMM: 128</td>
<td>10.6 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB, L2: 4 MB (4 MB Total)</td>
</tr>
<tr>
<td>Intel Xeon Processor 7140</td>
<td>2006</td>
<td>Intel NetBurst Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.40 GHz</td>
<td>1.3 B</td>
<td>GP: 32, FPU: 64 MMX: 64 XMM: 128</td>
<td>12.8 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB, L2: 1 MB (2 MB Total), L3: 16 MB (16 MB Total)</td>
</tr>
<tr>
<td>Intel Core 2 Extreme Processor QX6700</td>
<td>2006</td>
<td>Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.66 GHz</td>
<td>582 M</td>
<td>GP: 32, FPU: 64 MMX: 64 XMM: 128</td>
<td>8.5 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB, L2: 4 MB (4 MB Total)</td>
</tr>
<tr>
<td>Quad-core Intel Xeon Processor 5355</td>
<td>2006</td>
<td>Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.66 GHz</td>
<td>582 M</td>
<td>GP: 32, FPU: 64 MMX: 64 XMM: 128</td>
<td>10.6 GB/s</td>
<td>256 GB</td>
<td>L1: 64 KB, L2: 4 MB (8 MB Total)</td>
</tr>
<tr>
<td>Intel Core 2 Duo Processor E6850</td>
<td>2007</td>
<td>Intel Core Microarchitecture; Dual Core; Intel 64 Architecture; Intel Virtualization Technology; Intel Trusted Execution Technology</td>
<td>3.00 GHz</td>
<td>291 M</td>
<td>GP: 32, FPU: 64 MMX: 64 XMM: 128</td>
<td>10.6 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB, L2: 4 MB (4 MB Total)</td>
</tr>
<tr>
<td>Intel Xeon Processor 7350</td>
<td>2007</td>
<td>Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.93 GHz</td>
<td>582 M</td>
<td>GP: 32, FPU: 64 MMX: 64 XMM: 128</td>
<td>8.5 GB/s</td>
<td>1024 GB</td>
<td>L1: 64 KB, L2: 4 MB (8 MB Total)</td>
</tr>
</tbody>
</table>
### Table 2-2. Key Features of Most Recent Intel® 64 Processors (Contd.)

<table>
<thead>
<tr>
<th>Intel Processor</th>
<th>Date Introduced</th>
<th>Micro-architecture</th>
<th>Highest Processor Base Frequency at Introduction</th>
<th>Transistors</th>
<th>Register Sizes</th>
<th>System Bus/QPI Link Speed</th>
<th>Max. Extern. Addr. Space</th>
<th>On-Die Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Processor 5472</td>
<td>2007</td>
<td>Enhanced Intel Core Microarchitecture; Quad Core; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.00 GHz</td>
<td>820 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>12.8 GB/s</td>
<td>256 GB</td>
<td>L1: 64 KB L2: 6 MB (12 MB Total)</td>
</tr>
<tr>
<td>Intel Atom Processor</td>
<td>2008</td>
<td>Intel Atom Microarchitecture; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.0 - 1.60 GHz</td>
<td>47 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>Up to 4.2 GB/s</td>
<td>Up to 64GB</td>
<td>L1: 56 KB</td>
</tr>
<tr>
<td>Intel Xeon Processor 7460</td>
<td>2008</td>
<td>Enhanced Intel Core Microarchitecture; Six Cores; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.67 GHz</td>
<td>1.9 B</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>8.5 GB/s</td>
<td>1024 GB</td>
<td>L1: 64 KB L2: 3 MB (9 MB Total) L3: 16 MB</td>
</tr>
<tr>
<td>Intel Atom Processor 330</td>
<td>2008</td>
<td>Intel Atom Microarchitecture; Intel 64 Architecture; Dual core; Intel Virtualization Technology.</td>
<td>1.60 GHz</td>
<td>94 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>Up to 4.2 GB/s</td>
<td>Up to 64GB</td>
<td>L1: 56 KB</td>
</tr>
<tr>
<td>Intel Core i7-965 Processor Extreme Edition</td>
<td>2008</td>
<td>Nehalem microarchitecture; Quadcore; HyperThreading Technology; Intel QPI; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.20 GHz</td>
<td>731 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>QPI: 6.4 GT/s; Memory: 25 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB L2: 256 KB L3: 8 MB</td>
</tr>
<tr>
<td>Intel Processor</td>
<td>Date Introduced</td>
<td>Micro-architecture</td>
<td>Highest Processor Base Frequency at Introduction</td>
<td>Transistors</td>
<td>Register Sizes</td>
<td>System Bus/QPI Link Speed</td>
<td>Max. Extern. Addr. Space</td>
<td>On-Die Caches</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------</td>
<td>--------------------</td>
<td>--------------------------------------------------</td>
<td>-------------</td>
<td>----------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Intel Core i7-620M Processor</td>
<td>2010</td>
<td>Intel Turbo Boost Technology, Westmere microarchitecture; Dual-core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology, Integrated graphics</td>
<td>2.66 GHz</td>
<td>383 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>64 GB</td>
<td>L1: 64 KB L2: 256 KB L3: 4 MB</td>
<td></td>
</tr>
<tr>
<td>Intel Xeon-Processor 5680</td>
<td>2010</td>
<td>Intel Turbo Boost Technology, Westmere microarchitecture; Six core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.33 GHz</td>
<td>1.1 B</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>QPI: 6.4 GT/s; 32 GB/s</td>
<td>1 TB</td>
<td>L1: 64 KB L2: 256 KB L3: 12 MB</td>
</tr>
<tr>
<td>Intel Xeon-Processor 7560</td>
<td>2010</td>
<td>Intel Turbo Boost Technology, Nehalem microarchitecture; Eight core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.26 GHz</td>
<td>2.3 B</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>QPI: 6.4 GT/s; Memory: 76 GB/s</td>
<td>16 TB</td>
<td>L1: 64 KB L2: 256 KB L3: 24 MB</td>
</tr>
<tr>
<td>Intel Core i7-2600K Processor</td>
<td>2011</td>
<td>Intel Turbo Boost Technology, Sandy Bridge microarchitecture; Four core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology, Processor graphics, Quicksync Video</td>
<td>3.40 GHz</td>
<td>995 M</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256</td>
<td>DMI: 5 GT/s; Memory: 21 GB/s</td>
<td>64 GB</td>
<td>L1: 64 KB L2: 256 KB L3: 8 MB</td>
</tr>
</tbody>
</table>
### Table 2-2. Key Features of Most Recent Intel® 64 Processors (Contd.)

<table>
<thead>
<tr>
<th>Intel Processor</th>
<th>Date Introduced</th>
<th>Micro-architecture</th>
<th>Highest Processor Base Frequency at Introduction</th>
<th>Transistors</th>
<th>Register Sizes</th>
<th>System Bus/QPI Link Speed</th>
<th>Max. Extern. Addr. Space</th>
<th>On-Die Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon-Processor E3-1280</td>
<td>2011</td>
<td>Intel Turbo Boost Technology, Sandy Bridge microarchitecture; Four core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>3.50 GHz</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256</td>
<td>DMI: 5 GT/s; Memory: 21 GB/s</td>
<td>1 TB</td>
<td>L1: 64 KB L2: 256 KB L3: 8 MB</td>
<td></td>
</tr>
<tr>
<td>Intel Xeon-Processor E7-8870</td>
<td>2011</td>
<td>Intel Turbo Boost Technology, Westmere microarchitecture; Ten core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.</td>
<td>2.40 GHz</td>
<td>2.2 B</td>
<td>GP: 32, 64 FPU: 80 MMX: 64 XMM: 128</td>
<td>QPI: 6.4 GT/s; Memory: 102 GB/s</td>
<td>16 TB</td>
<td>L1: 64 KB L2: 256 KB L3: 30 MB</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The 64-bit Intel Xeon Processor MP with an 8-MByte L3 supports a multi-processor platform with a dual system bus; this creates a platform bandwidth with 10.6 GBytes.
2. In Intel Pentium Processor Extreme Edition 840, the size of on-die cache is listed for each core. The total size of L2 in the physical package is 2 MBytes.
3. In Dual-Core Intel Xeon Processor 7041, the size of on-die cache is listed for each core. The total size of L2 in the physical package is 4 MBytes.
4. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.
5. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.
# Table 2-3. Key Features of Previous Generations of IA-32 Processors

<table>
<thead>
<tr>
<th>Intel Processor</th>
<th>Date Introduced</th>
<th>Max. Clock Frequency/Technology at Introduction</th>
<th>Transistors</th>
<th>Register Sizes$^1$</th>
<th>Ext. Data Bus Size$^2$</th>
<th>Max. Extern. Addr. Space</th>
<th>Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>8 MHz</td>
<td>29 K</td>
<td>16 GP</td>
<td></td>
<td>1 MB</td>
<td>None</td>
</tr>
<tr>
<td>Intel 286</td>
<td>1982</td>
<td>12.5 MHz</td>
<td>134 K</td>
<td>16 GP</td>
<td></td>
<td>16 MB</td>
<td>Note 3</td>
</tr>
<tr>
<td>Intel386 DX Processor</td>
<td>1985</td>
<td>20 MHz</td>
<td>275 K</td>
<td>32 GP</td>
<td>32</td>
<td>4 GB</td>
<td>Note 3</td>
</tr>
<tr>
<td>Intel486 DX Processor</td>
<td>1989</td>
<td>25 MHz</td>
<td>1.2 M</td>
<td>32 GP 80 FPU</td>
<td></td>
<td>4 GB</td>
<td>L1: 8 KB</td>
</tr>
<tr>
<td>Pentium Processor</td>
<td>1993</td>
<td>60 MHz</td>
<td>3.1 M</td>
<td>32 GP 80 FPU</td>
<td>64</td>
<td>4 GB</td>
<td>L1: 16 KB</td>
</tr>
<tr>
<td>Pentium Pro Processor</td>
<td>1995</td>
<td>200 MHz</td>
<td>5.5 M</td>
<td>32 GP 80 FPU 64 MMX</td>
<td></td>
<td>64 GB</td>
<td>L1: 16 KB; L2: 256 KB or 512 KB</td>
</tr>
<tr>
<td>Pentium II Processor</td>
<td>1997</td>
<td>266 MHz</td>
<td>7 M</td>
<td>32 GP 80 FPU 64 MMX</td>
<td></td>
<td>64 GB</td>
<td>L1: 32 KB; L2: 256 KB or 512 KB</td>
</tr>
<tr>
<td>Pentium III Processor</td>
<td>1999</td>
<td>500 MHz</td>
<td>8.2 M</td>
<td>32 GP 80 FPU 64 MMX</td>
<td></td>
<td>64 GB</td>
<td>L1: 32 KB; L2: 512 KB</td>
</tr>
<tr>
<td>Pentium III and Pentium III Xeon Processors</td>
<td>1999</td>
<td>700 MHz</td>
<td>28 M</td>
<td>32 GP 80 FPU 64 MMX 128 XMM</td>
<td></td>
<td>64 GB</td>
<td>L1: 32 KB; L2: 256 KB</td>
</tr>
<tr>
<td>Pentium 4 Processor</td>
<td>2000</td>
<td>1.50 GHz, Intel NetBurst Microarchitecture</td>
<td>42 M</td>
<td>32 GP 80 FPU 64 MMX 128 XMM</td>
<td></td>
<td>64 GB</td>
<td>12K µop Execution Trace Cache; L1: 8 KB; L2: 256 KB</td>
</tr>
<tr>
<td>Intel Xeon Processor</td>
<td>2001</td>
<td>1.70 GHz, Intel NetBurst Microarchitecture</td>
<td>42 M</td>
<td>32 GP 80 FPU 64 MMX 128 XMM</td>
<td></td>
<td>64 GB</td>
<td>12K µop Execution Trace Cache; L1: 8 KB; L2: 512 KB</td>
</tr>
<tr>
<td>Intel Xeon Processor</td>
<td>2002</td>
<td>2.20 GHz, Intel NetBurst Microarchitecture, HyperThreading Technology</td>
<td>55 M</td>
<td>32 GP 80 FPU 64 MMX 128 XMM</td>
<td></td>
<td>64 GB</td>
<td>12K µop Execution Trace Cache; L1: 8 KB; L2: 512 KB</td>
</tr>
<tr>
<td>Pentium M Processor</td>
<td>2003</td>
<td>1.60 GHz, Intel NetBurst Microarchitecture</td>
<td>77 M</td>
<td>32 GP 80 FPU 64 MMX 128 XMM</td>
<td></td>
<td>64 GB</td>
<td>L1: 64 KB; L2: 1 MB</td>
</tr>
</tbody>
</table>

$^1$Register Sizes: 16 GP = 16 General Purpose Registers; 32 GP = 32 General Purpose Registers; 80 FPU = 80 Floating Point Units; 64 MMX = 64 MMX Registers; 128 XMM = 128 XMM Registers

$^2$Ext. Data Bus Size: 16 = 16-bit External Data Bus; 32 = 32-bit External Data Bus

Max. Extern. Addr. Space: None = No External Address Space; 1 MB = 1 MB External Address Space; 4 GB = 4 GB External Address Space; 64 GB = 64 GB External Address Space; 4 GB = 4 GB External Address Space; 256 KB or 512 KB = 256 KB or 512 KB External Address Space
NOTE:
1. The register size and external data bus size are given in bits. Note also that each 32-bit general-purpose (GP) registers can be addressed as an 8- or a 16-bit data registers in all of the processors.
2. Internal data paths are 2 to 4 times wider than the external data bus for each processor.

2.4 PLANNED REMOVAL OF INTEL® INSTRUCTION SET ARCHITECTURE AND FEATURES FROM UPCOMING PRODUCTS

This section lists Intel Instruction Set Architecture (ISA) and features that Intel plans to remove from select products starting from a specific year.

Table 2-4. Planned Intel® ISA and Features Removal List

<table>
<thead>
<tr>
<th>Intel ISA/Feature</th>
<th>Year of Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>xAPIC mode</td>
<td>2025 onwards</td>
</tr>
</tbody>
</table>

2.5 INTEL® INSTRUCTION SET ARCHITECTURE AND FEATURES REMOVED

This section lists Intel ISA and features that Intel has already removed for select upcoming products. All sections relevant to the removed features will be identified as such and may be moved to an archived section in future Intel® 64 and IA-32 Architectures Software Developer’s Manual releases.

Table 2-5. Intel® ISA and Features Removal List

<table>
<thead>
<tr>
<th>Intel ISA/Feature</th>
<th>Year of Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Memory Protection Extensions (Intel® MPX)</td>
<td>2019 onwards</td>
</tr>
<tr>
<td>MSR_TEST_CTRL, bit 31 (MSR address 33H)</td>
<td>2019 onwards</td>
</tr>
<tr>
<td>Hardware Lock Elision (HLE)</td>
<td>2019 onwards</td>
</tr>
</tbody>
</table>
This chapter describes the basic execution environment of an Intel 64 or IA-32 processor as seen by assembly-language programmers. It describes how the processor executes instructions and how it stores and manipulates data. The execution environment described here includes memory (the address space), general-purpose data registers, segment registers, the flag register, and the instruction pointer register.

3.1 MODES OF OPERATION

The IA-32 architecture supports three basic operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

- **Protected mode** — This mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute “real-address mode” 8086 software in a protected, multi-tasking environment. This feature is called virtual-8086 mode, although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.

- **Real-address mode** — This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.

- **System management mode (SMM)** — This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC).

   In SMM, the processor switches to a separate address space while saving the basic context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386™ SL and Intel486™ SL processors and became a standard IA-32 feature with the Pentium processor family.

3.1.1 Intel® 64 Architecture

Intel 64 architecture adds IA-32e mode. IA-32e mode has two sub-modes. These are:

- **Compatibility mode (sub-mode of IA-32e mode)** — Compatibility mode permits most legacy 16-bit and 32-bit applications to run without re-compilation under a 64-bit operating system. For brevity, the compatibility sub-mode is referred to as compatibility mode in IA-32 architecture. The execution environment of compatibility mode is the same as described in Section 3.2. Compatibility mode also supports all of the privilege levels that are supported in 64-bit and protected modes. Legacy applications that run in Virtual 8086 mode or use hardware task management will not work in this mode.

   Compatibility mode is enabled by the operating system (OS) on a code segment basis. This means that a single 64-bit OS can support 64-bit applications running in 64-bit mode and support legacy 32-bit applications (not recompiled for 64-bits) running in compatibility mode.

   Compatibility mode is similar to 32-bit protected mode. Applications access only the first 4 GByte of linear-address space. Compatibility mode uses 16-bit and 32-bit address and operand sizes. Like protected mode, this mode allows applications to access physical memory greater than 4 GByte using PAE (Physical Address Extensions).

- **64-bit mode (sub-mode of IA-32e mode)** — This mode enables a 64-bit operating system to run applications written to access 64-bit linear address space. For brevity, the 64-bit sub-mode is referred to as 64-bit mode in IA-32 architecture.
64-bit mode extends the number of general purpose registers and SIMD extension registers from 8 to 16. General purpose registers are widened to 64 bits. The mode also introduces a new opcode prefix (REX) to access the register extensions. See Section 3.2.1 for a detailed description.

64-bit mode is enabled by the operating system on a code-segment basis. Its default address size is 64 bits and its default operand size is 32 bits. The default operand size can be overridden on an instruction-by-instruction basis using a REX opcode prefix in conjunction with an operand size override prefix.

REX prefixes allow a 64-bit operand to be specified when operating in 64-bit mode. By using this mechanism, many existing instructions have been promoted to allow the use of 64-bit registers and 64-bit addresses.

### 3.2 OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT

Any program or task running on an IA-32 processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (described briefly in the following paragraphs and shown in Figure 3-1) make up the basic execution environment for an IA-32 processor.

An Intel 64 processor supports the basic execution environment of an IA-32 processor, and a similar environment under IA-32e mode that can execute 64-bit programs (64-bit sub-mode) and 32-bit programs (compatibility sub-mode).

The basic execution environment is used jointly by the application programs and the operating system or executive running on the processor.

- **Address space** — Any task or program running on an IA-32 processor can address a linear address space of up to 4 GBytes (2^32 bytes) and a physical address space of up to 64 GBytes (2^36 bytes). See Section 3.3.6, “Extended Physical Addressing in Protected Mode,” for more information about addressing an address space greater than 4 GBytes.

- **Basic program execution registers** — The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on byte, word, and doubleword integers, handle program flow control, operate on bit and byte strings, and address memory. See Section 3.4, “Basic Program Execution Registers,” for more information about these registers.

- **x87 FPU registers** — The eight x87 FPU data registers, the x87 FPU control register, the status register, the x87 FPU instruction pointer register, the x87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single precision, double precision, and double extended precision floating-point values, word integers, doubleword integers, quadword integers, and binary coded decimal (BCD) values. See Section 8.1, “x87 FPU Execution Environment,” for more information about these registers.

- **MMX registers** — The eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and doubleword integers. See Section 9.2, “The MMX Technology Programming Environment,” for more information about these registers.

- **XMM registers** — The eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single precision and double precision floating-point values and on 128-bit packed byte, word, doubleword, and quadword integers. See Section 10.2, “Intel® SSE Programming Environment,” for more information about these registers.

- **YMM registers** — The YMM data registers support execution of 256-bit SIMD operations on 256-bit packed single precision and double precision floating-point values and on 256-bit packed byte, word, doubleword, and quadword integers.

- **Bounds registers** — Each of the BND0-BND3 register stores the lower and upper bounds (64 bits each) associated with the pointer to a memory buffer. They support execution of the Intel MPX instructions.

- **BNDCFGU and BNDSTATUS** — BNDCFGU configures user mode MPX operations on bounds checking. BNDSTATUS provides additional information on the #BR caused by an MPX operation.
**Figure 3-1. IA-32 Basic Execution Environment for Non-64-Bit Modes**
• **Stack** — To support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack and stack management resources are included in the execution environment. The stack (not shown in Figure 3-1) is located in memory. See Section 6.2, “Stacks,” for more information about stack structure.

In addition to the resources provided in the basic execution environment, the IA-32 architecture provides the following resources as part of its system-level architecture. They provide extensive support for operating-system and system-development software. Except for the I/O ports, the system resources are described in detail in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C & 3D.

• **I/O ports** — The IA-32 architecture supports a transfers of data to and from input/output (I/O) ports. See Chapter 19, “Input/Output,” in this volume.

• **Control registers** — The five control registers (CR0 through CR4) determine the operating mode of the processor and the characteristics of the currently executing task. See Chapter 2, “System Architecture Overview,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

• **Memory management registers** — The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management. See Chapter 2, “System Architecture Overview,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

• **Debug registers** — The debug registers (DR0 through DR7) control and allow monitoring of the processor’s debugging operations. See in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

• **Memory type range registers (MTRRs)** — The MTRRs are used to assign memory types to regions of memory. See the sections on MTRRs in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C & 3D.

• **Model-specific registers (MSRs)** — The processor provides a variety of model-specific registers that are used to control and report on processor performance. Virtually all MSRs handle system related functions and are not accessible to an application program. One exception to this rule is the time-stamp counter. The MSRs are described in Chapter 2, “Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

• **Machine check registers** — The machine check registers consist of a set of control, status, and error-reporting MSRs that are used to detect and report on hardware (machine) errors. See Chapter 16, “Machine-Check Architecture,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

• **Performance monitoring counters** — The performance monitoring counters allow processor performance events to be monitored. See Chapter 20, “Performance Monitoring,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

The remainder of this chapter describes the organization of memory and the address space, the basic program execution registers, and addressing modes. Refer to the following chapters in this volume for descriptions of the other program execution resources shown in Figure 3-1:

• **X87 FPU registers** — See Chapter 8, “Programming with the x87 FPU.”

• **MMX registers** — See Chapter 9, “Programming with Intel® MMX™ Technology.”


• **YMM registers** — See Chapter 14, “Programming with Intel® AVX, FMA, and Intel® AVX2.”

• **BND registers, BNDCFGU, BNDSTATUS** — See Chapter 13, “Managing State Using the XSAVE Feature Set,” and Appendix E, “Intel® Memory Protection Extensions.”

• **Stack implementation and procedure calls** — See Chapter 6, “Procedure Calls, Interrupts, and Exceptions.”
3.2.1 64-Bit Mode Execution Environment

The execution environment for 64-bit mode is similar to that described in Section 3.2. The following paragraphs describe the differences that apply.

- **Address space** — A task or program running in 64-bit mode on an IA-32 processor can address linear address space of up to $2^{64}$ bytes (subject to the canonical addressing requirement described in Section 3.3.7.1) and physical address space of up to $2^{52}$ bytes. Software can query CPUID for the physical address size supported by a processor.

- **Basic program execution registers** — The number of general-purpose registers (GPRs) available is 16. GPRs are 64-bits wide and they support operations on byte, word, doubleword, and quadword integers. Accessing byte registers is done uniformly to the lowest 8 bits. The instruction pointer register becomes 64 bits. The EFLAGS register is extended to 64 bits wide, and is referred to as the RFLAGS register. The upper 32 bits of RFLAGS is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS. See Figure 3-2.

- **XMM registers** — There are 16 XMM data registers for SIMD operations. See Section 10.2, “Intel® SSE Programming Environment,” for more information about these registers.

- **YMM registers** — There are 16 YMM data registers for SIMD operations. See Chapter 14, “Programming with Intel® AVX, FMA, and Intel® AVX2,” for more information about these registers.

- **BND registers, BNDCFGU, BNDSTATUS** — See Chapter 13, “Managing State Using the XSAVE Feature Set,” and Appendix E, “Intel® Memory Protection Extensions.”

- **Stack** — The stack pointer size is 64 bits. Stack size is not controlled by a bit in the SS descriptor (as it is in non-64-bit modes) nor can the pointer size be overridden by an instruction prefix.

- **Control registers** — Control registers expand to 64 bits. A new control register (the task priority register: CR8 or TPR) has been added. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in this volume.

- **Debug registers** — Debug registers expand to 64 bits. See Chapter 18, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
• **Descriptor table registers** — The global descriptor table register (GDTR) and interrupt descriptor table register (IDTR) expand to 10 bytes so that they can hold a full 64-bit base address. The local descriptor table register (LDTR) and the task register (TR) also expand to hold a full 64-bit base address.

![Diagram of Basic Program Execution Registers, FPU Registers, MMX Registers, XMM Registers, YMM Registers, and Address Space]

**Figure 3-2. 64-Bit Mode Execution Environment**

### 3.3 MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called **physical memory**. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The **physical address space** ranges from zero to a maximum of \(2^{36} - 1\) (64 GBytes) if the processor does not support Intel 64 architecture. Intel 64 architecture introduces a set of changes in physical and linear address space; these are described in Section 3.3.3, Section 3.3.4, and Section 3.3.7.
Virtually any operating system or executive designed to work with an IA-32 or Intel 64 processor will use the processor’s memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. Memory management is described in detail in Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. The following paragraphs describe the basic methods of addressing memory when memory management is used.

### 3.3.1 IA-32 Memory Models

When employing the processor’s memory management facilities, programs do not directly address physical memory. Instead, they access memory using one of three memory models: flat, segmented, or real address mode:

- **Flat memory model** — Memory appears to a program as a single, continuous address space (Figure 3-3). This space is called a linear address space. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$ (if not in 64-bit mode). An address for any byte in linear address space is called a linear address.

- **Segmented memory model** — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. Programs running on an IA-32 processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as $2^{32}$ bytes.

  Internally, all the segments that are defined for a system are mapped into the processor’s linear address space. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program.

  The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program’s stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively.

- **Real-address mode memory model** — This is the memory model for the Intel 8086 processor. It is supported to provide compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each. The maximum size of the linear address space in real-address mode is $2^{20}$ bytes.

  See also: Chapter 21, “8086 Emulation,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
3.3.2 Paging and Virtual Memory

With the flat or the segmented memory model, linear address space is mapped into the processor’s physical address space either directly or through paging. When using direct mapping (paging disabled), each linear address has a one-to-one correspondence with a physical address. Linear addresses are sent out on the processor’s address lines without translation.

When using the IA-32 architecture’s paging mechanism (paging enabled), linear address space is divided into pages which are mapped to virtual memory. The pages of virtual memory are then mapped as needed into physical memory. When an operating system or executive uses paging, the paging mechanism is transparent to an application program. All that the application sees is linear address space.

In addition, IA-32 architecture’s paging mechanism includes extensions that support:

- Physical Address Extensions (PAE) to address physical address space greater than 4 GBytes.
- Page Size Extensions (PSE) to map linear address to physical address in 4-MBytes pages.

See also: Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

3.3.3 Memory Organization in 64-Bit Mode

Intel 64 architecture supports physical address space greater than 64 GBytes; the actual physical address size of IA-32 processors is implementation specific. In 64-bit mode, there is architectural support for 64-bit linear address space. However, processors supporting Intel 64 architecture may implement less than 64-bits (see Section 3.3.7.1). The linear address space is mapped into the processor physical address space through the PAE paging mechanism.
3.3.4 Modes of Operation vs. Memory Model

When writing code for an IA-32 or Intel 64 processor, a programmer needs to know the operating mode the processor is going to be in when executing the code and the memory model being used. The relationship between operating modes and memory models is as follows:

- **Protected mode** — When in protected mode, the processor can use any of the memory models described in this section. (The real-addressing mode memory model is ordinarily used only when the processor is in the virtual-8086 mode.) The memory model used depends on the design of the operating system or executive. When multitasking is implemented, individual tasks can use different memory models.

- **Real-address mode** — When in real-address mode, the processor only supports the real-address mode memory model.

- **System management mode** — When in SMM, the processor switches to a separate address space, called the system management RAM (SMRAM). The memory model used to address bytes in this address space is similar to the real-address mode model. See Chapter 32, “System Management Mode,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information on the memory model used in SMM.

- **Compatibility mode** — Software that needs to run in compatibility mode should observe the same memory model as those targeted to run in 32-bit protected mode. The effect of segmentation is the same as it is in 32-bit protected mode semantics.

- **64-bit mode** — Segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. Specifically, the processor treats the segment base of CS, DS, ES, and SS as zero in 64-bit mode (this makes a linear address equal an effective address). Segmented and real address modes are not available in 64-bit mode.

3.3.5 32-Bit and 16-Bit Address and Operand Sizes

IA-32 processors in protected mode can be configured for 32-bit or 16-bit address and operand sizes. With 32-bit address and operand sizes, the maximum linear address or segment offset is FFFFFFFFH (2^32-1); operand sizes are typically 8 bits or 32 bits. With 16-bit address and operand sizes, the maximum linear address or segment offset is FFFF (2^16-1); operand sizes are typically 8 bits or 16 bits.

When using 32-bit addressing, a logical address (or far pointer) consists of a 16-bit segment selector and a 32-bit offset; when using 16-bit addressing, an address consists of a 16-bit segment selector and a 16-bit offset.

Instruction prefixes allow temporary overrides of the default address and/or operand sizes from within a program. When operating in protected mode, the segment descriptor for the currently executing code segment defines the default address and operand size. A segment descriptor is a system data structure not normally visible to application code. Assembler directives allow the default addressing and operand size to be chosen for a program. The assembler and other tools then set up the segment descriptor for the code segment appropriately.

When operating in real-address mode, the default addressing and operand size is 16 bits. An address-size override can be used in real-address mode to enable 32-bit addressing. However, the maximum allowable 32-bit linear address is still 000FFFFFFH (2^20-1).

3.3.6 Extended Physical Addressing in Protected Mode

Beginning with P6 family processors, the IA-32 architecture supports addressing of up to 64 GBytes (2^36 bytes) of physical memory. A program or task could not address locations in this address space directly. Instead, it addresses individual linear address spaces of up to 4 GBytes that mapped to 64-GByte physical address space through a virtual memory management mechanism. Using this mechanism, an operating system can enable a program to switch 4-GByte linear address spaces within 64-GByte physical address space.

The use of extended physical addressing requires the processor to operate in protected mode and the operating system to provide a virtual memory management system. See “36-Bit Physical Addressing Using the PAE Paging Mechanism” in Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
3.3.7 Address Calculations in 64-Bit Mode

In most cases, 64-bit mode uses flat address space for code, data, and stacks. In 64-bit mode (if there is no address-size override), the size of effective address calculations is 64 bits. An effective-address calculation uses a 64-bit base and index registers and sign-extend displacements to 64 bits.

In the flat address space of 64-bit mode, linear addresses are equal to effective addresses because the base address is zero. In the event that FS or GS segments are used with a non-zero base, this rule does not hold. In 64-bit mode, the effective address components are added and the effective address is truncated (See for example the instruction LEA) before adding the full 64-bit segment base. The base is never truncated, regardless of addressing mode in 64-bit mode.

The instruction pointer is extended to 64 bits to support 64-bit code offsets. The 64-bit instruction pointer is called the RIP. Table 3-1 shows the relationship between RIP, EIP, and IP.

<table>
<thead>
<tr>
<th>Instruction Pointer</th>
<th>Bits 63:32</th>
<th>Bits 31:16</th>
<th>Bits 15:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit instruction pointer</td>
<td>Not Modified</td>
<td></td>
<td>IP</td>
</tr>
<tr>
<td>32-bit instruction pointer</td>
<td>Zero Extension</td>
<td>EIP</td>
<td></td>
</tr>
<tr>
<td>64-bit instruction pointer</td>
<td>RIP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generally, displacements and immediates in 64-bit mode are not extended to 64 bits. They are still limited to 32 bits and sign-extended during effective-address calculations. In 64-bit mode, however, support is provided for 64-bit displacement and immediate forms of the MOV instruction.

All 16-bit and 32-bit address calculations are zero-extended in IA-32e mode to form 64-bit addresses. Address calculations are first truncated to the effective address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width. Because of this, 16-bit and 32-bit applications running in compatibility mode can access only the low 4 GBytes of the 64-bit mode effective addresses. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4 GBytes of the 64-bit mode effective addresses.

3.3.7.1 Canonical Addressing

In 64-bit mode, an address is considered to be in canonical form if address bits 63 through to the most-significant implemented bit by the microarchitecture are set to either all ones or all zeros.

Intel 64 architecture defines a 64-bit linear address. Implementations can support less. The first implementation of IA-32 processors with Intel 64 architecture supports a 48-bit linear address. This means a canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one).

Although implementations may not use all 64 bits of the linear address, they should check bits 63 through the most-significant implemented bit to see if the address is in canonical form. If a linear-memory reference is not in canonical form, the implementation should generate an exception. In most cases, a general-protection exception (#GP) is generated. However, in the case of explicit or implied stack references, a stack fault (#SS) is generated.

Instructions that have implied stack references, by default, use the SS segment register. These include PUSH/POP-related instructions and instructions using RSP/RBP as base registers. In these cases, the canonical fault is #SS.

If an instruction uses base registers RSP/RBP and uses a segment override prefix to specify a non-SS segment, a canonical fault generates a #GP (instead of an #SS). In 64-bit mode, only FS and GS segment-overrides are applicable in this situation. Other segment override prefixes (CS, DS, ES, and SS) are ignored. Note that this also means that an SS segment-override applied to a “non-stack” register reference is ignored. Such a sequence still produces a #GP for a canonical fault (and not an #SS).

3.4 BASIC PROGRAM EXECUTION REGISTERS

IA-32 architecture provides 16 basic program execution registers for use in general system and application programing (see Figure 3-4). These registers can be grouped as follows:
• **General-purpose registers.** These eight registers are available for storing operands and pointers.
• **Segment registers.** These registers hold up to six segment selectors.
• **EFLAGS (program status and control) register.** The EFLAGS register report on the status of the program being executed and allows limited (application-program level) control of the processor.
• **EIP (instruction pointer) register.** The EIP register contains a 32-bit pointer to the next instruction to be executed.

### 3.4.1 General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations.
- Operands for address calculations.
- Memory pointers.

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

![Figure 3-4. General System and Application Programming Registers](image-url)
The special uses of general-purpose registers by instructions are described in Chapter 5, “Instruction Set Summary,” in this volume. See also: Chapter 3, Chapter 4, Chapter 5, and Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D. The following is a summary of special uses:

- **EAX** — Accumulator for operands and results data.
- **EBX** — Pointer to data in the DS segment.
- **ECX** — Counter for string and loop operations.
- **EDX** — I/O pointer.
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- **ESP** — Stack pointer (in the SS segment).
- **EBP** — Pointer to data on the stack (in the SS segment).

As shown in Figure 3-5, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

![General-Purpose Registers](image)

**Figure 3-5. Alternate General-Purpose Register Names**

### 3.4.1.1 General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers. All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.
In 64-bit mode, there are limitations on accessing byte registers. An instruction cannot reference legacy high-bytes (for example: AH, BH, CH, DH) and one of the new byte registers at the same time (for example: the low byte of the RAX register). However, instructions may reference legacy low-bytes (for example: AL, BL, CL, or DL) and new byte registers at the same time (for example: the low byte of the R8 register, or RBP). The architecture enforces this limitation by changing high-byte references (AH, BH, CH, DH) to low byte references (BPL, SPL, DIL, SIL: the low 8 bits for RBP, RSP, RDI, and RSI) for instructions using a REX prefix.

When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.

Because the upper 32 bits of 64-bit general-purpose registers are undefined in 32-bit modes, the upper 32 bits of any general-purpose register are not preserved when switching from 64-bit mode to a 32-bit mode (to protected mode or compatibility mode). Software must not depend on these bits to maintain a value after a 64-bit to 32-bit mode switch.

### 3.4.2 Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

When writing application code, programmers generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If writing system code, programmers may need to create segment selectors directly. See Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (see Figure 3-6). These overlapping segments then comprise the linear address space for the program. Typically, two overlapping segments are defined: one for code and another for data and stacks. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.

When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space (see Figure 3-7). At any time, a program can thus access up to six segments in the linear address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

### Table 3-2. Addressable General Purpose Registers

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Without REX</th>
<th>With REX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Registers</td>
<td>AL, BL, CL, DL, AH, BH, CH, DH</td>
<td>AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8B - R15B</td>
</tr>
<tr>
<td>Word Registers</td>
<td>AX, BX, CX, DX, DI, SI, BP, SP</td>
<td>AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W</td>
</tr>
<tr>
<td>Doubleword Registers</td>
<td>EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP</td>
<td>EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D</td>
</tr>
<tr>
<td>Quadword Registers</td>
<td>N.A.</td>
<td>RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15</td>
</tr>
</tbody>
</table>
Each of the segment registers is associated with one of three types of storage: code, data, or stack. For example, the CS register contains the segment selector for the **code segment**, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four **data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for the **stack segment**, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack.
segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

See Section 3.3, “Memory Organization,” for an overview of how the segment registers are used in real-address mode.

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the IA-32 Architecture with the Intel386™ family of processors.

### 3.4.2.1 Segment Registers in 64-Bit Mode

In 64-bit mode: CS, DS, ES, SS are treated as if each segment base is 0, regardless of the value of the associated segment descriptor base. This creates a flat address space for code, data, and stack. FS and GS are exceptions. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures).

Even though segmentation is generally disabled, segment register loads may cause the processor to perform segment access assists. During these activities, enabled processors will still perform most of the legacy checks on loaded values (even if the checks are not applicable in 64-bit mode). Such checks are needed because a segment register loaded in 64-bit mode may be used by an application running in compatibility mode.

Limit checks for CS, DS, ES, SS, FS, and GS are disabled in 64-bit mode.

### 3.4.3 EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 3-8 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly.

The following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor’s bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor’s multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task’s TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.
As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

### 3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- **CF (bit 0)**  
  **Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

- **PF (bit 2)**  
  **Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

- **AF (bit 4)**  
  **Auxiliary Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

- **ZF (bit 6)**  
  **Zero flag** — Set if the result is zero; cleared otherwise.

- **SF (bit 7)**  
  **Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

- **OF (bit 11)**  
  **Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two’s complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.
The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two’s complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions Jcc (jump on condition code cc), SETcc (byte set on condition code cc), LOOPcc, and CMOVcc (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

### 3.4.3.2 DF Flag

The direction flag (DF, located in bit 10 of the EFLAGS register) controls string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

### 3.4.3.3 System Flags and IOPL Field

The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. **They should not be modified by application programs.** The functions of the system flags are as follows:

- **TF (bit 8)** Trap flag — Set to enable single-step mode for debugging; clear to disable single-step mode.
- **IF (bit 9)** Interrupt enable flag — Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.
- **IOPL (bits 12 and 13)**
  - I/O privilege level field — Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.
- **NT (bit 14)** Nested task flag — Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.
- **RF (bit 16)** Resume flag — Controls the processor's response to debug exceptions.
- **VM (bit 17)** Virtual-8086 mode flag — Set to enable virtual-8086 mode; clear to return to protected mode without virtual-8086 mode semantics.
- **AC (bit 18)** Alignment check (or access control) flag — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1. See Section 4.6, "Access Rights," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
- **VIF (bit 19)** Virtual interrupt flag — Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)
- **VIP (bit 20)** Virtual interrupt pending flag — Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.
- **ID (bit 21)** Identification flag — The ability of a program to set or clear this flag indicates support for the CPUID instruction.
3.4.3.4  RFLAGS Register in 64-Bit Mode

In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.

3.5  INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). See Section 6.2.4.2, “Return Instruction Pointer.”

All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of the EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

3.5.1  Instruction Pointer in 64-Bit Mode

In 64-bit mode, the RIP register becomes the instruction pointer. This register holds the 64-bit offset of the next instruction to be executed. 64-bit mode also supports a technique called RIP-relative addressing. Using this technique, the effective address is determined by adding a displacement to the RIP of the next instruction.

3.6  OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES

When the processor is executing in protected mode, every code segment has a default operand-size attribute and address-size attribute. These attributes are selected with the D (default size) flag in the segment descriptor for the code segment (see Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). When the D flag is set, the 32-bit operand-size and address-size attributes are selected; when the flag is clear, the 16-bit size attributes are selected. When the processor is executing in real-address mode, virtual-8086 mode, or SMM, the default operand-size and address-size attributes are always 16 bits.

The operand-size attribute selects the size of operands. When the 16-bit operand-size attribute is in force, operands can generally be either 8 bits or 16 bits, and when the 32-bit operand-size attribute is in force, operands can generally be 8 bits or 32 bits.

The address-size attribute selects the sizes of addresses used to address memory: 16 bits or 32 bits. When the 16-bit address-size attribute is in force, segment offsets and displacements are 16 bits. This restriction limits the size of a segment to 64 KBytes. When the 32-bit address-size attribute is in force, segment offsets and displacements are 32 bits, allowing up to 4 GBytes to be addressed.

The default operand-size attribute and/or address-size attribute can be overridden for a particular instruction by adding an operand-size and/or address-size prefix to an instruction. See Chapter 2, “Instruction Format,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. The effect of this prefix applies only to the targeted instruction.

Table 3-4 shows effective operand size and address size (when executing in protected mode or compatibility mode) depending on the settings of the D flag and the operand-size and address-size prefixes.
3.6.1 Operand Size and Address Size in 64-Bit Mode

In 64-bit mode, the default address size is 64 bits and the default operand size is 32 bits. Defaults can be over-ridden using prefixes. Address-size and operand-size prefixes allow mixing of 32/64-bit data and 32/64-bit addresses on an instruction-by-instruction basis. Table 3-4 shows valid combinations of the 66H instruction prefix and the REX.W prefix that may be used to specify operand-size overrides in 64-bit mode. Note that 16-bit addresses are not supported in 64-bit mode.

REX prefixes consist of 4-bit fields that form 16 different values. The W-bit field in the REX prefixes is referred to as REX.W. If the REX.W field is properly set, the prefix specifies an operand size override to 64 bits. Note that software can still use the operand-size 66H prefix to toggle to a 16-bit operand size. However, setting REX.W takes precedence over the operand-size prefix (66H) when both are used.

In the case of SSE/SSE2/SSE3/SSSE3 SIMD instructions: the 66H, F2H, and F3H prefixes are mandatory for opcode extensions. In such a case, there is no interaction between a valid REX.W prefix and a 66H opcode extension prefix.


<table>
<thead>
<tr>
<th>Table 3-3. Effective Operand- and Address-Size Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Flag in Code Segment Descriptor</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>Operand-Size Prefix 66H</td>
</tr>
<tr>
<td>Address-Size Prefix 67H</td>
</tr>
<tr>
<td>Effective Operand Size</td>
</tr>
<tr>
<td>Effective Address Size</td>
</tr>
</tbody>
</table>

NOTES:
Y: Yes - this instruction prefix is present.
N: No - this instruction prefix is not present.

<table>
<thead>
<tr>
<th>Table 3-4. Effective Operand- and Address-Size Attributes in 64-Bit Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>L Flag in Code Segment Descriptor</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>REX.W Prefix</td>
</tr>
<tr>
<td>Operand-Size Prefix 66H</td>
</tr>
<tr>
<td>Address-Size Prefix 67H</td>
</tr>
<tr>
<td>Effective Operand Size</td>
</tr>
<tr>
<td>Effective Address Size</td>
</tr>
</tbody>
</table>

NOTES:
Y: Yes - this instruction prefix is present.
N: No - this instruction prefix is not present.

3.7 OPERAND ADDRESSING

IA-32 machine-instructions act on zero or more operands. Some operands are specified explicitly and others are implicit. The data for a source operand can be located in:
- The instruction itself (an immediate operand).
- A register.
- A memory location.
- An I/O port.
When an instruction returns data to a destination operand, it can be returned to:

- A register.
- A memory location.
- An I/O port.

### 3.7.1 Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called immediate operands (or simply immediates). For example, the following ADD instruction adds an immediate value of 14 to the contents of the EAX register:

```
ADD EAX, 14
```

All arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer ($2^{32}$).

### 3.7.2 Register Operands

Source and destination operands can be any of the following registers, depending on the instruction being executed:

- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP).
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP).
- 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL).
- Segment registers (CS, DS, SS, ES, FS, and GS).
- EFLAGS register.
- X87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer).
- MMX registers (MM0 through MM7).
- XMM registers (XMM0 through XMM7) and the MXCSR register.
- Control registers (CR0, CR2, CR3, and CR4) and system table pointer registers (GDTR, LDTR, IDTR, and task register).
- Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7).
- MSR registers.

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the Jcc instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.
3.7.2.1  Register Operands in 64-Bit Mode

Register operands in 64-bit mode can be any of the following:

- 64-bit general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, or R8-R15).
- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, or R8D-R15D).
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, BP, or R8W-R15W).
- 8-bit general-purpose registers: AL, BL, CL, DL, SIL, DIL, SPL, BPL, and R8B-R15B are available using REX prefixes; AL, BL, CL, DL, AH, BH, CH, DH are available without using REX prefixes.
- Segment registers (CS, DS, SS, ES, FS, and GS).
- RFLAGS register.
- X87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer).
- MMX registers (MM0 through MM7).
- XMM registers (XMM0 through XMM15) and the MXCSR register.
- Control registers (CR0, CR2, CR3, CR4, and CR8) and system table pointer registers (GDTR, LDTR, IDTR, and task register).
- Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7).
- MSR registers.
- RDX:RAX register pair representing a 128-bit operand.

3.7.3  Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset (see Figure 3-9). Segment selectors specify the segment containing the operand. Offsets specify the linear or effective address of the operand. Offsets can be 32 bits (represented by the notation m16:32) or 16 bits (represented by the notation m16:16).

![Figure 3-9. Memory Operand Address](image)

3.7.3.1  Memory Operands in 64-Bit Mode

In 64-bit mode, a memory operand can be referenced by a segment selector and an offset. The offset can be 16 bits, 32 bits or 64 bits (see Figure 3-10).

![Figure 3-10. Memory Operand Address in 64-Bit Mode](image)

3.7.4  Specifying a Segment Selector

The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the rules given in Table 3-5.
When storing data in memory or loading data from memory, the DS segment default can be overridden to allow other segments to be accessed. Within an assembler, the segment override is generally handled with a colon "::" operator. For example, the following MOV instruction moves a value from register EAX into the segment pointed to by the ES register. The offset into the segment is contained in the EBX register:

MOV ES[EBX], EAX

### Table 3-5. Default Segment Selection Rules

<table>
<thead>
<tr>
<th>Reference Type</th>
<th>Register Used</th>
<th>Segment Used</th>
<th>Default Selection Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Stack</td>
<td>CS, SS</td>
<td>Code Segment, Stack Segment</td>
<td>All instruction fetches. All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.</td>
</tr>
<tr>
<td>Local Data</td>
<td>DS, ES</td>
<td>Data Segment, Data Segment pointed to with the ES register</td>
<td>All data references, except when relative to stack or string destination. Destination of string instructions.</td>
</tr>
</tbody>
</table>

At the machine level, a segment override is specified with a segment-override prefix, which is a byte placed at the beginning of an instruction. The following default segment selections cannot be overridden:

- Instruction fetches must be made from the code segment.
- Destination strings in string instructions must be stored in the data segment pointed to by the ES register.
- Push and pop operations must always reference the SS segment.

Some instructions require a segment selector to be specified explicitly. In these cases, the 16-bit segment selector can be located in a memory location or in a 16-bit register. For example, the following MOV instruction moves a segment selector located in register BX into segment register DS:

MOV DS, BX

Segment selectors can also be specified explicitly as part of a 48-bit far pointer in memory. Here, the first double-word in memory contains the offset and the next word contains the segment selector.

#### 3.7.4.1 Segmentation in 64-Bit Mode

In IA-32e mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does in legacy IA-32 mode, using the 16-bit or 32-bit protected mode semantics described above.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The exceptions are the FS and GS segments, whose segment registers (which hold the segment base) can be used as additional base registers in some linear address calculations.

#### 3.7.5 Specifying an Offset

The offset part of a memory address can be specified directly as a static value (called a displacement) or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-, 16-, or 32-bit value.
- **Base** — The value in a general-purpose register.
- **Index** — The value in a general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.
The offset which results from adding these components is called an **effective address**. Each of these components can have either a positive or negative (2s complement) value, with the exception of the scaling factor. Figure 3-11 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

The uses of general-purpose registers as base or index components are restricted in the following manner:

- **The ESP register cannot be used as an index register.**
- **When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.**

The base, index, and displacement components can be used in any combination, and any of these components can be NULL. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language.

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.

- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

- **(Index × Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

- **Base + (Index × Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.
3.7.5.1 Specifying an Offset in 64-Bit Mode

The offset part of a memory address in 64-bit mode can be specified directly as a static value or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-bit, 16-bit, or 32-bit value.
- **Base** — The value in a 64-bit general-purpose register.
- **Index** — The value in a 64-bit general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.

The base and index value can be specified in one of sixteen available general-purpose registers in most cases. See Chapter 2, "Instruction Format," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

The following unique combination of address components is also available.

- **RIP + Displacement** — In 64-bit mode, RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extend the 32-bit value and add to the 64-bit value in RIP.

3.7.6 Assembler and Compiler Addressing Modes

At the machine-code level, the selected combination of displacement, base register, index register, and scale factor is encoded in an instruction. All assemblers permit a programmer to use any of the allowable combinations of these addressing components to address operands. High-level language compilers will select an appropriate combination of these components based on the language construct a programmer defines.

3.7.7 I/O Port Addressing

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register. See Chapter 19, “Input/Output,” for more information about I/O port addressing.
This chapter introduces data types defined for the Intel 64 and IA-32 architectures. A section at the end of this chapter describes the real-number and floating-point concepts used in x87 FPU and Intel SSE, SSE2, SSE3, SSSE3, SSE4, and AVX extensions.

4.1 FUNDAMENTAL DATA TYPES

The fundamental data types are bytes, words, doublewords, quadwords, and double quadwords (see Figure 4-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), a quadword is 8 bytes (64 bits), and a double quadword is 16 bytes (128 bits). A subset of the IA-32 architecture instructions operates on these fundamental data types without any additional operand typing.

The quadword data type was introduced into the IA-32 architecture in the Intel 486 processor; the double quadword data type was introduced in the Pentium III processor with the Intel SSE extensions.

Figure 4-2 shows the byte order of each of the fundamental data types when referenced as operands in memory. The low byte (bits 0 through 7) of each data type occupies the lowest address in memory and that address is also the address of the operand.
### DATA TYPES

#### 4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, doublewords, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

Some instructions that operate on double quadwords require memory operands to be aligned on a natural boundary. These instructions generate a general-protection exception (#GP) if an unaligned operand is specified. A natural boundary for a double quadword is any address evenly divisible by 16. Other instructions that operate on double quadwords permit unaligned access (without generating a general-protection exception). However, additional memory bus cycles are required to access unaligned data from memory.

#### 4.2 NUMERIC DATA TYPES

Although bytes, words, and doublewords are fundamental data types, some instructions support additional interpretations of these data types to allow operations to be performed on numeric data types (signed and unsigned integers, and floating-point numbers). Single precision (32-bit) floating-point and double precision (64-bit) floating-point data types are supported across all generations of Intel SSE extensions and Intel AVX extensions. The half precision (16-bit) floating-point data type was supported only with F16C extensions (VCVTPH2PS and VCVTPS2PH) beginning with the third generation of Intel® Core™ processors based on Ivy Bridge microarchitecture. Starting with the 4th generation Intel® Xeon® Scalable Processor Family, an Intel® AVX-512 instruction set architecture (ISA) for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16), which complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products. This ISA also provides complex-valued native hardware support for half precision floating-point. See Figure 4-3.

---

**Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory**

Word at Address BH Contains FE06H

Byte at Address 9H Contains 1FH

Word at Address 6H Contains 230BH

Word at Address 2H Contains 74CBH

Word at Address 1H Contains CB31H

Doubleword at Address AH Contains 7AFE0636H

Quadword at Address 6H Contains 7AFE06361FA4230BH

Double quadword at Address 0H Contains 4E127AFE06361FA4230B456774CB311.
4.2.1 Integers

The Intel 64 and IA-32 architectures define two types of integers: unsigned and signed. Unsigned integers are ordinary binary values ranging from 0 to the maximum positive number that can be encoded in the selected operand size. Signed integers are two’s complement binary values that can be used to represent both positive and negative integer values.

Some integer instructions (such as the ADD, SUB, PADD, and PSUB instructions) operate on either unsigned or signed integer operands. Other integer instructions (such as IMUL, MUL, IDIV, DIV, FIADD, and FISUB) operate on only one integer type.

The following sections describe the encodings and ranges of the two types of integers.

4.2.1.1 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, word, doubleword, and quadword. Their values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned word integer, from 0 to 4,294,967,295 for an unsigned doubleword integer, and from 0 to 18,446,744,073,709,551,615 for an unsigned quadword integer.
to $2^{32} - 1$ for an unsigned doubleword integer, and from 0 to $2^{64} - 1$ for an unsigned quadword integer. Unsigned integers are sometimes referred to as **ordinals**.

### 4.2.1.2 Signed Integers

Signed integers are signed binary numbers held in a byte, word, doubleword, or quadword. All operations on signed integers assume a two’s complement representation. The sign bit is located in bit 7 in a byte integer, bit 15 in a word integer, bit 31 in a doubleword integer, and bit 63 in a quadword integer (see the signed integer encodings in Table 4-1).

#### Table 4-1. Signed Integer Encodings

<table>
<thead>
<tr>
<th>Class</th>
<th>Two’s Complement Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign</td>
</tr>
<tr>
<td>Positive</td>
<td></td>
</tr>
<tr>
<td>Largest</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Smallest</td>
<td>0</td>
</tr>
<tr>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>Negative</td>
<td></td>
</tr>
<tr>
<td>Smallest</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Largest</td>
<td>1</td>
</tr>
<tr>
<td>Integer indefinite</td>
<td>1</td>
</tr>
</tbody>
</table>

The sign bit is set for negative integers and cleared for positive integers and zero. Integer values range from –128 to +127 for a byte integer, from –32,768 to +32,767 for a word integer, from –$2^{31}$ to $+2^{31} - 1$ for a doubleword integer, and from –$2^{63}$ to $+2^{63} - 1$ for a quadword integer.

When storing integer values in memory, word integers are stored in 2 consecutive bytes; doubleword integers are stored in 4 consecutive bytes; and quadword integers are stored in 8 consecutive bytes.

The integer indefinite is a special value that is sometimes returned by the x87 FPU when operating on integer values. For more information, see Section 8.2.1, “Indefinites.”

### 4.2.2 Floating-Point Data Types

The IA-32 architecture defines and operates on four floating-point data types: half precision floating-point, single precision floating-point, double precision floating-point, and double-extended precision floating-point (see Figure 4-3). The data formats for these data types correspond directly to formats specified in the IEEE Standard 754 for Floating-Point Arithmetic.

The half precision (16-bit) floating-point data type was supported only with F16C extensions (VCVTPH2PS and VCVTPS2PH) beginning with the third generation of Intel Core processors based on Ivy Bridge microarchitecture. Starting with the 4th generation Intel Xeon Scalable Processor Family, an Intel AVX-512 instruction set architecture (ISA) for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in the IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16), which complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products.
Table 4-2 gives the length, precision, and approximate normalized range that can be represented by each of these data types. Denormal values are also supported in each of these types.

### Table 4-2. Length, Precision, and Range of Floating-Point Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Length (Bits)</th>
<th>Precision (Bits)</th>
<th>Approximate Normalized Range</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Binary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decimal</td>
</tr>
<tr>
<td>Half Precision</td>
<td>16</td>
<td>11</td>
<td>$2^{-14}$ to $2^{16}$</td>
</tr>
<tr>
<td>Single Precision</td>
<td>32</td>
<td>24</td>
<td>$2^{-126}$ to $2^{128}$</td>
</tr>
<tr>
<td>Double Precision</td>
<td>64</td>
<td>53</td>
<td>$2^{-1022}$ to $2^{1024}$</td>
</tr>
<tr>
<td>Double-Extended Precision</td>
<td>80</td>
<td>64</td>
<td>$2^{-16382}$ to $2^{16384}$</td>
</tr>
</tbody>
</table>

**NOTE**

Section 4.8, “Real Numbers and Floating-Point Formats,” gives an overview of the IEEE Standard 754 floating-point formats and defines the terms integer bit, QNaN, SNaN, and denormal value.

Table 4-3 shows the floating-point encodings for zeros, denormalized finite numbers, normalized finite numbers, infinites, and NaNs for each of the three floating-point data types. It also gives the format for the QNaN floating-point indefinite value. (See Section 4.8.3.7, “QNaN Floating-Point Indefinite,” for a discussion of the use of the QNaN floating-point indefinite value.)

For the half precision, single precision, and double precision formats, only the fraction part of the significand is encoded. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers. For the double extended precision format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinites, and NaNs, and to 0 for zero and denormalized numbers.

### Table 4-3. Floating-Point Number and NaN Encodings

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Biased Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer¹</td>
</tr>
<tr>
<td>Positive</td>
<td>+∞</td>
<td>0 11.11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>+Normals</td>
<td>0</td>
<td>11.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 00.01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>+Denormals</td>
<td>0</td>
<td>00.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 00.00</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>+Zero</td>
<td>0</td>
<td>00.00</td>
</tr>
<tr>
<td>Negative</td>
<td>−Zero</td>
<td>1</td>
<td>00.00</td>
</tr>
<tr>
<td></td>
<td>−Denormals</td>
<td>1</td>
<td>00.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 00.00</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>−Normals</td>
<td>1</td>
<td>00.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 11.10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>−∞</td>
<td>1 11.11</td>
<td>1</td>
</tr>
</tbody>
</table>
DATA TYPES

4.3 POINTER DATA TYPES

Pointers are addresses of locations in memory.

In non-64-bit modes, the architecture defines two types of pointers: a near pointer and a far pointer. A near pointer is a 32-bit (or 16-bit) offset (also called an effective address) within a segment. Near pointers are used for all memory references in a flat memory model or for references in a segmented model where the identity of the segment being accessed is implied.

A far pointer is a logical address, consisting of a 16-bit segment selector and a 32-bit (or 16-bit) offset. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly. Near and far pointers with 32-bit offsets are shown in Figure 4-4.
4.3.1 Pointer Data Types in 64-Bit Mode

In 64-bit mode (a sub-mode of IA-32e mode), a near pointer is 64 bits. This equates to an effective address. Far pointers in 64-bit mode can be one of three forms:

- 16-bit segment selector, 16-bit offset if the operand size is 32 bits.
- 16-bit segment selector, 32-bit offset if the operand size is 32 bits.
- 16-bit segment selector, 64-bit offset if the operand size is 64 bits.

See Figure 4-5.

4.4 BIT FIELD DATA TYPE

A bit field (see Figure 4-6) is a contiguous sequence of bits. It can begin at any bit position of any byte in memory and can contain up to 32 bits.
4.5 STRING DATA TYPES

Strings are continuous sequences of bits, bytes, words, or doublewords. A **bit string** can begin at any bit position of any byte and can contain up to $2^{32} - 1$ bits. A **byte string** can contain bytes, words, or doublewords and can range from zero to $2^{32} - 1$ bytes (4 GBytes).

4.6 PACKED SIMD DATA TYPES

Intel 64 and IA-32 architectures define and operate on a set of 64-bit and 128-bit packed data type for use in SIMD operations. These data types consist of fundamental data types (packed bytes, words, doublewords, and quadwords) and numeric interpretations of fundamental types for use in packed integer and packed floating-point operations.

4.6.1 64-Bit SIMD Packed Data Types

The 64-bit packed SIMD data types were introduced into the IA-32 architecture in the Intel MMX technology. They are operated on in MMX registers. The fundamental 64-bit packed data types are packed bytes, packed words, and packed doublewords (see Figure 4-7). When performing numeric SIMD operations on these data types, these data types are interpreted as containing byte, word, or doubleword integer values.

![Figure 4-7. 64-Bit Packed SIMD Data Types](image)

4.6.2 128-Bit Packed SIMD Data Types

The 128-bit packed SIMD data types were introduced into the IA-32 architecture in the Intel SSE extensions and used with Intel SSE2, SSE3, SSSE3, SSE4.1, and AVX extensions. They are operated on primarily in the 128-bit XMM registers and memory. The fundamental 128-bit packed data types are packed bytes, packed words, packed doublewords, and packed quadwords (see Figure 4-8). When performing SIMD operations on these fundamental data types in XMM registers, these data types are interpreted as containing packed or scalar half precision floating-point, single precision floating-point or double precision floating-point values, or as containing packed byte, word, doubleword, or quadword integer values.
4.7 BCD AND PACKED BCD INTEGERS

Binary-coded decimal integers (BCD integers) are unsigned 4-bit integers with valid values ranging from 0 to 9. IA-32 architecture defines operations on BCD integers located in one or more general-purpose registers or in one or more x87 FPU registers (see Figure 4-9).
DATA TYPES

When operating on BCD integers in general-purpose registers, the BCD values can be unpacked (one BCD digit per byte) or packed (two BCD digits per byte). The value of an unpacked BCD integer is the binary value of the low half-byte (bits 0 through 3). The high half-byte (bits 4 through 7) can be any value during addition and subtraction, but must be zero during multiplication and division. Packed BCD integers allow two BCD digits to be contained in one byte. Here, the digit in the high half-byte is more significant than the digit in the low half-byte.

When operating on BCD integers in x87 FPU data registers, BCD values are packed in an 80-bit format and referred to as decimal integers. In this format, the first 9 bytes hold 18 BCD digits, 2 digits per byte. The least-significant digit is contained in the lower half-byte of byte 0 and the most-significant digit is contained in the upper half-byte of byte 9. The most significant bit of byte 10 contains the sign bit (0 = positive and 1 = negative; bits 0 through 6 of byte 10 are don’t care bits). Negative decimal integers are not stored in two’s complement form; they are distinguished from positive decimal integers only by the sign bit. The range of decimal integers that can be encoded in this format is $-10^{18} + 1$ to $10^{18} - 1$.

The decimal integer format exists in memory only. When a decimal integer is loaded in an x87 FPU data register, it is automatically converted to the double extended precision floating-point format. All decimal integers are exactly representable in double extended precision format.

Table 4-4 gives the possible encodings of value in the decimal integer data type.

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>digit</td>
</tr>
<tr>
<td>Positive</td>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>Largest</td>
<td></td>
<td>0000000</td>
</tr>
<tr>
<td>Smallest</td>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td>0000000</td>
</tr>
<tr>
<td>Negative</td>
<td>1</td>
<td>0000000</td>
</tr>
<tr>
<td>Zero</td>
<td>1</td>
<td>0000000</td>
</tr>
<tr>
<td>Smallest</td>
<td></td>
<td>0000000</td>
</tr>
<tr>
<td>Largest</td>
<td>1</td>
<td>0000000</td>
</tr>
</tbody>
</table>
The packed BCD integer indefinite encoding (FFFFC000000000000000H) is stored by the FBSTP instruction in response to a masked floating-point invalid-operation exception. Attempting to load this value with the FBLD instruction produces an undefined result.

### 4.8 REAL NUMBERS AND FLOATING-POINT FORMATS

This section describes how real numbers are represented in floating-point format in x87 FPU and SSE/SSE2/SSE3/SSE4.1 and Intel AVX floating-point instructions. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and the IEEE Standard 754 for Floating-Point Arithmetic may wish to skip this section.

#### 4.8.1 Real Number System

As shown in Figure 4-10, the real-number system comprises the continuum of real numbers from minus infinity (−∞) to plus infinity (+∞).

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number (floating-point) calculations. As shown at the bottom of Figure 4-10, the subset of real numbers that the IA-32 architecture supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the IEEE Standard 754 floating-point formats.

#### 4.8.2 Floating-Point Format

To increase the speed and efficiency of real-number computations, computers and microprocessors typically represent real numbers in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent (see Figure 4-11).

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The integer-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power by which the significand is multiplied.

Table 4-5 shows how the real number 178.125 (in ordinary decimal format) is stored in IEEE Standard 754 floating-point format. The table lists a progression of real number notations that leads to the single precision, 32-bit floating-point format. In this format, the significand is normalized (see Section 4.8.2.1, “Normalized Numbers”) and the exponent is biased (see Section 4.8.2.2, “Biased Exponent”). For the single precision floating-point format, the biasing constant is +127.
**Figure 4-10. Binary Real Number System**

Subset of binary real numbers that can be represented with IEEE single precision (32-bit) floating-point format

**Figure 4-11. Binary Floating-Point Format**

**Table 4-5. Real and Floating-Point Number Notation**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordinary Decimal</td>
<td>178.125</td>
</tr>
<tr>
<td>Scientific Decimal</td>
<td>1.78125E10</td>
</tr>
<tr>
<td>Scientific Binary</td>
<td>1.0110010001E2111</td>
</tr>
<tr>
<td>Scientific Binary (Biased Exponent)</td>
<td>1.0110010001E21110</td>
</tr>
<tr>
<td>IEEE Single Precision Format</td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>0</td>
</tr>
<tr>
<td>Biased Exponent</td>
<td>10000110</td>
</tr>
<tr>
<td>Normalized Significand</td>
<td>011001000100000000000000000000000000000001 (Implied)</td>
</tr>
</tbody>
</table>
4.8.2.1 Normalized Numbers

In most cases, floating-point numbers are encoded in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

\[
1.f\ldots f
\]

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number’s binary point.

4.8.2.2 Biased Exponent

In the IA-32 architecture, the exponents of floating-point numbers are encoded in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

See Section 4.2.2, "Floating-Point Data Types," for a list of the biasing constants that the IA-32 architecture uses for the various sizes of floating-point data-types.

4.8.3 Real Number and Non-number Encodings

A variety of real numbers and special values can be encoded in the IEEE Standard 754 floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros
- Denormalized finite numbers
- Normalized finite numbers
- Signed infinities
- NaNs
- Indefinite numbers

(The term NaN stands for "Not a Number.")

Figure 4-12 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single precision floating-point format. The term "S" indicates the sign bit, "E" the biased exponent, and "Sig" the significand. The exponent values are given in decimal. The integer bit is shown for the significands, even though the integer bit is implied in single precision floating-point format.
DATA TYPES

An IA-32 processor can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

4.8.3.1  Signed Zeros

Zero can be represented as a +0 or a −0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an $\infty$ that has been reciprocated.

4.8.3.2  Normalized and Denormalized Finite Numbers

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and $\infty$. In the single precision floating-point format shown in Figure 4-12, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254 ($0\leq E < 255$).

When floating-point numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called denormalized numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization may cause a loss of precision (the number of significant bits is reduced by the leading zeros).

When performing normalized floating-point computations, an IA-32 processor normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an underflow condition. The exact conditions are specified in Section 4.9.1.5, “Numeric Underflow Exception (#U).”

A denormalized number is computed through a technique called gradual underflow. Table 4-6 gives an example of gradual underflow in the denormalization process. Here the single precision format is being used, so the minimum exponent (unbiased) is $-126_{10}$. The true result in this example requires an exponent of $-129_{10}$ in order to have a
normalized number. Since \(-129\) is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of \(-126\) is reached.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The Intel 64 and IA-32 architectures deal with denormal values in the following ways:
- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

### 4.8.3.3 Signed Infinities

The two infinities, \(+\infty\) and \(-\infty\), represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a significand of 1.00...00 (the integer bit may be implied) and the maximum biased exponent allowed in the specified format (for example, 255 for the single precision format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, \(-\infty\) is less than any finite number and \(+\infty\) is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers may represent an underflow condition, the two \(\infty\) numbers may represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

### 4.8.3.4 NaNs

Since NaNs are non-numbers, they are not part of the real number line. In Figure 4-12, the encoding space for NaNs in the floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction (the sign bit is ignored for NaNs).

The IA-32 architecture defines two classes of NaNs: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal a floating-point invalid-operation exception whenever they appear as operands in arithmetic operations.

SNaNs are typically used to trap or invoke an exception handler. They must be inserted by software; that is, the processor never generates an SNaN as a result of a floating-point operation.

---

**Table 4-6. Denormalization Process**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sign</th>
<th>Exponent*</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Result</td>
<td>0</td>
<td>-129</td>
<td>1.01011100000...00</td>
</tr>
<tr>
<td>Denormalize</td>
<td>0</td>
<td>-128</td>
<td>0.10101110000...00</td>
</tr>
<tr>
<td>Denormalize</td>
<td>0</td>
<td>-127</td>
<td>0.01010111000...00</td>
</tr>
<tr>
<td>Denormalize</td>
<td>0</td>
<td>-126</td>
<td>0.00101011100...00</td>
</tr>
<tr>
<td>Denormal Result</td>
<td>0</td>
<td>-126</td>
<td>0.00101011100...00</td>
</tr>
</tbody>
</table>

* Expressed as an unbiased, decimal number.
4.8.3.5 Operating on SNaNs and QNaNs

When a floating-point operation is performed on an SNaN and/or a QNaN, the result of the operation is either a QNaN delivered to the destination operand or the generation of a floating-point invalid operation exception, depending on the following rules:

- If one of the source operands is an SNaN and the floating-point invalid-operation exception is not masked (see Section 4.9.1.1, "Invalid Operation Exception (#I)"), then a floating-point invalid-operation exception is signaled and no result is stored in the destination operand. If one of the source operands is a QNaN and the floating-point invalid-operation exception is not masked and the operation is one that generates an invalid-operation exception for QNaN operands as described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," or Section 11.5.2.1, "Invalid Operation Exception (#I)," then a floating-point invalid-operation exception is signaled and no result is stored in the destination operand.

- If either or both of the source operands are NaNs and floating-point invalid-operation exception is masked, the result is as shown in Table 4-7. When an SNaN is converted to a QNaN, the conversion is handled by setting the most-significant fraction bit of the SNaN to 1. Also, when one of the source operands is an SNaN, or when it is a QNaN and the operation is one that generates an invalid-operation exception for QNaN operands as described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," or Section 11.5.2.1, "Invalid Operation Exception (#I)," then the floating-point invalid-operation exception flag is set. Note that for some combinations of source operands, the result is different for x87 FPU operations and for Intel SSE/SSE2/SSE3/SSE4.1 operations. Intel AVX follows the same behavior as Intel SSE/SSE2/SSE3/SSE4.1 in this respect.

- When neither of the source operands is a NaN, but the operation generates a floating-point invalid-operation exception (see Tables 8-10 and 11-1), the result is commonly a QNaN FP Indefinite (Section 4.8.3.7).

Any exceptions to the behavior described in Table 4-7 are described in Section 8.5.1.2, "Invalid Arithmetic Operand Exception (#IA)," and Section 11.5.2.1, "Invalid Operation Exception (#I)."

### Table 4-7. Rules for Handling NaNs

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Result 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNaN and QNaN</td>
<td>X87 FPU — QNaN source operand.</td>
</tr>
<tr>
<td></td>
<td>SSE/SSE2/SSE3/SSE4.1/AVX — First source operand (if this operand is an</td>
</tr>
<tr>
<td></td>
<td>SNaN, it is converted to a QNaN).</td>
</tr>
<tr>
<td>Two SNaNs</td>
<td>X87 FPU — SNaN source operand with the larger significand, converted into</td>
</tr>
<tr>
<td></td>
<td>a QNaN.</td>
</tr>
<tr>
<td></td>
<td>SSE/SSE2/SSE3/SSE4.1/AVX — First source operand converted to a QNaN.</td>
</tr>
<tr>
<td>Two QNaNs</td>
<td>X87 FPU — QNaN source operand with the larger significand.</td>
</tr>
<tr>
<td>SNaN and a floating-point value</td>
<td>SNaN source operand, converted into a QNaN.</td>
</tr>
<tr>
<td>QNaN and a floating-point value</td>
<td>QNaN source operand.</td>
</tr>
<tr>
<td>SNaN (for instructions that take</td>
<td>SNaN source operand, converted into a QNaN.</td>
</tr>
<tr>
<td>only one operand)</td>
<td></td>
</tr>
<tr>
<td>QNaN (for instructions that take</td>
<td>QNaN source operand.</td>
</tr>
<tr>
<td>only one operand)</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**
1. For SSE/SSE2/SSE3/SSE4.1 instructions, the first operand is generally a source operand that becomes the destination operand. For AVX instructions, the first source operand is usually the 2nd operand in a non-destructive source syntax. Within the Result column, the x87 FPU notation also applies to the FISTTP instruction in SSE3; the SSE3 notation applies to the SIMD floating-point instructions.

4.8.3.6 Using SNaNs and QNaNs in Applications

Except for the rules given at the beginning of Section 4.8.3.4, “NaNs,” for encoding SNaNs and QNaNs, software is free to use the bits in the significand of a NaN for any purpose. Both SNaNs and QNaNs can be encoded to carry and store data, such as diagnostic information.
By unmasking the invalid operation exception, the programmer can use signaling NaNs to trap to the exception handler. The generality of this approach and the large number of NaN values that are available provide the sophisticated programmer with a tool that can be applied to a variety of special situations.

For example, a compiler can use signaling NaNs as references to uninitialized (real) array elements. The compiler can preinitialize each array element with a signaling NaN whose significand contains the index (relative position) of the element. Then, if an application program attempts to access an element that it has not initialized, it can use the NaN placed there by the compiler. If the invalid operation exception is unmasked, an interrupt will occur, and the exception handler will be invoked. The exception handler can determine which element has been accessed, since the operand address field of the exception pointer will point to the NaN, and the NaN will contain the index number of the array element.

Quiet NaNs are often used to speed up debugging. In its early testing phase, a program often contains multiple errors. An exception handler can be written to save diagnostic information in memory whenever it is invoked. After storing the diagnostic data, it can supply a quiet NaN as the result of the erroneous instruction, and that NaN can point to its associated diagnostic area in memory. The program will then continue, creating a different NaN for each error. When the program ends, the NaN results can be used to access the diagnostic data saved at the time the errors occurred. Many errors can thus be diagnosed and corrected in one test run.

In embedded applications that use computed results in further computations, an undetected QNaN can invalidate all subsequent results. Such applications should therefore periodically check for QNaNs and provide a recovery mechanism to be used if a QNaN result is detected.

### 4.8.3.7 QNaN Floating-Point Indefinite

For the floating-point data type encodings (single precision, double precision, and double extended precision), one unique encoding (a QNaN) is reserved for representing the special value QNaN floating-point indefinite. The x87 FPU and the Intel SSE/SSE2/SSE3/SSE4.1/AVX extensions return these indefinite values as responses to some masked floating-point exceptions. Table 4-3 shows the encoding used for the QNaN floating-point indefinite.

### 4.8.3.8 Half Precision Floating-Point Operation

Two instructions, VCVTPH2PS and VCVTPS2PH, which provide conversion only between half precision and single precision floating-point values, were introduced with the F16C extensions beginning with the third generation of Intel Core processors based on Ivy Bridge microarchitecture. Starting with the 4th generation Intel Xeon Scalable Processor Family, an Intel AVX-512 instruction set architecture (ISA) for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in the IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16). These additions complement the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products. The SIMD floating-point exception behavior of the VCVTPH2PS and VCVTPS2PH instructions, as well as of the other half precision instructions, are described in Section 14.4.1.

### 4.8.4 Rounding

When performing floating-point operations, the processor produces an infinitely precise floating-point result in the destination format (half precision, single precision, double precision, or double extended precision floating-point) whenever possible. However, because only a subset of the numbers in the real number continuum can be represented in IEEE Standard 754 floating-point formats, it is often the case that an infinitely precise result cannot be encoded exactly in the format of the destination operand.

For example, the following value ($a$) has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the single precision format (which has only a 23-bit fraction):

\[
(a) \ 1.0001 \ 0000 \ 1000 \ 0011 \ 1001 \ 0111_{E2} \ 101
\]

To round this result ($a$), the processor first selects two representable fractions $b$ and $c$ that most closely bracket $a$ in value ($b < a < c$).

\[
(b) \ 1.0001 \ 0000 \ 1000 \ 0011 \ 1001 \ 011_{E2} \ 101
\]

\[
(c) \ 1.0001 \ 0000 \ 1000 \ 0011 \ 1001 \ 100_{E2} \ 101
\]
The processor then sets the result to \( b \) or to \( c \) according to the selected rounding mode. Rounding introduces an error in a result that is less than one unit in the last place (the least significant bit position of the floating-point value) to which the result is rounded.

The IEEE Standard 754 defines four rounding modes (see Table 4-8): round to nearest, round up, round down, and round toward zero. The default rounding mode (for the Intel 64 and IA-32 architectures) is round to nearest. This mode provides the most accurate and statistically unbiased estimate of the true result and is suitable for most applications.

### Table 4-8. Rounding Modes and Encoding of Rounding Control (RC) Field

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>RC Field Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to nearest (even)</td>
<td>00B</td>
<td>Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero). Default</td>
</tr>
<tr>
<td>Round down (toward (-\infty))</td>
<td>01B</td>
<td>Rounded result is closest to but no greater than the infinitely precise result.</td>
</tr>
<tr>
<td>Round up (toward (+\infty))</td>
<td>10B</td>
<td>Rounded result is closest to but no less than the infinitely precise result.</td>
</tr>
<tr>
<td>Round toward zero (Truncate)</td>
<td>11B</td>
<td>Rounded result is closest to but no greater in absolute value than the infinitely precise result.</td>
</tr>
</tbody>
</table>

The round up and round down modes are termed **directed rounding** and can be used to implement interval arithmetic. Interval arithmetic is used to determine upper and lower bounds for the true result of a multistep computation, when the intermediate results of the computation are subject to rounding.

The round toward zero mode (sometimes called the “chop” mode) is commonly used when performing integer arithmetic with the x87 FPU.

The rounded result is called the inexact result. When the processor produces an inexact result, the floating-point precision (inexact) flag (PE) is set (see Section 4.9.1.6, “Inexact-Result (Precision) Exception (#P)”).

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

### 4.8.4.1 Rounding Control (RC) Fields

In the Intel 64 and IA-32 architectures, the rounding mode is controlled by a 2-bit rounding-control (RC) field (Table 4-8 shows the encoding of this field). The RC field is implemented in two different locations:

- X87 FPU control register (bits 10 and 11).
- The MXCSR register (bits 13 and 14).

Although these two RC fields perform the same function, they control rounding for different execution environments within the processor. The RC field in the x87 FPU control register controls rounding for computations performed with the x87 FPU instructions; the RC field in the MXCSR register controls rounding for SIMD floating-point computations performed with the Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions.

### 4.8.4.2 Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions

The following Intel SSE/SSE2 instructions automatically truncate the results of conversions from floating-point values to integers when the result is inexact: CVTTPD2DQ, CVTTPS2DQ, CVTTPD2PI, CVTTPS2PI, CVTTSD2SI, and CVTSS2SI. Here, truncation means the round toward zero mode described in Table 4-8. There are also several Intel AVX2 and AVX-512 instructions which use truncation (VCVT*).
4.9 OVERVIEW OF FLOATING-POINT EXCEPTIONS

The following section provides an overview of floating-point exceptions and their handling in the IA-32 architecture. For information specific to the x87 FPU and to the Intel SSE/SSE2/SSE3/SSE4.1/AVX extensions, refer to the following sections:

- Section 4.9, “Overview of Floating-Point Exceptions.”
- Section 11.5, “Intel® SSE, SSE2, and SSE3 Exceptions.”
- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”

When operating on floating-point operands, the IA-32 architecture recognizes and detects six classes of exception conditions:

- Invalid operation (#I).
- Divide-by-zero (#Z).
- Denormalized operand (#D).
- Numeric overflow (#O).
- Numeric underflow (#U).
- Inexact result (precision) (#P).

The nomenclature of “#” symbol followed by one or two letters (for example, #P) is used in this manual to indicate exception conditions. It is merely a short-hand form and is not related to assembler mnemonics.

NOTE

All of the exceptions listed above except the denormal-operand exception (#D) are defined in IEEE Standard 754.

The invalid-operation, divide-by-zero and denormal-operand exceptions are pre-computation exceptions (that is, they are detected before any arithmetic operation occurs). The numeric-underflow, numeric-overflow and precision exceptions are post-computation exceptions.

Each of the six exception classes has a corresponding flag bit (IE, ZE, OE, UE, DE, or PE) and mask bit (IM, ZM, OM, UM, DM, or PM). When one or more floating-point exception conditions are detected, the processor sets the appropriate flag bits, then takes one of two possible courses of action, depending on the settings of the corresponding mask bits:

- Mask bit set. Handles the exception automatically, producing a predefined (and often times usable) result, while allowing program execution to continue undisturbed.
- Mask bit clear. Invokes a software exception handler to handle the exception.

The masked (default) responses to exceptions have been chosen to deliver a reasonable result for each exception condition and are generally satisfactory for most floating-point applications. By masking or unmasking specific floating-point exceptions, programmers can delegate responsibility for most exceptions to the processor and reserve the most severe exception conditions for software exception handlers.

Because the exception flags are “sticky,” they provide a cumulative record of the exceptions that have occurred since they were last cleared. A programmer can thus mask all exceptions, run a calculation, and then inspect the exception flags to see if any exceptions were detected during the calculation.

In the IA-32 architecture, floating-point exception flag and mask bits are implemented in two different locations:

- X87 FPU status word and control word. The flag bits are located at bits 0 through 5 of the x87 FPU status word and the mask bits are located at bits 0 through 5 of the x87 FPU control word (see Figures 8-4 and 8-6).
- MXCSR register. The flag bits are located at bits 0 through 5 of the MXCSR register and the mask bits are located at bits 7 through 12 of the register (see Figure 10-3).

Although these two sets of flag and mask bits perform the same function, they report on and control exceptions for different execution environments within the processor. The flag and mask bits in the x87 FPU status and control words control exception reporting and masking for computations performed with the x87 FPU instructions; the
companion bits in the MXCSR register control exception reporting and masking for SIMD floating-point computations performed with the Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions.

Note that when exceptions are masked, the processor may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the processor can detect a denormalized operand, perform its masked response to this exception, and then detect numeric underflow.

See Section 4.9.2, “Floating-Point Exception Priority,” for a description of the rules for exception precedence when more than one floating-point exception condition is detected for an instruction.

4.9.1 Floating-Point Exception Conditions

The following sections describe the various conditions that cause a floating-point exception to be generated and the masked response of the processor when these conditions are detected. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C, & 3D, lists the floating-point exceptions that can be signaled for each floating-point instruction.

4.9.1.1 Invalid Operation Exception (#I)

The processor reports an invalid operation exception in response to one or more invalid arithmetic operands. If the invalid operation exception is masked, the processor sets the IE flag and returns an indefinite value or a QNaN. This value overwrites the destination register specified by the instruction. If the invalid operation exception is not masked, the IE flag is set, a software exception handler is invoked, and the operands remain unaltered.

See Section 4.8.3.6, “Using SNaNs and QNaNs in Applications,” for information about the result returned when an exception is caused by an SNaN.

The processor can detect a variety of invalid arithmetic operations that can be coded in a program. These operations generally indicate a programming error, such as dividing by zero or attempting to divide by zero. See the following sections for information regarding the invalid-operation exception when detected while executing x87 FPU or Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.1, “Invalid Operation Exception.”
- SIMD floating-point exceptions; Section 11.5.2.1, “Invalid Operation Exception (#I).”
- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”

4.9.1.2 Denormal Operand Exception (#D)

The processor reports the denormal-operand exception if an arithmetic instruction attempts to operate on a denormal operand (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers"). When the exception is masked, the processor sets the DE flag and proceeds with the instruction. Operating on denormal numbers will produce results at least as good as, and often better than, what can be obtained when denormal numbers are flushed to zero. Programmers can mask this exception so that a computation may proceed, then analyze any loss of accuracy when the final result is delivered.

When a denormal-operand exception is not masked, the DE flag is set, a software exception handler is invoked, and the operands remain unaltered. When denormal operands have reduced significance due to loss of low-order bits, it may be advisable to not operate on them. Precluding denormal operands from computations can be accomplished by an exception handler that responds to unmasked denormal-operand exceptions.

See the following sections for information regarding the denormal-operand exception when detected while executing x87 FPU or Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:

- X87 FPU; Section 8.5.2, “Denormal Operand Exception (#D).”
- SIMD floating-point exceptions; Section 11.5.2.2, “Denormal-Operand Exception (#D).”
- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”
4.9.1.3 Divide-By-Zero Exception (#Z)
The processor reports the floating-point divide-by-zero exception whenever an instruction attempts to divide a finite non-zero operand by 0. The masked response for the divide-by-zero exception is to set the ZE flag and return an infinity signed with the exclusive OR of the sign of the operands. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked, and the operands remain unaltered.

See the following sections for information regarding the divide-by-zero exception when detected while executing x87 FPU or Intel SSE/SSE2/AVX instructions:
- X87 FPU; Section 8.5.3, "Divide-By-Zero Exception (#Z)."
- SIMD floating-point exceptions; Section 11.5.2.3, "Divide-By-Zero Exception (#Z)."
- Section 12.8.4, "IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions."
- Section 14.10, "SIMD Floating-Point Exceptions."

4.9.1.4 Numeric Overflow Exception (#O)
The processor reports a floating-point numeric overflow exception whenever the rounded result of an instruction exceeds the largest allowable finite value that will fit into the destination operand. Table 4-9 shows the threshold range for numeric overflow for each of the floating-point formats; overflow occurs when a rounded result falls at or outside this threshold range.

<table>
<thead>
<tr>
<th>Floating-Point Format</th>
<th>Overflow Thresholds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half Precision</td>
<td>$</td>
</tr>
<tr>
<td>Single Precision</td>
<td>$</td>
</tr>
<tr>
<td>Double Precision</td>
<td>$</td>
</tr>
<tr>
<td>Double Extended Precision</td>
<td>$</td>
</tr>
</tbody>
</table>

When a numeric-overflow exception occurs and the exception is masked, the processor sets the OE flag and returns one of the values shown in Table 4-10, according to the current rounding mode. See Section 4.8.4, "Rounding."

When numeric overflow occurs and the numeric-overflow exception is not masked, the OE flag is set, a software exception handler is invoked, and the source and destination operands either remain unchanged or a biased result is stored in the destination operand (depending whether the overflow exception was generated during an Intel SSE/SSE2/SSE3/SSE4.1/AVX floating-point operation or an x87 FPU operation).

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Sign of True Result</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>To nearest</td>
<td>+</td>
<td>$\pm\infty$</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>$\mp\infty$</td>
</tr>
<tr>
<td>Toward $-\infty$</td>
<td>+</td>
<td>Largest finite positive number</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Toward $+\infty$</td>
<td>+</td>
<td>$\pm\infty$</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Largest finite negative number</td>
</tr>
<tr>
<td>Toward zero</td>
<td>+</td>
<td>Largest finite positive number</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Largest finite negative number</td>
</tr>
</tbody>
</table>

See the following sections for information regarding the numeric overflow exception when detected while executing x87 FPU instructions or while executing Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:
- X87 FPU; Section 8.5.4, "Numeric Overflow Exception (#O)."
• SIMD floating-point exceptions; Section 11.5.2.4, “Numeric Overflow Exception (#O).”
• Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
• Section 14.10, “SIMD Floating-Point Exceptions.”

4.9.1.5 Numeric Underflow Exception (#U)

The processor detects a potential floating-point numeric underflow condition whenever the result of rounding with unbounded exponent (taking into account precision control for x87) is non-zero and tiny; that is, non-zero and less than the smallest possible normalized, finite value that will fit into the destination operand. Table 4-11 shows the threshold range for numeric underflow for each of the floating-point formats (assuming normalized results); underflow occurs when a rounded result falls strictly within the threshold range. The ability to detect and handle underflow is provided to prevent a very small result from propagating through a computation and causing another exception (such as overflow during division) to be generated at a later time. Results which trigger underflow are also potentially less accurate.

<table>
<thead>
<tr>
<th>Floating-Point Format</th>
<th>Underflow Thresholds¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half Precision</td>
<td></td>
</tr>
<tr>
<td>Single Precision</td>
<td></td>
</tr>
<tr>
<td>Double Precision</td>
<td></td>
</tr>
<tr>
<td>Double Extended Precision</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. Where ‘x’ is the result rounded to destination precision with an unbounded exponent range.

How the processor handles an underflow condition, depends on two related conditions:
• Creation of a tiny, non-zero result.
• Creation of an inexact result; that is, a result that cannot be represented exactly in the destination format.

Which of these events causes an underflow exception to be reported and how the processor responds to the exception condition depends on whether the underflow exception is masked:
• Underflow exception masked — The underflow exception is reported (the UE flag is set) only when the result is both tiny and inexact. The processor returns a correctly signed result whose magnitude is less than or equal to the smallest positive normal floating-point number to the destination operand, regardless of inexactness.
• Underflow exception not masked — The underflow exception is reported when the result is non-zero tiny, regardless of inexactness. The processor leaves the source and destination operands unaltered or stores a biased result in the destination operand (depending whether the underflow exception was generated during an Intel SSE/SSE2/SSE3/AVX floating-point operation or an x87 FPU operation) and invokes a software exception handler.

See the following sections for information regarding the numeric underflow exception when detected while executing x87 FPU instructions or while executing Intel SSE/SSE2/SSE3/AVX4.1/AVX instructions:
• X87 FPU; Section 8.5.5, “Numeric Underflow Exception (#U).”
• SIMD floating-point exceptions; Section 11.5.2.5, “Numeric Underflow Exception (#U).”
• Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
• Section 14.10, “SIMD Floating-Point Exceptions.”

4.9.1.6 Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary floating-point form. This exception occurs frequently and indicates that some (normally acceptable) accuracy will be lost due to rounding. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked.
If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or underflow condition has not occurred, the processor sets the PE flag and stores the rounded result in the destination operand. The current rounding mode determines the method used to round the result. See Section 4.8.4, “Rounding.”

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has not occurred, the PE flag is set, the rounded result is stored in the destination operand, and a software exception handler is invoked.

If an inexact result occurs in conjunction with numeric overflow or underflow, one of the following operations is carried out:

- If an inexact result occurs along with masked overflow or underflow, the OE flag or UE flag and the PE flag are set and the result is stored as described for the overflow or underflow exceptions; see Section 4.9.1.4, “Numeric Overflow Exception (#O),” or Section 4.9.1.5, “Numeric Underflow Exception (#U).” If the inexact result exception is unmasked, the processor also invokes a software exception handler.
- If an inexact result occurs along with unmasked overflow or underflow and the destination operand is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or underflow exceptions, and a software exception handler is invoked.

If an unmasked numeric overflow or underflow exception occurs and the destination operand is a memory location (which can happen only for a floating-point store), the inexact-result condition is not reported and the C1 flag is cleared.

See the following sections for information regarding the inexact-result exception when detected while executing x87 FPU or Intel SSE/SSE2/SSE3/SSE4.1/AVX instructions:
- X87 FPU; Section 8.5.6, “Inexact-Result (Precision) Exception (#P).”
- SIMD floating-point exceptions; Section 11.5.2.3, “Divide-By-Zero Exception (#Z).”
- Section 12.8.4, “IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions.”
- Section 14.10, “SIMD Floating-Point Exceptions.”

### 4.9.2 Floating-Point Exception Priority

The processor handles exceptions according to a predetermined precedence. When an instruction generates two or more exception conditions, the exception precedence sometimes results in the higher-priority exception being handled and the lower-priority exceptions being ignored. For example, dividing an SNaN by zero can potentially signal an invalid-operation exception (due to the SNaN operand) and a divide-by-zero exception. Here, if both exceptions are masked, the processor handles the higher-priority exception only (the invalid-operation exception), returning a QNaN to the destination. Alternately, a denormal-operand or inexact-result exception can accompany a numeric underflow or overflow exception with both exceptions being handled.

The precedence for floating-point exceptions is as follows:

1. Invalid-operation exception, subdivided as follows:
   a. Stack underflow (occurs with x87 FPU only).
   b. Stack overflow (occurs with x87 FPU only).
   c. Operand of unsupported format (occurs with x87 FPU only when using the double extended precision floating-point format).
   d. SNaN operand.
2. QNaN operand. Though this is not an exception, the handling of a QNaN operand has precedence over lower-priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a zero-divide exception.
3. Any other invalid-operation exception not mentioned above or a divide-by-zero exception.
4. Denormal-operand exception. If masked, then instruction execution continues, and a lower-priority exception can occur as well.
5. Numeric overflow and underflow exceptions; possibly in conjunction with the inexact-result exception.
6. Inexact-result exception.
4.9.3 Typical Actions of a Floating-Point Exception Handler

After the floating-point exception handler is invoked, the processor handles the exception in the same manner that it handles non-floating-point exceptions. The floating-point exception handler is normally part of the operating system or executive software, and it usually invokes a user-registered floating-point exception handle.

A typical action of the exception handler is to store state information in memory. Other typical exception handler actions include:

• Examining the stored state information to determine the nature of the error.
• Taking actions to correct the condition that caused the error.
• Clearing the exception flags.
• Returning to the interrupted program and resuming normal execution.

In lieu of writing recovery procedures, the exception handler can do the following:

• Increment in software an exception counter for later display or printing.
• Print or display diagnostic information (such as the state information).
• Halt further program execution.
This chapter provides an abridged overview of Intel 64 and IA-32 instructions. Instructions are divided into the following groups:

- **Section 5.1, “General-Purpose Instructions.”**
- **Section 5.2, “x87 FPU Instructions.”**
- **Section 5.3, “x87 FPU AND SIMD State Management Instructions.”**
- **Section 5.4, “MMX Instructions.”**
- **Section 5.5, “Intel® SSE Instructions.”**
- **Section 5.6, “Intel® SSE2 Instructions.”**
- **Section 5.7, “Intel® SSE3 Instructions.”**
- **Section 5.8, “Supplemental Streaming SIMD Extensions 3 (SSSE3) Instructions.”**
- **Section 5.9, “Intel® SSE4 Instructions.”**
- **Section 5.10, “Intel® SSE4.1 Instructions.”**
- **Section 5.11, “Intel® SSE4.2 Instruction Set.”**
- **Section 5.12, “Intel® AES-NI and PCLMULQDQ.”**
- **Section 5.13, “Intel® Advanced Vector Extensions (Intel® AVX).”**
- **Section 5.14, “16-bit Floating-Point Conversion.”**
- **Section 5.15, “Fused-Multiply-ADD (FMA).”**
- **Section 5.16, “Intel® Advanced Vector Extensions 2 (Intel® AVX2).”**
- **Section 5.17, “Intel® Transactional Synchronization Extensions (Intel® TSX).”**
- **Section 5.18, “Intel® SHA Extensions.”**
- **Section 5.19, “Intel® Advanced Vector Extensions 512 (Intel® AVX-512).”**
- **Section 5.20, “System Instructions.”**
- **Section 5.21, “64-Bit Mode Instructions.”**
- **Section 5.22, “Virtual-Machine Extensions.”**
- **Section 5.23, “Safer Mode Extensions.”**
- **Section 5.24, “Intel® Memory Protection Extensions.”**
- **Section 5.25, “Intel® Software Guard Extensions.”**
- **Section 5.26, “Shadow Stack Management Instructions.”**
- **Section 5.27, “Control Transfer Terminating Instructions.”**
- **Section 5.28, “Intel® AMX Instructions.”**
- **Section 5.29, “User Interrupt Instructions.”**
- **Section 5.30, “Enqueue Store Instructions.”**

Table 5-1 lists the groups and IA-32 processors that support each group. More recent instruction set extensions are listed in Table 5-2. Within these groups, most instructions are collected into functional subgroups.
### Table 5-1. Instruction Groups in Intel® 64 and IA-32 Processors

<table>
<thead>
<tr>
<th>Instruction Set Architecture</th>
<th>Intel 64 and IA-32 Processor Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose</td>
<td>All Intel 64 and IA-32 processors.</td>
</tr>
<tr>
<td>SSE Extensions</td>
<td>Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.</td>
</tr>
<tr>
<td>SSE2 Extensions</td>
<td>Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.</td>
</tr>
<tr>
<td>SSE3 Extensions</td>
<td>Pentium 4 supporting HT Technology (built on 90 nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxx, 5xxx, 7xxx Series, Intel Atom processors.</td>
</tr>
<tr>
<td>SSSE3 Extensions</td>
<td>Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors.</td>
</tr>
<tr>
<td>IA-32e mode: 64-bit mode instructions</td>
<td>Intel 64 processors.</td>
</tr>
<tr>
<td>System Instructions</td>
<td>Intel 64 and IA-32 processors.</td>
</tr>
<tr>
<td>VMX Instructions</td>
<td>Intel 64 and IA-32 processors supporting Intel Virtualization Technology.</td>
</tr>
<tr>
<td>SMX Instructions</td>
<td>Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx.</td>
</tr>
</tbody>
</table>

### Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors

<table>
<thead>
<tr>
<th>Instruction Set Architecture</th>
<th>Processor Generation Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE4.1 Extensions</td>
<td>Intel® Xeon® processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel® Core™ 2 Extreme processors QX9000 series, Intel® Core™ 2 Quad processor Q9000 series, Intel® Core™ 2 Duo processors 8000 series and T9000 series, Intel Atom® processor based on Silvermont microarchitecture.</td>
</tr>
<tr>
<td>SSE4.2 Extensions, CRC32, POPCNT</td>
<td>Intel® Core™ i7 965 processor, Intel® Xeon® processors X3400, X3500, X5500, X6500, X7500 series, Intel Atom processor based on Silvermont microarchitecture.</td>
</tr>
<tr>
<td>Intel® AES-NI, PCLMULQDQ</td>
<td>Intel® Xeon® processor E7 series, Intel® Xeon® processors X3600 and X5600, Intel® Core™ i7 980X processor, Intel Atom processor based on Silvermont microarchitecture. Use CPUID to verify presence of Intel AES-NI and PCLMULQDQ across Intel® Core™ processor families.</td>
</tr>
<tr>
<td>Intel® AVX</td>
<td>Intel® Xeon® processor E3 and E5 families, 2nd Generation Intel® Core™ i7, i5, i3 processor 2xxx families.</td>
</tr>
<tr>
<td>F16C</td>
<td>3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Intel® Xeon® processor E5 v2 and E7 v2 families.</td>
</tr>
<tr>
<td>RDRAND</td>
<td>3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom processor based on Silvermont microarchitecture.</td>
</tr>
<tr>
<td>FS/GS base access</td>
<td>3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Intel Xeon processor E5 v2 and E7 v2 families, Intel Atom® processor based on Goldmont microarchitecture.</td>
</tr>
</tbody>
</table>
### Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors (Contd.)

<table>
<thead>
<tr>
<th>Instruction Set Architecture</th>
<th>Processor Generation Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMA, AVX2, BM1, BM2, INVPCID, LZCNT, Intel® TSX</td>
<td>Intel® Xeon* processor E3/E5/E7 v3 product families, 4th Generation Intel® Core™ processor family.</td>
</tr>
<tr>
<td>PREFETCHW</td>
<td>Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family, Intel Atom processor based on Silvermont microarchitecture.</td>
</tr>
<tr>
<td>ADX</td>
<td>Intel Core M processor family, 5th Generation Intel Core processor family.</td>
</tr>
<tr>
<td>RDSEED, CLAC, STAC</td>
<td>Intel Core M processor family, 5th Generation Intel Core processor family, Intel Atom processor based on Goldmont microarchitecture.</td>
</tr>
<tr>
<td>AVX512ER, AVX512PF, PREFETCHWT1</td>
<td>Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series.</td>
</tr>
<tr>
<td>AVX512F, AVX512CD</td>
<td>Intel Xeon Scalable Processor Family, Intel i3-8121U processor.</td>
</tr>
<tr>
<td>CLFLUSHOPT, XSAVEC, XSAVES, Intel® MPX</td>
<td>Intel Xeon Scalable Processor Family, 6th Generation Intel® Core™ processor family, Intel Atom processor based on Goldmont microarchitecture.</td>
</tr>
<tr>
<td>SGX1</td>
<td>6th Generation Intel Core processor family, Intel Atom® processor based on Goldmont Plus microarchitecture.</td>
</tr>
<tr>
<td>AVX512DQ, AVX512BW, AVX512VL</td>
<td>Intel Xeon Scalable Processor Family, Intel Core i3-8121U processor based on Cannon Lake microarchitecture.</td>
</tr>
<tr>
<td>CLWB</td>
<td>Intel Xeon Scalable Processor Family, Intel Atom® processor based on Tremont microarchitecture, 11th Generation Intel Core processor family based on Tiger Lake microarchitecture.</td>
</tr>
<tr>
<td>PKU</td>
<td>Intel Xeon Scalable Processor Family, 10th generation Intel® Core™ processors based on Comet Lake microarchitecture.</td>
</tr>
<tr>
<td>AVX512_IFMA, AVX512_VBMI</td>
<td>Intel Core i3-8121U processor based on Cannon Lake microarchitecture.</td>
</tr>
<tr>
<td>Intel® SHA Extensions</td>
<td>Intel Core i3-8121U processor based on Cannon Lake microarchitecture, Intel Atom processor based on Goldmont microarchitecture, 3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>UMIP</td>
<td>Intel Core i3-8121U processor based on Cannon Lake microarchitecture, Intel Atom processor based on Goldmont Plus microarchitecture.</td>
</tr>
<tr>
<td>PWRITE</td>
<td>Intel Atom processor based on Goldmont Plus microarchitecture, 12th generation Intel® Core™ processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>RDPID</td>
<td>10th Generation Intel® Core™ processor family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>AVX512_4FMAPS, AVX512_4VNNIW</td>
<td>Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series.</td>
</tr>
<tr>
<td>AVX512_VNNI</td>
<td>2nd Generation Intel® Xeon® Scalable Processor Family, 10th Generation Intel Core processor family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>AVX512_VPOPCNTDQ</td>
<td>Intel Xeon Phi Processor 7215, 7285, 7295 Series, 10th Generation Intel Core processor family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>Fast Short REP MOV</td>
<td>10th Generation Intel Core processor family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>GFNI (SSE)</td>
<td>10th Generation Intel Core processor family based on Ice Lake microarchitecture, Intel Atom processor based on Tremont microarchitecture.</td>
</tr>
</tbody>
</table>
### Table 5-2. Instruction Set Extensions Introduction in Intel® 64 and IA-32 Processors (Contd.)

<table>
<thead>
<tr>
<th>Instruction Set Architecture</th>
<th>Processor Generation Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAEs, GFNI (AVX/AVX512), AVX512_VBM2, VPCLMULQDQ, AVX512_BITALG</td>
<td>10th Generation Intel Core processor family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>ENCLV</td>
<td>Future processors.</td>
</tr>
<tr>
<td>Split Lock Detection</td>
<td>10th Generation Intel Core processor family based on Ice Lake microarchitecture, Intel Atom processor based on Tremont microarchitecture.</td>
</tr>
<tr>
<td>CLDEMOTE</td>
<td>Intel Atom processor based on Tremont microarchitecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>Direct stores: MOVDIRI, MOVDIR64B</td>
<td>Intel Atom processor based on Tremont microarchitecture, 11th Generation Intel Core processor family based on Tiger Lake microarchitecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>User wait: TPAUSE, UMONITOR, UMWAIT</td>
<td>Intel Atom processor based on Tremont microarchitecture, 12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>AVX512_BF16</td>
<td>3rd Generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>AVX512_VP2INTERSECT</td>
<td>11th Generation Intel Core processor family based on Tiger Lake microarchitecture.</td>
</tr>
<tr>
<td>Key Locker¹</td>
<td>11th Generation Intel Core processor family based on Tiger Lake microarchitecture, 12th generation Intel Core processor based on Alder Lake performance hybrid architecture.</td>
</tr>
<tr>
<td>Control-flow Enforcement Technology (CET)</td>
<td>11th Generation Intel Core processor family based on Tiger Lake microarchitecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>TME-MK², PCONFIG</td>
<td>3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>WBNOINVD</td>
<td>3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>LBRs (architectural)</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>Intel® Virtualization Technology - Redirect Protection (Intel® VT-rp) and HLAT</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>AVX-VNNI</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture³, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>SERIALIZE</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>Intel® Thread Director and HRESET</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture.</td>
</tr>
<tr>
<td>Fast zero-length REP MOVSB, fast short REP STOSB</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>Fast Short REP CMPSB, fast short REP SCASB</td>
<td>4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>Supervisor Memory Protection Keys (PKS)</td>
<td>12th generation Intel Core processor based on Alder Lake performance hybrid architecture, 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
<tr>
<td>Attestation Services for Intel® SGX</td>
<td>3rd Generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture.</td>
</tr>
<tr>
<td>Enqueue Stores: ENQCMD and ENQCMDS</td>
<td>4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.</td>
</tr>
</tbody>
</table>
The following sections list instructions in each major group and subgroup. Given for each instruction is its mnemonic and descriptive names. When two or more mnemonics are given (for example, CMova/CMovNbe), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMova (Conditional move if above) and CMovNbe (Conditional move if not below or equal) represent the same condition. For detailed information about specific instructions, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

### 5.1 GENERAL-PURPOSE INSTRUCTIONS

The general-purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run on Intel 64 and IA-32 processors. They operate on data contained in memory, in the general-purpose registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP) and in the EFLAGS register. They also operate on address information contained in memory, the general-purpose registers, and the segment registers (CS, DS, SS, ES, FS, and GS).

This group of instructions includes the data transfer, binary integer arithmetic, decimal arithmetic, logic operations, shift and rotate, bit and byte operations, program control, string, flag control, segment register operations, and miscellaneous subgroups. The sections that follow introduce each subgroup.

For more detailed information on general-purpose instructions, see Chapter 7, “Programming With General-Purpose Instructions.”

#### 5.1.1 Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.
INSTRUCTION SET SUMMARY

MOV  Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ Conditional move if not equal/Conditional move if not zero.
CMOVA/CMOVBE Conditional move if above/Conditional move if not below or equal.
CMOVBE/CMOVNA Conditional move if above or equal/Conditional move if not below.
CMOVGB/CMOVNAE Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNB Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG Conditional move if less or equal/Conditional move if not greater.
CMOVC Conditional move if carry.
CMOVNC Conditional move if not carry.
CMOVO Conditional move if overflow.
CMOVNO Conditional move if not overflow.
CMOVS Conditional move if sign (negative).
CMOVNS Conditional move if not sign (non-negative).
CMOVPE/CMOVPO Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO Conditional move if not parity/Conditional move if parity odd.
XCHG Exchange.
BSWAP Byte swap.
XADD Exchange and add.
CMPXCHG Compare and exchange.
CMPXCHG8B Compare and exchange 8 bytes.
PUSH Push onto stack.
POP Pop off of stack.
PUSHA/PUSHAD Push general-purpose registers onto stack.
POPA/POPAD Pop general-purpose registers from stack.
CWD/CDQ Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE Convert byte to word/Convert word to doubleword in EAX register.
MOVSX Move and sign extend.
MOVZX Move and zero extend.

5.1.2 Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.
ADCPX Unsigned integer add with carry.
ADDOX Unsigned integer add with overflow.
ADDC Integer add.
ADDC Add with carry.
SUB Subtract.
SBB Subtract with borrow.
IMUL Signed multiply.
MUL Unsigned multiply.
DIV Signed divide.
DIV          Unsigned divide.
INC          Increment.
DEC          Decrement.
NEG          Negate.
CMP          Compare.

5.1.3   Decimal Arithmetic Instructions
The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.
DAA          Decimal adjust after addition.
DAS          Decimal adjust after subtraction.
AAA          ASCII adjust after addition.
AAS          ASCII adjust after subtraction.
AAM          ASCII adjust after multiplication.
AAD          ASCII adjust before division.

5.1.4   Logical Instructions
The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.
AND          Perform bitwise logical AND.
OR           Perform bitwise logical OR.
XOR          Perform bitwise logical exclusive OR.
NOT          Perform bitwise logical NOT.

5.1.5   Shift and Rotate Instructions
The shift and rotate instructions shift and rotate the bits in word and doubleword operands.
SAR          Shift arithmetic right.
SHR          Shift logical right.
SAL/SHL      Shift arithmetic left/Shift logical left.
SHRD         Shift right double.
SHLD         Shift left double.
ROR          Rotate right.
ROL          Rotate left.
RCR          Rotate through carry right.
RCL          Rotate through carry left.

5.1.6   Bit and Byte Instructions
Bit instructions test and modify individual bits in word and doubleword operands. Byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.
BT           Bit test.
BTS          Bit test and set.
BTR          Bit test and reset.
BTC          Bit test and complement.
BSF          Bit scan forward.
### INSTRUCTION SET SUMMARY

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR</td>
<td>Bit scan reverse.</td>
</tr>
<tr>
<td>SETE/SETZ</td>
<td>Set byte if equal/Set byte if zero.</td>
</tr>
<tr>
<td>SETNE/SETNZ</td>
<td>Set byte if not equal/Set byte if not zero.</td>
</tr>
<tr>
<td>SETA/SETNBE</td>
<td>Set byte if above/Set byte if not below or equal.</td>
</tr>
<tr>
<td>SETAE/SETNB/SETNC</td>
<td>Set byte if above or equal/Set byte if not below/Set byte if not carry.</td>
</tr>
<tr>
<td>SETB/SETNAE/SETC</td>
<td>Set byte if below/Set byte if not above or equal/Set byte if carry.</td>
</tr>
<tr>
<td>SETBE/SETNLE</td>
<td>Set byte if below or equal/Set byte if not above.</td>
</tr>
<tr>
<td>SETGE/SETNL</td>
<td>Set byte if greater or equal/Set byte if not less.</td>
</tr>
<tr>
<td>SETLE/SETNG</td>
<td>Set byte if less or equal/Set byte if not greater.</td>
</tr>
<tr>
<td>SETS</td>
<td>Set byte if sign (negative).</td>
</tr>
<tr>
<td>SETNS</td>
<td>Set byte if not sign (non-negative).</td>
</tr>
<tr>
<td>SETO</td>
<td>Set byte if overflow.</td>
</tr>
<tr>
<td>SETNO</td>
<td>Set byte if not overflow.</td>
</tr>
<tr>
<td>SETPE/SETP</td>
<td>Set byte if parity even/Set byte if parity.</td>
</tr>
<tr>
<td>SETPO/SETNP</td>
<td>Set byte if parity odd/Set byte if not parity.</td>
</tr>
<tr>
<td>TEST</td>
<td>Logical compare.</td>
</tr>
<tr>
<td>CRC32(^1)</td>
<td>Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.</td>
</tr>
<tr>
<td>POPCNT(^2)</td>
<td>Calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).</td>
</tr>
</tbody>
</table>

### 5.1.7 Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>Jump.</td>
</tr>
<tr>
<td>JE/JZ</td>
<td>Jump if equal/Jump if zero.</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>Jump if not equal/Jump if not zero.</td>
</tr>
<tr>
<td>JA/JNBE</td>
<td>Jump if above/Jump if not below or equal.</td>
</tr>
<tr>
<td>JAE/JNB</td>
<td>Jump if above or equal/Jump if not below.</td>
</tr>
<tr>
<td>JB/JNAE</td>
<td>Jump if below/Jump if not above or equal.</td>
</tr>
<tr>
<td>JBE/JNA</td>
<td>Jump if below or equal/Jump if not above.</td>
</tr>
<tr>
<td>JG/JNLE</td>
<td>Jump if greater/Jump if not less or equal.</td>
</tr>
<tr>
<td>JGE/JNL</td>
<td>Jump if greater or equal/Jump if not less.</td>
</tr>
<tr>
<td>JL/JNGE</td>
<td>Jump if less/Jump if not greater or equal.</td>
</tr>
<tr>
<td>JLE/JNG</td>
<td>Jump if less or equal/Jump if not greater.</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry.</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry.</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if overflow.</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow.</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign (negative).</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not sign (non-negative).</td>
</tr>
</tbody>
</table>

---

1. Processor support of CRC32 is enumerated by CPUID.01:ECX[SSE4.2] = 1
2. Processor support of POPCNT is enumerated by CPUID.01:ECX[POPCNT] = 1
INSTRUCTION SET SUMMARY

JPO/JNP Jump if parity odd/Jump if not parity.
JPE/JP Jump if parity even/Jump if parity.
JCXZ/JECXZ Jump register CX zero/Jump register ECX zero.
LOOP Loop with ECX counter.
LOOPZ/LOOPE Loop with ECX and zero/Loop with ECX and equal.
LOOPNZ/LOOPNE Loop with ECX and not zero/Loop with ECX and not equal.
CALL Call procedure.
RET Return.
IRET Return from interrupt.
INT Software interrupt.
INTO Interrupt on overflow.
BOUND Detect value out of range.
ENTER High-level procedure entry.
LEAVE High-level procedure exit.

5.1.8 String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

MOVS/MOVSB Move string/Move byte string.
MOVS/MOVSW Move string/Move word string.
MOVS/MOVSD Move string/Move doubleword string.
CMPS/CMPSB Compare string/Compare byte string.
CMPS/CMPSW Compare string/Compare word string.
CMPS/CMPSD Compare string/Compare doubleword string.
SCAS/SCASB Scan string/Scan byte string.
SCAS/SCASW Scan string/Scan word string.
SCAS/SCASD Scan string/Scan doubleword string.
LODS/LODSB Load string/Load byte string.
LODS/LODSW Load string/Load word string.
LODS/LODSD Load string/Load doubleword string.
STOS/STOSB Store string/Store byte string.
STOS/STOSW Store string/Store word string.
STOS/STOSD Store string/Store doubleword string.
REP Repeat while ECX not zero.
REPE/REPZ Repeat while equal/Repeat while zero.
REPNE/REPNZ Repeat while not equal/Repeat while not zero.

5.1.9 I/O Instructions

These instructions move data between the processor’s I/O ports and a register or memory.

IN Read from a port.
OUT Write to a port.
INS/INSB Input string from port/Input byte string from port.
INS/INSW Input string from port/Input word string from port.
INS/INSD Input string from port/Input doubleword string from port.
OUTS/OUTSB Output string to port/Output byte string to port.
OUTS/OUTSW Output string to port/Output word string to port.
OUTS/OUTSD  Output string to port/Output doubleword string to port.

5.1.10  Enter and Leave Instructions
These instructions provide machine-language support for procedure calls in block-structured languages.
ENTER  High-level procedure entry.
LEAVE  High-level procedure exit.

5.1.11  Flag Control (EFLAG) Instructions
The flag control instructions operate on the flags in the EFLAGS register.
STC  Set carry flag.
CLC  Clear the carry flag.
CMC  Complement the carry flag.
CLD  Clear the direction flag.
STD  Set direction flag.
LAHF  Load flags into AH register.
SAHF  Store AH register into flags.
PUSHF/PUSHFD  Push EFLAGS onto stack.
POPF/POPFD  Pop EFLAGS from stack.
STI  Set interrupt flag.
CLI  Clear the interrupt flag.

5.1.12  Segment Register Instructions
The segment register instructions allow far pointers (segment addresses) to be loaded into the segment registers.
LDS  Load far pointer using DS.
LES  Load far pointer using ES.
LFS  Load far pointer using FS.
LGS  Load far pointer using GS.
LSS  Load far pointer using SS.

5.1.13  Miscellaneous Instructions
The miscellaneous instructions provide such functions as loading an effective address, executing a “no-operation,” and retrieving processor identification information.
LEA  Load effective address.
NOP  No operation.
UD  Undefined instruction.
XLAT/XLATB  Table lookup translation.
CPUID  Processor identification.
MOVBE\(^1\)  Move data after swapping data bytes.
PREFETCHW  Prefetch data into cache in anticipation of write.
PREFETCHWT1  Prefetch hint T1 with intent to write.

---

1. Processor support of MOVBE is enumerated by CPUID.01:ECX.MOVBE[bit 22] = 1.
INSTRUCTION SET SUMMARY

CLFLUSH  Flushes and invalidates a memory operand and its associated cache line from all levels of the processor’s cache hierarchy.

CLFLUSHOPT  Flushes and invalidates a memory operand and its associated cache line from all levels of the processor’s cache hierarchy with optimized memory system throughput.

5.1.14 User Mode Extended State Save/Restore Instructions

XSAVE  Save processor extended states to memory.
XSAVEC  Save processor extended states with compaction to memory.
XSAVEOPT  Save processor extended states to memory, optimized.
XRSTOR  Restore processor extended states from memory.
XGETBV  Reads the state of an extended control register.

5.1.15 Random Number Generator Instructions

RDRAND  Retrieves a random number generated from hardware.
RDSEED  Retrieves a random number generated from hardware.

5.1.16 BMI1 and BMI2 Instructions

ANDN  Bitwise AND of first source with inverted second source operands.
BEXTR  Contiguous bitwise extract.
BLSI  Extract lowest set bit.
BLSMSK  Set all lower bits below first set bit to 1.
BLSR  Reset lowest set bit.
BZHI  Zero high bits starting from specified bit position.
LZCNT  Count the number of leading zero bits.
MULX  Unsigned multiply without affecting arithmetic flags.
PDEP  Parallel deposit of bits using a mask.
PEXT  Parallel extraction of bits using a mask.
RORX  Rotate right without affecting arithmetic flags.
SARX  Shift arithmetic right.
SHLX  Shift logic left.
SHRX  Shift logic right.
TZCNT  Count the number of trailing zero bits.

5.1.16.1 Detection of VEX-Encoded GPR Instructions, LZCNT, TZCNT, and PREFETCHW

VEX-encoded general-purpose instructions do not operate on any vector registers.

There are separate feature flags for the following subsets of instructions that operate on general purpose registers, and the detection requirements for hardware support are:

CPUID.(EAX=07H, ECX=0H):EBX.BMI1[bit 3]: if 1 indicates the processor supports the first group of advanced bit manipulation extensions (ANDN, BEXTR, BLSI, BLSMSK, BLSR, TZCNT);

CPUID.(EAX=07H, ECX=0H):EBX.BMI2[bit 8]: if 1 indicates the processor supports the second group of advanced bit manipulation extensions (BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX);

CPUID.EAX=80000001H:ECX.LZCNT[bit 5]: if 1 indicates the processor supports the LZCNT instruction.

CPUID.EAX=80000001H:ECX.PREFETCHW[bit 8]: if 1 indicates the processor supports the PREFETCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFETCHWT1[bit 0]: if 1 indicates the processor supports the PREFETCHWT1 instruction.
5.2 X87 FPU INSTRUCTIONS

The x87 FPU instructions are executed by the processor’s x87 FPU. These instructions operate on floating-point, integer, and binary-coded decimal (BCD) operands. For more detail on x87 FPU instructions, see Chapter 8, “Programming with the x87 FPU.”

These instructions are divided into the following subgroups: data transfer, load constants, and FPU control instructions. The sections that follow introduce each subgroup.

5.2.1 X87 FPU Data Transfer Instructions

The data transfer instructions move floating-point, integer, and BCD values between memory and the x87 FPU registers. They also perform conditional move operations on floating-point operands.

- **FLD** Load floating-point value.
- **FST** Store floating-point value.
- **FSTP** Store floating-point value and pop.
- **FILD** Load integer.
- **FIST** Store integer.
- **FISTP** Store integer and pop.
- **FBLD** Load BCD.
- **FBSTP** Store BCD and pop.
- **FXCH** Exchange registers.
- **FCMOVE** Floating-point conditional move if equal.
- **FCMOVNE** Floating-point conditional move if not equal.
- **FCMOVB** Floating-point conditional move if below.
- **FCMOVBE** Floating-point conditional move if below or equal.
- **FCMOVNB** Floating-point conditional move if not below.
- **FCMOVNBE** Floating-point conditional move if not below or equal.
- **FCMOVU** Floating-point conditional move if unordered.
- **FCMOVNU** Floating-point conditional move if not unordered.

5.2.2 X87 FPU Basic Arithmetic Instructions

The basic arithmetic instructions perform basic arithmetic operations on floating-point and integer operands.

- **FADD** Add floating-point.
- **FADDP** Add floating-point and pop.
- **FIADD** Add integer.
- **FSUB** Subtract floating-point.
- **FSUBP** Subtract floating-point and pop.
- **FISUB** Subtract integer.
- **FSUBR** Subtract floating-point reverse.
- **FSUBRP** Subtract floating-point reverse and pop.
- **FISUBR** Subtract integer reverse.
- **FMUL** Multiply floating-point.
- **FMULP** Multiply floating-point and pop.
- **FIMUL** Multiply integer.
- **FDIV** Divide floating-point.

---

1. SSE3 provides an instruction FISTTP for integer conversion.
FDIVP  Divide floating-point and pop.
FIDIV  Divide integer.
FDIVR  Divide floating-point reverse.
FDIVRP Divide floating-point reverse and pop.
FIDIVR Divide integer reverse.
FPREM  Partial remainder.
FPREM1 IEEE partial remainder.
FABS  Absolute value.
FCHS  Change sign.
FRNDINT Round to integer.
FSCALE Scale by power of two.
FSQRT Square root.
FXTRACT Extract exponent and significand.

5.2.3  X87 FPU Comparison Instructions

The compare instructions examine or compare floating-point or integer operands.
FCOM   Compare floating-point.
FCOMP  Compare floating-point and pop.
FCOMPP Compare floating-point and pop twice.
FUCOM  Unordered compare floating-point.
FUCOMP Unordered compare floating-point and pop.
FUCOMPP Unordered compare floating-point and pop twice.
FICOM  Compare integer.
FICOMP Compare integer and pop.
FCOMI  Compare floating-point and set EFLAGS.
FUCOMI Unordered compare floating-point and set EFLAGS.
FCOMIP Compare floating-point, set EFLAGS, and pop.
FUCOMIP Unordered compare floating-point, set EFLAGS, and pop.
FTST   Test floating-point (compare with 0.0).
FXAM   Examine floating-point.

5.2.4  X87 FPU Transcendental Instructions

The transcendental instructions perform basic trigonometric and logarithmic operations on floating-point operands.
FSIN   Sine.
FCOS   Cosine.
FSINCOS Sine and cosine.
FPTAN  Partial tangent.
FPATAN Partial arctangent.
F2XM1  \(2^x - 1\).
FYL2X  \(y \times \log_2 x\).
FYL2XP1 \(y \times \log_2 (x+1)\).
5.2.5 X87 FPU Load Constants Instructions

The load constants instructions load common constants, such as $\pi$, into the x87 floating-point registers.

- FLD1: Load +1.0.
- FLNZ: Load +0.0.
- FLDPI: Load $\pi$.
- FDL2E: Load $\log_2 e$.
- FDLN2: Load $\log_e 2$.
- FDL2T: Load $\log_2 10$.
- FDLG2: Load $\log_{10} 2$.

5.2.6 X87 FPU Control Instructions

The x87 FPU control instructions operate on the x87 FPU register stack and save and restore the x87 FPU state.

- FINCSTP: Increment FPU register stack pointer.
- FDECSTP: Decrement FPU register stack pointer.
- FFREE: Free floating-point register.
- FINIT: Initialize FPU after checking error conditions.
- FNINIT: Initialize FPU without checking error conditions.
- FCLEX: Clear floating-point exception flags after checking for error conditions.
- FNCLEX: Clear floating-point exception flags without checking for error conditions.
- FSTCW: Store FPU control word after checking error conditions.
- FNSTCW: Store FPU control word without checking error conditions.
- FLDCW: Load FPU control word.
- FSTENV: Store FPU environment after checking error conditions.
- FNSTENV: Store FPU environment without checking error conditions.
- FLDENV: Load FPU environment.
- FSAVE: Save FPU state after checking error conditions.
- FNSAVE: Save FPU state without checking error conditions.
- FRSTOR: Restore FPU state.
- FSTSW: Store FPU status word after checking error conditions.
- FNSTSW: Store FPU status word without checking error conditions.
- WAIT/FWAIT: Wait for FPU.
- FNOP: FPU no operation.

5.3 X87 FPU AND SIMD STATE MANAGEMENT INSTRUCTIONS

Two state management instructions were introduced into the IA-32 architecture with the Pentium II processor family:

- FXSAVE: Save x87 FPU and SIMD state.
- FXRSTOR: Restore x87 FPU and SIMD state.

Initially, these instructions operated only on the x87 FPU (and MMX) registers to perform a fast save and restore, respectively, of the x87 FPU and MMX state. With the introduction of SSE extensions in the Pentium III processor family, these instructions were expanded to also save and restore the state of the XMM and MXCSR registers. Intel 64 architecture also supports these instructions.

See Section 10.5, "FXSAVE and FXRSTOR Instructions," for more detail.
5.4 MMX INSTRUCTIONS

Four extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, and SSE3 extensions. For a discussion that puts SIMD instructions in their historical context, see Section 2.2.7, “SIMD Instructions.”

MMX instructions operate on packed byte, word, doubleword, or quadword integer operands contained in memory, in MMX registers, and/or in general-purpose registers. For more detail on these instructions, see Chapter 9, “Programming with Intel® MMX™ Technology.”

MMX instructions can only be executed on Intel 64 and IA-32 processors that support the MMX technology. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

MMX instructions are divided into the following subgroups: data transfer, conversion, packed arithmetic, comparison, logical, shift and rotate, and state management instructions. The sections that follow introduce each subgroup.

5.4.1 MMX Data Transfer Instructions

The data transfer instructions move doubleword and quadword operands between MMX registers and memory.

MOVD Move doubleword.
MOVQ Move quadword.

5.4.2 MMX Conversion Instructions

The conversion instructions pack and unpack bytes, words, and doublewords

PACKSSWB Pack words into bytes with signed saturation.
PACKSSDW Pack doublewords into words with signed saturation.
PACKUSWB Pack words into bytes with unsigned saturation.
PUNPCKHBW Unpack high-order bytes.
PUNPCKHWD Unpack high-order words.
PUNPCKHDQ Unpack high-order doublewords.
PUNPCKLBW Unpack low-order bytes.
PUNPCKLWD Unpack low-order words.
PUNPCKLDQ Unpack low-order doublewords.

5.4.3 MMX Packed Arithmetic Instructions

The packed arithmetic instructions perform packed integer arithmetic on packed byte, word, and doubleword integers.

PADDB Add packed byte integers.
PADDW Add packed word integers.
PADD Add packed doubleword integers.
PADDSB Add packed signed byte integers with signed saturation.
PADDSW Add packed signed word integers with signed saturation.
PADDUSB Add packed unsigned byte integers with unsigned saturation.
PADDUSW Add packed unsigned word integers with unsigned saturation.
PSUBB Subtract packed byte integers.
PSUBW Subtract packed word integers.
INSTRUCTION SET SUMMARY

PSUBD             Subtract packed doubleword integers.
PSUBSB            Subtract packed signed byte integers with signed saturation.
PSUBSW            Subtract packed signed word integers with signed saturation.
PSUBUSB           Subtract packed unsigned byte integers with unsigned saturation.
PSUBUSW           Subtract packed unsigned word integers with unsigned saturation.
PMULHW            Multiply packed signed word integers and store high result.
PMULLW            Multiply packed signed word integers and store low result.
PMADDWD           Multiply and add packed word integers.

5.4.4 MMX Comparison Instructions
The compare instructions compare packed bytes, words, or doublewords.
PCMPEQB           Compare packed bytes for equal.
PCMPEQW           Compare packed words for equal.
PCMPEQD           Compare packed doublewords for equal.
PCMMPGTB          Compare packed signed byte integers for greater than.
PCMMPGTW          Compare packed signed word integers for greater than.
PCMMPGTD          Compare packed signed doubleword integers for greater than.

5.4.5 MMX Logical Instructions
The logical instructions perform AND, AND NOT, OR, and XOR operations on quadword operands.
PAND               Bitwise logical AND.
PANDN              Bitwise logical AND NOT.
POR                Bitwise logical OR.
PXOR               Bitwise logical exclusive OR.

5.4.6 MMX Shift and Rotate Instructions
The shift and rotate instructions shift and rotate packed bytes, words, or doublewords, or quadwords in 64-bit operands.
PSLLW             Shift packed words left logical.
PSLLD             Shift packed doublewords left logical.
PSLLQ             Shift packed quadword left logical.
PSRLW             Shift packed words right logical.
PSRLD             Shift packed doublewords right logical.
PSRLQ             Shift packed quadword right logical.
PSRAW             Shift packed words right arithmetic.
PSRAD             Shift packed doublewords right arithmetic.

5.4.7 MMX State Management Instructions
The EMMS instruction clears the MMX state from the MMX registers.
EMMS              Empty MMX state.
5.5 INTEL® SSE INSTRUCTIONS

Intel SSE instructions represent an extension of the SIMD execution model introduced with the MMX technology. For more detail on these instructions, see Chapter 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE).”

Intel SSE instructions can only be executed on Intel 64 and IA-32 processors that support Intel SSE extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

Intel SSE instructions are divided into four subgroups (note that the first subgroup has subordinate subgroups of its own):

- SIMD single precision floating-point instructions that operate on the XMM registers.
- MXCSR state management instructions.
- 64-bit SIMD integer instructions that operate on the MMX registers.
- Cacheability control, prefetch, and instruction ordering instructions.

The following sections provide an overview of these groups.

5.5.1 Intel® SSE SIMD Single Precision Floating-Point Instructions

These instructions operate on packed and scalar single precision floating-point values located in XMM registers and/or memory. This subgroup is further divided into the following subordinate subgroups: data transfer, packed arithmetic, comparison, logical, shuffle and unpack, and conversion instructions.

5.5.1.1 Intel® SSE Data Transfer Instructions

Intel SSE data transfer instructions move packed and scalar single precision floating-point operands between XMM registers and between XMM registers and memory.

- **MOVAPS** Move four aligned packed single precision floating-point values between XMM registers or between an XMM register and memory.
- **MOVUPS** Move four unaligned packed single precision floating-point values between XMM registers or between an XMM register and memory.
- **MOVHPS** Move two packed single precision floating-point values to and from the high quadword of an XMM register and memory.
- **MOVHLPS** Move two packed single precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register.
- **MOVLPS** Move two packed single precision floating-point values to and from the low quadword of an XMM register and memory.
- **MOVHLPS** Move two packed single precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register.
- **MOVMSKPS** Extract sign mask from four packed single precision floating-point values.
- **MOVSS** Move scalar single precision floating-point value between XMM registers or between an XMM register and memory.

5.5.1.2 Intel® SSE Packed Arithmetic Instructions

Intel SSE packed arithmetic instructions perform packed and scalar arithmetic operations on packed and scalar single precision floating-point operands.

- **ADDRS** Add packed single precision floating-point values.
- **ADDS** Add scalar single precision floating-point values.
- **SUBPS** Subtract packed single precision floating-point values.
- **SUBSS** Subtract scalar single precision floating-point values.
INSTRUCTION SET SUMMARY

MULPS Multiply packed single precision floating-point values.
MULSS Multiply scalar single precision floating-point values.
DIVPS Divide packed single precision floating-point values.
DIVSS Divide scalar single precision floating-point values.
RCPPS Compute reciprocals of packed single precision floating-point values.
RCPSS Compute reciprocal of scalar single precision floating-point values.
SQRTPS Compute square roots of packed single precision floating-point values.
SQRTPS Compute square root of scalar single precision floating-point values.
RSQRTPS Compute reciprocals of square roots of packed single precision floating-point values.
RSQRTSS Compute reciprocal of square root of scalar single precision floating-point values.
MAXPS Return maximum packed single precision floating-point values.
MAXSS Return maximum scalar single precision floating-point values.
MINPS Return minimum packed single precision floating-point values.
MINSS Return minimum scalar single precision floating-point values.

5.5.1.3 Intel® SSE Comparison Instructions

Intel SSE compare instructions compare packed and scalar single precision floating-point operands.
CMPPS Compare packed single precision floating-point values.
CMPSS Compare scalar single precision floating-point values.
COMISS Perform ordered comparison of scalar single precision floating-point values and set flags in EFLAGS register.
UCOMISS Perform unordered comparison of scalar single precision floating-point values and set flags in EFLAGS register.

5.5.1.4 Intel® SSE Logical Instructions

Intel SSE logical instructions perform bitwise AND, AND NOT, OR, and XOR operations on packed single precision floating-point operands.
ANDPS Perform bitwise logical AND of packed single precision floating-point values.
ANDNPS Perform bitwise logical AND NOT of packed single precision floating-point values.
ORPS Perform bitwise logical OR of packed single precision floating-point values.
XORPS Perform bitwise logical XOR of packed single precision floating-point values.

5.5.1.5 Intel® SSE Shuffle and Unpack Instructions

Intel SSE shuffle and unpack instructions shuffle or interleave single precision floating-point values in packed single precision floating-point operands.
SHUFPS Shuffles values in packed single precision floating-point operands.
UNPCKHPS Unpacks and interleave the two high-order values from two single precision floating-point operands.
UNPCKLPS Unpacks and interleave the two low-order values from two single precision floating-point operands.

5.5.1.6 Intel® SSE Conversion Instructions

Intel SSE conversion instructions convert packed and individual doubleword integers into packed and scalar single precision floating-point values and vice versa.
CVTPSI2PS Convert packed doubleword integers to packed single precision floating-point values.
CVTSI2SS Convert doubleword integer to scalar single precision floating-point value.
5.5.2 Intel® SSE MXCSR State Management Instructions

MXCSR state management instructions allow saving and restoring the state of the MXCSR control and status register.

LDMXCSR Load MXCSR register.
STMXCSR Save MXCSR register state.

5.5.3 Intel® SSE 64-Bit SIMD Integer Instructions

These Intel SSE 64-bit SIMD integer instructions perform additional operations on packed bytes, words, or doublewords contained in MMX registers. They represent enhancements to the MMX instruction set described in Section 5.4, “MMX Instructions.”

PAVGB Compute average of packed unsigned byte integers.
PAVGW Compute average of packed unsigned word integers.
PEXTRW Extract word.
PINSRW Insert word.
PMAXUB Maximum of packed unsigned byte integers.
PMAXSW Maximum of packed signed word integers.
PMINUB Minimum of packed unsigned byte integers.
PMINSW Minimum of packed signed word integers.
PMOVMSKB Move byte mask.
PMULHUW Multiply packed unsigned integers and store high result.
PSADBW Compute sum of absolute differences.
PSHUFW Shuffle packed integer word in MMX register.

5.5.4 Intel® SSE Cacheability Control, Prefetch, and Instruction Ordering Instructions

The cacheability control instructions provide control over the caching of non-temporal data when storing data from the MMX and XMM registers to memory. The PREFETCHh allows data to be prefetched to a selected cache level. The SFENCE instruction controls instruction ordering on store operations.

MASKMOVQ Non-temporal store of selected bytes from an MMX register into memory.
MOVNTQ Non-temporal store of quadword from an MMX register into memory.
MOVNTPS Non-temporal store of four packed single precision floating-point values from an XMM register into memory.
PREFETCHh Load 32 or more of bytes from memory to a selected level of the processor’s cache hierarchy.
SFENCE Serializes store operations.

5.6 INTEL® SSE2 INSTRUCTIONS

Intel SSE2 extensions represent an extension of the SIMD execution model introduced with MMX technology and the Intel SSE extensions. Intel SSE2 instructions operate on packed double precision floating-point operands and
on packed byte, word, doubleword, and quadword operands located in the XMM registers. For more detail on these instructions, see Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2).”

Intel SSE2 instructions can only be executed on Intel 64 and IA-32 processors that support the Intel SSE2 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

These instructions are divided into four subgroups (note that the first subgroup is further divided into subordinate subgroups):

- Packed and scalar double precision floating-point instructions.
- Packed single precision floating-point conversion instructions.
- 128-bit SIMD integer instructions.
- Cacheability-control and instruction ordering instructions.

The following sections give an overview of each subgroup.

### 5.6.1 Intel® SSE2 Packed and Scalar Double Precision Floating-Point Instructions

Intel SSE2 packed and scalar double precision floating-point instructions are divided into the following subordinate subgroups: data movement, arithmetic, comparison, conversion, logical, and shuffle operations on double precision floating-point operands. These are introduced in the sections that follow.

#### 5.6.1.1 Intel® SSE2 Data Movement Instructions

Intel SSE2 data movement instructions move double precision floating-point data between XMM registers and between XMM registers and memory.

- **MOVAPD** Move two aligned packed double precision floating-point values between XMM registers or between an XMM register and memory.
- **MOVUPD** Move two unaligned packed double precision floating-point values between XMM registers or between an XMM register and memory.
- **MOVHPD** Move high packed double precision floating-point value to and from the high quadword of an XMM register and memory.
- **MOVLPD** Move low packed single precision floating-point value to and from the low quadword of an XMM register and memory.
- **MOVMSKPD** Extract sign mask from two packed double precision floating-point values.
- **MOVSD** Move scalar double precision floating-point value between XMM registers or between an XMM register and memory.

#### 5.6.1.2 Intel® SSE2 Packed Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double precision floating-point operands.

- **ADDPD** Add packed double precision floating-point values.
- **ADDSD** Add scalar double precision floating-point values.
- **SUBPD** Subtract packed double precision floating-point values.
- **SUBSD** Subtract scalar double precision floating-point values.
- **MULPD** Multiply packed double precision floating-point values.
- **MULSD** Multiply scalar double precision floating-point values.
- **DIVPD** Divide packed double precision floating-point values.
- **DIVSD** Divide scalar double precision floating-point values.
- **SQRTPD** Compute packed square roots of packed double precision floating-point values.
- **SQRTSID** Compute scalar square root of scalar double precision floating-point values.
MAXPD  Return maximum packed double precision floating-point values.
MAXSD  Return maximum scalar double precision floating-point values.
MINPD  Return minimum packed double precision floating-point values.
MINSD  Return minimum scalar double precision floating-point values.

5.6.1.3  Intel® SSE2 Logical Instructions
Intel SSE2 logical instructions perform AND, AND NOT, OR, and XOR operations on packed double precision floating-point values.
ANDPD  Perform bitwise logical AND of packed double precision floating-point values.
ANDNPD Perform bitwise logical AND NOT of packed double precision floating-point values.
ORPD   Perform bitwise logical OR of packed double precision floating-point values.
XORPD  Perform bitwise logical XOR of packed double precision floating-point values.

5.6.1.4  Intel® SSE2 Compare Instructions
Intel SSE2 compare instructions compare packed and scalar double precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.
CMPPD  Compare packed double precision floating-point values.
CMPSD  Compare scalar double precision floating-point values.
COMISD Perform ordered comparison of scalar double precision floating-point values and set flags in EFLAGS register.
UCOMISD Perform unordered comparison of scalar double precision floating-point values and set flags in EFLAGS register.

5.6.1.5  Intel® SSE2 Shuffle and Unpack Instructions
Intel SSE2 shuffle and unpack instructions shuffle or interleave double precision floating-point values in packed double precision floating-point operands.
SHUFPD Shuffles values in packed double precision floating-point operands.
UNPCKHPD Unpacks and interleaves the high values from two packed double precision floating-point operands.
UNPCKLPD Unpacks and interleaves the low values from two packed double precision floating-point operands.

5.6.1.6  Intel® SSE2 Conversion Instructions
Intel SSE2 conversion instructions convert packed and individual doubleword integers into packed and scalar double precision floating-point values and vice versa. They also convert between packed and scalar single precision and double precision floating-point values.
CVTPD2PI Convert packed double precision floating-point values to packed doubleword integers.
CVTTPD2PI Convert with truncation packed double precision floating-point values to packed doubleword integers.
CVTPI2PD Convert packed doubleword integers to packed double precision floating-point values.
CVTPD2DQ Convert packed double precision floating-point values to packed doubleword integers.
CVTTPD2DQ Convert with truncation packed double precision floating-point values to packed doubleword integers.
CVTDQ2PD Convert packed doubleword integers to packed double precision floating-point values.
CVTPS2PD Convert packed single precision floating-point values to packed double precision floating-point values.
INSTRUCTION SET SUMMARY

CVTPD2PS Convert packed double precision floating-point values to packed single precision floating-point values.
CVTSS2SD Convert scalar single precision floating-point values to scalar double precision floating-point values.
CVTSD2SS Convert scalar double precision floating-point values to scalar single precision floating-point values.
CVTSD2SI Convert scalar double precision floating-point values to a doubleword integer.
CVTTS2SI Convert with truncation scalar double precision floating-point values to scalar doubleword integers.
CVTSI2SD Convert doubleword integer to scalar double precision floating-point value.

5.6.2 Intel® SSE2 Packed Single Precision Floating-Point Instructions

Intel SSE2 packed single precision floating-point instructions perform conversion operations on single precision floating-point and integer operands. These instructions represent enhancements to the Intel SSE single precision floating-point instructions.
CVTDQ2PS Convert packed doubleword integers to packed single precision floating-point values.
CVTPS2DQ Convert packed single precision floating-point values to packed doubleword integers.
CVTTPS2DQ Convert with truncation packed single precision floating-point values to packed doubleword integers.

5.6.3 Intel® SSE2 128-Bit SIMD Integer Instructions

Intel SSE2 SIMD integer instructions perform additional operations on packed words, doublewords, and quadwords contained in XMM and MMX registers.
MOVDQA Move aligned double quadword.
MOVDQU Move unaligned double quadword.
MOVQ2DQ Move quadword integer from MMX to XMM registers.
MOVDQ2Q Move quadword integer from XMM to MMX registers.
PMULUDQ Multiply packed unsigned doubleword integers.
ADDQ Add packed quadword integers.
PSUBQ Subtract packed quadword integers.
PSHUF LW Shuffle packed low words.
PSHUF HW Shuffle packed high words.
PSHUF D Shuffle packed doublewords.
PSLLDQ Shift double quadword left logical.
PSRLDQ Shift double quadword right logical.
PUNPCKHQDQ Unpack high quadwords.
PUNPCKLDQDQ Unpack low quadwords.

5.6.4 Intel® SSE2 Cacheability Control and Ordering Instructions

Intel SSE2 cacheability control instructions provide additional operations for caching of non-temporal data when storing data from XMM registers to memory. LFENCE and MFENCE provide additional control of instruction ordering on store operations.
CLFLUSH See Section 5.1.13.
LFENCE Serializes load operations.
MFENCE Serializes load and store operations.
PAUSE Improves the performance of "spin-wait loops".

5-22 Vol. 1
INSTRUCTION SET SUMMARY

5.7 INTEL® SSE3 INSTRUCTIONS

The Intel SSE3 extensions offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities. These instructions can be grouped into the following categories:

- One x87 FPU instruction used in integer conversion.
- One SIMD integer instruction that addresses unaligned data loads.
- Two SIMD floating-point packed ADD/SUB instructions.
- Four SIMD floating-point horizontal ADD/SUB instructions.
- Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions.
- Two thread synchronization instructions.

Intel SSE3 instructions can only be executed on Intel 64 and IA-32 processors that support Intel SSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

The sections that follow describe each subgroup.

5.7.1 Intel® SSE3 x87-FP Integer Conversion Instruction

FISTTP Behaves like the FISTP instruction but uses truncation, irrespective of the rounding mode specified in the floating-point control word (FCW).

5.7.2 Intel® SSE3 Specialized 128-Bit Unaligned Data Load Instruction

LDDQU Special 128-bit unaligned load designed to avoid cache line splits.

5.7.3 Intel® SSE3 SIMD Floating-Point Packed ADD/SUB Instructions

ADDSUBPS Performs single precision addition on the second and fourth pairs of 32-bit data elements within the operands; single precision subtraction on the first and third pairs.

ADDSUBPD Performs double precision addition on the second pair of quadwords, and double precision subtraction on the first pair.

5.7.4 Intel® SSE3 SIMD Floating-Point Horizontal ADD/SUB Instructions

HADDPS Performs a single precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.

HSUBPS Performs a single precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by...
subracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.

HADDPD Performs a double precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.

HSUBPD Performs a double precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

5.7.5 Intel® SSE3 SIMD Floating-Point LOAD/MOVE/DUPLICATE Instructions

MOVSHDUP Loads/moves 128 bits; duplicating the second and fourth 32-bit data elements.

MOVSLDUP Loads/moves 128 bits; duplicating the first and third 32-bit data elements.

MOVDDUP Loads/moves 64 bits (bits[63:0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register; duplicates the 64 bits from the source.

5.7.6 Intel® SSE3 Agent Synchronization Instructions

MONITOR Sets up an address range used to monitor write-back stores.

MWAIT Enables a logical processor to enter into an optimized state while waiting for a write-back store to the address range set up by the MONITOR instruction.

5.8 SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (SSSE3) INSTRUCTIONS

SSSE3 provide 32 instructions (represented by 14 mnemonics) to accelerate computations on packed integers. These include:

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

SSSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

The sections that follow describe each subgroup.

5.8.1 Horizontal Addition/Subtraction

PHADDW Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.

PHADDSW Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
PHADDD

Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.

PHSUBW

Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed 16-bit results are packed and written to the destination operand.

PHSUBSW

Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.

PHSUBD

Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

5.8.2 Packed Absolute Values

PABSB

Computes the absolute value of each signed byte data element.

PABSW

Computes the absolute value of each signed 16-bit data element.

PABSD

Computes the absolute value of each signed 32-bit data element.

5.8.3 Multiply and Add Packed Signed and Unsigned Bytes

PMADDUBSW

Multiplies each unsigned byte value with the corresponding signed byte value to produce an intermediate, 16-bit signed integer. Each adjacent pair of 16-bit signed values are added horizontally. The signed, saturated 16-bit results are packed to the destination operand.

5.8.4 Packed Multiply High with Round and Scale

PMULHRSW

Multiplies vertically each signed 16-bit integer from the destination operand with the corresponding signed 16-bit integer of the source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

5.8.5 Packed Shuffle Bytes

PHUFDB

Permutes each byte in place, according to a shuffle control mask. The least significant three or four bits of each shuffle control byte of the control mask form the shuffle index. The shuffle mask is unaffected. If the most significant bit (bit 7) of a shuffle control byte is set, the constant zero is written in the result byte.

5.8.6 Packed Sign

PSIGNB/W/D

Negates each signed integer element of the destination operand if the sign of the corresponding data element in the source operand is less than zero.
5.8.7 Packed Align Right

PALIGNR Source operand is appended after the destination operand forming an intermediate value of twice the width of an operand. The result is extracted from the intermediate value into the destination operand by selecting the 128-bit or 64-bit value that are right-aligned to the byte offset specified by the immediate value.

5.9 INTEL® SSE4 INSTRUCTIONS

Intel Streaming SIMD Extensions 4 (Intel SSE4) introduces 54 new instructions. 47 of the Intel SSE4 instructions are referred to as Intel SSE4.1 in this document, and 7 new Intel SSE4 instructions are referred to as Intel SSE4.2.

Intel SSE4.1 is targeted to improve the performance of media, imaging, and 3D workloads. Intel SSE4.1 adds instructions that improve compiler vectorization and significantly increase support for packed dword computation. The technology also provides a hint that can improve memory throughput when reading from uncacheable WC memory type.

The 47 Intel SSE4.1 instructions include:

• Two instructions perform packed dword multiplies.
• Two instructions perform floating-point dot products with input/output selects.
• One instruction performs a load with a streaming hint.
• Six instructions simplify packed blending.
• Eight instructions expand support for packed integer MIN/MAX.
• Four instructions support floating-point round with selectable rounding mode and precision exception override.
• Seven instructions improve data insertion and extractions from XMM registers
• Twelve instructions improve packed integer format conversions (sign and zero extensions).
• One instruction improves SAD (sum absolute difference) generation for small block sizes.
• One instruction aids horizontal searching operations.
• One instruction improves masked comparisons.
• One instruction adds qword packed equality comparisons.
• One instruction adds dword packing with unsigned saturation.

The Intel SSE4.2 instructions operating on XMM registers include:

• String and text processing that can take advantage of single-instruction multiple-data programming techniques.
• A SIMD integer instruction that enhances the capability of the 128-bit integer SIMD capability in SSE4.1.

5.10 INTEL® SSE4.1 INSTRUCTIONS

Intel SSE4.1 instructions can use an XMM register as a source or destination. Programming Intel SSE4.1 is similar to programming 128-bit Integer SIMD and floating-point SIMD instructions in Intel SSE/SSE2/SSE3/SSSE3. Intel SSE4.1 does not provide any 64-bit integer SIMD instructions operating on MMX registers. The sections that follow describe each subgroup.

5.10.1 Dword Multiply Instructions

PMULLD Returns four lower 32-bits of the 64-bit results of signed 32-bit integer multiplies.
PMULDQ Returns two 64-bit signed result of signed 32-bit integer multiplies.
5.10.2 Floating-Point Dot Product Instructions

DPPD  Perform double precision dot product for up to 2 elements and broadcast.
DPPS  Perform single precision dot product for up to 4 elements and broadcast.

5.10.3 Streaming Load Hint Instruction

MOVNTDQA  Provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region (a streaming line) to be fetched and held in a small set of temporary buffers ("streaming load buffers"). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be supplied from the streaming load buffer and can improve throughput.

5.10.4 Packed Blending Instructions

BLENDPD  Conditionally copies specified double precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDPS  Conditionally copies specified single precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDVPD  Conditionally copies specified double precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
BLENDVPS  Conditionally copies specified single precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
PBLENDVB  Conditionally copies specified byte elements in the source operand to the corresponding elements in the destination, using an implied mask.
PBLENDW  Conditionally copies specified word elements in the source operand to the corresponding elements in the destination, using an immediate byte control.

5.10.5 Packed Integer MIN/MAX Instructions

PMINUW  Compare packed unsigned word integers.
PMINUD  Compare packed unsigned dword integers.
PMINSB  Compare packed signed byte integers.
PMINSD  Compare packed signed dword integers.
PMAUXW  Compare packed unsigned word integers.
PMAXUD  Compare packed unsigned dword integers.
PMAXSB  Compare packed signed byte integers.
PMAXSD  Compare packed signed dword integers.

5.10.6 Floating-Point Round Instructions with Selectable Rounding Mode

ROUNDPS  Round packed single precision floating-point values into integer values and return rounded floating-point values.
ROUNDPD  Round packed double precision floating-point values into integer values and return rounded floating-point values.
ROUNDSS  Round the low packed single precision floating-point value into an integer value and return a rounded floating-point value.
ROUNSDD  Round the low packed double precision floating-point value into an integer value and return a rounded floating-point value.
5.10.7 Insertion and Extractions from XMM Registers

**EXTRACTPS** Extracts a single precision floating-point value from a specified offset in an XMM register and stores the result to memory or a general-purpose register.

**INSERTPS** Inserts a single precision floating-point value from either a 32-bit memory location or selected from a specified offset in an XMM register to a specified offset in the destination XMM register. In addition, INSERTPS allows zeroing out selected data elements in the destination, using a mask.

**PINSRB** Insert a byte value from a register or memory into an XMM register.

**PINSRD** Insert a dword value from 32-bit register or memory into an XMM register.

**PINSRQ** Insert a qword value from 64-bit register or memory into an XMM register.

**PEXTRB** Extract a byte from an XMM register and insert the value into a general-purpose register or memory.

**PEXTRW** Extract a word from an XMM register and insert the value into a general-purpose register or memory.

**PEXTRD** Extract a dword from an XMM register and insert the value into a general-purpose register or memory.

**PEXTRQ** Extract a qword from an XMM register and insert the value into a general-purpose register or memory.

5.10.8 Packed Integer Format Conversions

**PMOVSX BW** Sign extend the lower 8-bit integer of each packed word element into packed signed word integers.

**PMOVZX BW** Zero extend the lower 8-bit integer of each packed word element into packed signed word integers.

**PMOVSX BD** Sign extend the lower 8-bit integer of each packed dword element into packed signed dword integers.

**PMOVZX BD** Zero extend the lower 8-bit integer of each packed dword element into packed signed dword integers.

**PMOV SX WD** Sign extend the lower 16-bit integer of each packed dword element into packed signed dword integers.

**PMOVZX WD** Zero extend the lower 16-bit integer of each packed dword element into packed signed dword integers.

**PMOV SXB Q** Sign extend the lower 8-bit integer of each packed qword element into packed signed qword integers.

**PMOVZ XB Q** Zero extend the lower 8-bit integer of each packed qword element into packed signed qword integers.

**PMOV SXW Q** Sign extend the lower 16-bit integer of each packed qword element into packed signed qword integers.

**PMOVZ WX Q** Zero extend the lower 16-bit integer of each packed qword element into packed signed qword integers.

**PMOV SXD Q** Sign extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

**PMOVZ XD Q** Zero extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

5.10.9 Improved Sums of Absolute Differences (SAD) for 4-Byte Blocks

**MPSAD BW** Performs eight 4-byte wide Sum of Absolute Differences operations to produce eight word integers.
5.10.10 Horizontal Search
PHMINPOSUW Finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words. The resulting value and location (offset within the source) are packed into the low dword of the destination XMM register.

5.10.11 Packed Test
PTEST Performs a logical AND between the destination with this mask and sets the ZF flag if the result is zero. The CF flag (zero for TEST) is set if the inverted mask AND’d with the destination is all zeroes.

5.10.12 Packed Qword Equality Comparisons
PCMPEQQ 128-bit packed qword equality test.

5.10.13 Dword Packing With Unsigned Saturation
PACKUSDW Packs dword to word with unsigned saturation.

5.11 INTEL® SSE4.2 INSTRUCTION SET
Five of the Intel SSE4.2 instructions operate on XMM register as a source or destination. These include four text/string processing instructions and one packed quadword compare SIMD instruction. Programming these five Intel SSE4.2 instructions is similar to programming 128-bit Integer SIMD in Intel SSE2/SSSE3. Intel SSE4.2 does not provide any 64-bit integer SIMD instructions.

CRC32 operates on general-purpose registers and is summarized in Section 5.1.6. The sections that follow summarize each subgroup.

5.11.1 String and Text Processing Instructions
PCMPESTRI Packed compare explicit-length strings, return index in ECX/RCX.
PCMPESTRM Packed compare explicit-length strings, return mask in XMM0.
PCMPISTRI Packed compare implicit-length strings, return index in ECX/RCX.
PCMPISTRM Packed compare implicit-length strings, return mask in XMM0.

5.11.2 Packed Comparison SIMD Integer Instruction
PCMPGTQ Performs logical compare of greater-than on packed integer quadwords.

5.12 INTEL® AES-NI AND PCLMULQDQ
Six Intel® AES-NI instructions operate on XMM registers to provide accelerated primitives for block encryption/decryption using Advanced Encryption Standard (FIPS-197). The PCLMULQDQ instruction performs carry-less multiplication for two binary numbers up to 64-bit wide.

AESDEC Perform an AES decryption round using an 128-bit state and a round key.
AESDECLAST Perform the last AES decryption round using an 128-bit state and a round key.
AENSENC Perform an AES encryption round using an 128-bit state and a round key.
ASENCLAST Perform the last AES encryption round using an 128-bit state and a round key.
AESIMC Perform an inverse mix column transformation primitive.
INSTRUCTION SET SUMMARY

AESKEYGENASSIST Assist the creation of round keys with a key expansion schedule.
PCLMULQDQ Perform carryless multiplication of two 64-bit numbers.

5.13 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel® Advanced Vector Extensions (AVX) promote legacy 128-bit SIMD instruction sets that operate on the XMM register set to use a "vector extension" (VEX) prefix and operates on 256-bit vector registers (YMM). Almost all prior generations of 128-bit SIMD instructions that operate on XMM (but not on MMX registers) are promoted to support three-operand syntax with VEX-128 encoding.

VEX-prefix encoded Intel AVX instructions support 256-bit and 128-bit floating-point operations by extending the legacy 128-bit SIMD floating-point instructions to support three-operand syntax.

Additional functional enhancements are also provided with VEX-encoded Intel AVX instructions.

The list of Intel AVX instructions is included in the following tables:

- Table 14-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-3 lists 256-bit and 128-bit data movement and processing instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-4 lists functional enhancements of 256-bit Intel AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-5 lists 128-bit integer and floating-point instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-6 lists functional enhancements of 128-bit Intel AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-7 lists 128-bit data movement and processing instructions promoted from legacy instruction sets.

5.14 16-BIT FLOATING-POINT CONVERSION

Conversions between single precision floating-point (32-bit) and half precision floating-point (16-bit) data are provided by the VCVTPS2PH and VCVTPH2PS instructions, introduced beginning with the third generation of Intel Core processors based on Ivy Bridge microarchitecture:

VCVTPH2PS Convert eight/four data elements containing 16-bit floating-point data into eight/four single precision floating-point data.

VCVTPS2PH Convert eight/four data elements containing single precision floating-point data into eight/four 16-bit floating-point data.

Starting with the 4th generation Intel Xeon Scalable Processor Family based on Sapphire Rapids microarchitecture, Intel® AVX-512 instruction set architecture for FP16 was added, supporting a wide range of general-purpose numeric operations for 16-bit half precision floating-point values (binary16 in IEEE Standard 754-2019 for Floating-Point Arithmetic, aka half precision or FP16). Section 5.19 includes a list of these instructions.

5.15 FUSED-MULTIPLY-ADD (FMA)

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

- Table 14-15 lists FMA instruction sets.
5.16 INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

- Table 14-18 lists promoted vector integer instructions in AVX2.
- Table 14-19 lists new instructions in AVX2 that complements AVX.

5.17 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

- XABORT Abort an RTM transaction execution.
- XACQUIRE Prefix hint to the beginning of an HLE transaction region.
- XRELEASE Prefix hint to the end of an HLE transaction region.
- XBEGIN Transaction begin of an RTM transaction region.
- XEND Transaction end of an RTM transaction region.
- XTTEST Test if executing in a transactional region.
- XRESLDTRK Resume tracking load addresses.
- XSUSLDTRK Suspend tracking load addresses.

5.18 INTEL® SHA EXTENSIONS

Intel® SHA extensions provide a set of instructions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants.

- SHA1MSG1 Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
- SHA1MSG2 Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
- SHA1NEXTE Calculate SHA1 state E after four rounds.
- SHA1RND4 Perform four rounds of SHA1 operations.
- SHA256MSG1 Perform an intermediate calculation for the next four SHA256 message dwords.
- SHA256MSG2 Perform the final calculation for the next four SHA256 message dwords.
- SHA256RND4 Perform two rounds of SHA256 operations.

5.19 INTEL® ADVANCED VECTOR EXTENSIONS 512 (INTEL® AVX-512)

The Intel® AVX-512 family comprises a collection of 512-bit SIMD instruction sets to accelerate a diverse range of applications. Intel AVX-512 instructions provide a wide range of functionality that support programming in 512-bit, 256 and 128-bit vector register, plus support for opmask registers and instructions operating on opmask registers.

The collection of 512-bit SIMD instruction sets in Intel AVX-512 include new functionality not available in Intel AVX and Intel AVX2, and promoted instructions similar to equivalent ones in Intel AVX/Intel AVX2 but with enhancement provided by opmask registers not available to VEX-encoded Intel AVX/Intel AVX2. Some instruction mnemonics in Intel AVX/Intel AVX2 that are promoted into Intel AVX-512 can be replaced by new instruction mnemonics that are available only with EVEX encoding, e.g., VBBROADCASTF128 into VBBROADCASTF32X4. Details of EVEX instruction encoding are discussed in Section 2.7, “Intel® AVX-512 Encoding,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. Starting with the 4th generation Intel Xeon Scalable Processor Family, an Intel AVX-512 instruction set architecture for FP16 was added, supporting a wide range of
INSTRUCTION SET SUMMARY

general-purpose numeric operations for 16-bit half precision floating-point values, which complements the existing 32-bit and 64-bit floating-point instructions already available in the Intel Xeon processor-based products.

512-bit instruction mnemonics in AVX-512F instructions that are not Intel AVX or AVX2 promotions include:

VALIGND/Q       Perform dword/qword alignment of two concatenated source vectors.
VBLENDMPD/PS    Replace the VBLENDVDP/PS instructions (using opmask as select control).
VCOMPRESSPD/PS  Compress packed DP or SP elements of a vector.
VCVT(T)PD2UDQ   Convert packed DP FP elements of a vector to packed unsigned 32-bit integers.
VCVT(T)PS2UDQ   Convert packed SP FP elements of a vector to packed unsigned 32-bit integers.
VCVTQ2PD/PS     Convert packed signed 64-bit integers to packed DP/SP FP elements.
VCVT(T)SD2USI   Convert the low DP FP element of a vector to an unsigned integer.
VCVT(T)SS2USI   Convert the low SP FP element of a vector to an unsigned integer.
VCVTDQ2PD/PS    Convert packed unsigned 32-bit integers to packed DP/SP FP elements.
VCVTUSI2USD/S   Convert an unsigned integer to the low DP/SP FP element and merge to a vector.
VEXPANDPD/PS    Expand packed DP or SP elements of a vector.
VEXTRACTF32X4/64X4 Extract a vector from a full-length vector with 32/64-bit granular update.
VEXTRACTI32X4/66X4 Extract a vector from a full-length vector with 32/64-bit granular update.
VFIXUPIMMPD/PS  Perform fix-up to special values in DP/SP FP vectors.
VFIXUPIMMSD/SS  Perform fix-up to special values of the low DP/SP FP element.
VGETEXPPD/PS    Convert the exponent of DP/SP FP elements of a vector into FP values.
VGETEXPSD/SS    Convert the exponent of the low DP/SP FP element in a vector into FP value.
VGETMANTPD/PS   Convert the mantissa of DP/SP FP elements of a vector into FP values.
VGETMANTSD/SS   Convert the mantissa of the low DP/SP FP element of a vector into FP value.
VINSETRT32X4/66X4 Insert a 128/256-bit vector into a full-length vector with 32/64-bit granular update.
VMOVQ32A32/64A32 VMOVDQA32/64 VMOVDQA with 32/64-bit granular conditional update.
VMOVQ32U32/64A32 VMOVDQU32/64 VMOVDQU with 32/64-bit granular conditional update.
VPBLENMD/Q      Blend dword/qword elements using opmask as select control.
VPBROADCAST/Q   Broadcast from general-purpose register to vector register.
VPCMPD/UD       Compare packed signed/unsigned dwords using specified primitive.
VPCMPQ/UQ       Compare packed signed/unsigned quadwords using specified primitive.
VPCOMPRESSQ/D   Compress packed 64/32-bit elements of a vector.
VPERM12D/Q      Full permute of two tables of dword/qword elements overwriting the index vector.
VPERM12PD/PS    Full permute of two tables of DP/SP elements overwriting the index vector.
VPERM2T/Q       Full permute of two tables of dword/qword elements overwriting one source table.
VPERM2PD/PS     Full permute of two tables of DP/SP elements overwriting one source table.
VEXPANDD/Q      Expand packed dword/qword elements of a vector.
VPMAXSQ         Compute maximum of packed signed 64-bit integer elements.
VPMAXUD/UQ      Compute maximum of packed unsigned 32/64-bit integer elements.
VPMINSQ         Compute minimum of packed signed 64-bit integer elements.
VPMINUD/UQ      Compute minimum of packed unsigned 32/64-bit integer elements.
VPMOV(S|US)QB   Down convert qword elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)QW   Down convert qword elements in a vector to word elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)QD   Down convert qword elements in a vector to dword elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)DB   Down convert dword elements in a vector to byte elements using truncation (saturation | unsigned saturation).
INSTRUCTION SET SUMMARY

VPMOV(S|US)DW  Down convert dword elements in a vector to word elements using truncation (saturation | unsigned saturation).
VPROLD/Q  Rotate dword/qword element left by a constant shift count with conditional update.
VPROLVD/Q  Rotate dword/qword element left by shift counts specified in a vector with conditional update.
VPRORD/Q  Rotate dword/qword element right by a constant shift count with conditional update.
VPRORRD/Q  Rotate dword/qword element right by shift counts specified in a vector with conditional update.
VPSCATTERDD/DQ  Scatter dword/qword elements in a vector to memory using dword indices.
VPSCATTERQD/QQ  Scatter dword/qword elements in a vector to memory using qword indices.
VPSRAQ  Shift qwords right by a constant shift count and shifting in sign bits.
VPSRAQ  Shift qwords right by shift counts in a vector and shifting in sign bits.
VPTESTNMD/Q  Perform bitwise NAND of dword/qword elements of two vectors and write results to opmask.
VPTESTMD/Q  Perform bitwise AND of dword/qword elements of two vectors and write results to opmask.
VRCP14PD/PS  Compute approximate reciprocals of packed DP/SP FP elements of a vector.
VRCP14SD/SS  Compute the approximate reciprocal of the low DP/SP FP element of a vector.
VRNDSCALEPD/PS  Round packed DP/SP FP elements of a vector to specified number of fraction bits.
VRNDSCALESD/SS  Round the low DP/SP FP element of a vector to specified number of fraction bits.
VRSQRT14PD/PS  Compute approximate reciprocals of square roots of packed DP/SP FP elements of a vector.
VRSQRT14SD/SS  Compute the approximate reciprocal of square root of the low DP/SP FP element of a vector.
VSCELD/PD  Multiply packed DP/SP FP elements of a vector by powers of two with exponents specified in the corresponding element of a second vector.
VSCELD/SS  Multiply the low DP/SP FP element of a vector by powers of two with exponent specified in the corresponding element of a second vector.
VSCATTERDD/DQ  Scatter SP/DP FP elements in a vector to memory using dword indices.
VSCATTERQD/QQ  Scatter SP/DP FP elements in a vector to memory using qword indices.
VSHUFF32X4/64X2  Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.
VSHUFF32X4/64X2  Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.

512-bit instruction mnemonics in AVX-512DQ that are not Intel AVX or AVX2 promotions include:
VCVT(T)PD2QQ  Convert packed DP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PD2UQQ  Convert packed DP FP elements of a vector to packed unsigned 64-bit integers.
VCVT(T)PS2QQ  Convert packed SP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PS2UQQ  Convert packed SP FP elements of a vector to packed unsigned 64-bit integers.
VCVTUQQ2PD/PS  Convert packed unsigned 64-bit integers to packed DP/SP FP elements.
VEXTRACTF64X2  Extract a vector from a full-length vector with 64-bit granular update.
VEXTRACTI64X2  Extract a vector from a full-length vector with 64-bit granular update.
VFCLASSPD/PS  Test packed DP/SP FP elements in a vector by numeric/special-value category.
VFCLASSSD/SS  Test the low DP/SP FP element by numeric/special-value category.
VINSETF64X2  Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VINSETI64X2  Insert a 128-bit vector into a full-length vector with 64-bit granular update.
PROMOV2D/Q  Convert opmask register to vector register in 32/64-bit granularity.
PROMOV2D/Q  Convert a vector register in 32/64-bit granularity to an opmask register.
INSTRUCTION SET SUMMARY

VPMULLQ Multiply packed signed 64-bit integer elements of two vectors and store low 64-bit signed result.

VRANGEPD/PS Perform RANGE operation on each pair of DP/SP FP elements of two vectors using specified range primitive in imm8.

VRANGESD/SS Perform RANGE operation on the pair of low DP/SP FP element of two vectors using specified range primitive in imm8.


512-bit instruction mnemonics in AVX-512BW that are not Intel AVX or AVX2 promotions include:

VDBPSADBW Double block packed Sum-Absolute-Differences on unsigned bytes.

VMOVQ8/16 VMOVDQU with 8/16-bit granular conditional update.

VPBLENDMB Replaces the VPBLENDVB instruction (using opmask as select control).

VPBLENDMW Blend word elements using opmask as select control.

VPBROADCASTB/W Broadcast from general-purpose register to vector register.

VPCMPB/UB Compare packed signed/unsigned bytes using specified primitive.

VPCMPB/UW Compare packed signed/unsigned words using specified primitive.

VPERMW Permute packed word elements.

VPERMIB/W Full permute from two tables of byte/word elements overwriting the index vector.

VPMOVB/W Convert opmask register to vector register in 8/16-bit granularity.

VPMOVB2M/W2M Convert a vector register in 8/16-bit granularity to an opmask register.

VPMOV(S|US)WB Down convert word elements in a vector to byte elements using truncation (saturation | unsigned saturation).

VPSLLVW Shift word elements in a vector left by shift counts in a vector.

VPSRAWW Shift words right by shift counts in a vector and shifting in sign bits.

VPSRLVW Shift word elements in a vector right by shift counts in a vector.

VPTESTNMB/W Perform bitwise NAND of byte/word elements of two vectors and write results to opmask.

VPTESTMB/W Perform bitwise AND of byte/word elements of two vectors and write results to opmask.

512-bit instruction mnemonics in AVX-512CD that are not Intel AVX or AVX2 promotions include:

VPBROADCASTM Broadcast from opmask register to vector register.

VPCONFLICTD/Q Detect conflicts within a vector of packed 32/64-bit integers.

VPLZCNTD/Q Count the number of leading zero bits of packed dword/qword elements.

Opmask instructions include:

KADDB/W/D/Q Add two 8/16/32/64-bit opmasks.

KANDB/W/D/Q Logical AND two 8/16/32/64-bit opmasks.

KANDNB/W/D/Q Logical AND NOT two 8/16/32/64-bit opmasks.

KMOVW/D/Q Move from or move to opmask register of 8/16/32/64-bit data.

KNOTB/W/D/Q Bitwise NOT of two 8/16/32/64-bit opmasks.

KORB/W/D/Q Logical OR two 8/16/32/64-bit opmasks.

KORTESTB/W/D/Q Update EFLAGS according to the result of bitwise OR of two 8/16/32/64-bit opmasks.

KSHIFTL/W/D/Q Shift left 8/16/32/64-bit opmask by specified count.

KSHIFTR/W/D/Q Shift right 8/16/32/64-bit opmask by specified count.

KTESTB/W/D/Q Update EFLAGS according to the result of bitwise TEST of two 8/16/32/64-bit opmasks.
KUNPCKBW/WD/DQ: Unpack and interleave two 8/16/32-bit opmasks into 16/32/64-bit mask.
KXNORB/W/D/Q: Bitwise logical XNOR of two 8/16/32/64-bit opmasks.
K XORB/W/D/Q: Logical XOR of two 8/16/32/64-bit opmasks.

512-bit instruction mnemonics in AVX-512ER include:
VEXP2PD/PS: Compute approximate base-2 exponential of packed DP/SP FP elements of a vector.
VEXP2SD/SS: Compute approximate base-2 exponential of the low DP/SP FP element of a vector.
VRCP28PD/PS: Compute approximate reciprocals to 28 bits of packed DP/SP FP elements of a vector.
VRCP28SD/SS: Compute the approximate reciprocal to 28 bits of the low DP/SP FP element of a vector.
VRSQRT28PD/PS: Compute approximate reciprocals of square roots to 28 bits of packed DP/SP FP elements of a vector.
VRSQRT28SD/SS: Compute the approximate reciprocal of square root to 28 bits of the low DP/SP FP element of a vector.

512-bit instruction mnemonics in AVX-512PF include:
VGATHERPF0DPD/PS: Sparse prefetch of packed DP/SP FP vector with T0 hint using dword indices.
VGATHERPF0QPD/PS: Sparse prefetch of packed DP/SP FP vector with T0 hint using qword indices.
VGATHERPF1DPD/PS: Sparse prefetch of packed DP/SP FP vector with T1 hint using dword indices.
VGATHERPF1QPD/PS: Sparse prefetch of packed DP/SP FP vector with T1 hint using qword indices.
VSCATTERPF0DPD/PS: Sparse prefetch of packed DP/SP FP vector with T0 hint to write using dword indices.
VSCATTERPF0QPD/PS: Sparse prefetch of packed DP/SP FP vector with T0 hint to write using qword indices.
VSCATTERPF1DPD/PS: Sparse prefetch of packed DP/SP FP vector with T1 hint to write using dword indices.
VSCATTERPF1QPD/PS: Sparse prefetch of packed DP/SP FP vector with T1 hint to write using qword indices.

512-bit instruction mnemonics in AVX512-FP16 include:
VADDPH/SH: Add packed/scalar FP16 values.
VCMPHP/SH: Compare packed/scalar FP16 values.
VCOMISH: Compare scalar ordered FP16 values and set EFLAGS.
VCVTDQ2PH: Convert packed signed doubleword integers to packed FP16 values.
VCVTPD2PH: Convert packed double precision FP values to packed FP16 values.
VCVTPH2DQ/QQ: Convert packed FP16 values to signed doubleword/quadword integers.
VCVTPH2PD: Convert packed FP16 values to FP64 values.
VCVTPH2PS[X]: Convert packed FP16 values to single precision floating-point values.
VCVTPH2QQ: Convert packed FP16 values to signed quadword integer values.
VCVTPH2UDQ/QQ: Convert packed FP16 values to unsigned doubleword/quadword integers.
VCVTPH2UW/W: Convert packed FP16 values to unsigned/signed word integers.
VCVTPS2PH[X]: Convert packed single precision floating-point values to packed FP16 values.
VCVTQ2PH: Convert packed signed quadword integers to packed FP16 values.
VCVTS2D: Convert low FP64 value to an FP16 value.
VCVTSH2D/Q: Convert low FP16 value to an FP64/FP32 value.
VCVTSH2ISI/USI: Convert low FP16 value to signed/unsigned integer.
VCVTSH2ISI: Convert a signed doubleword/quadword integer to an FP16 value.
VCVTS2W: Convert low FP32 value to an FP16 value.
VCVTTPH2DQ/QQ: Convert with truncation packed FP16 values to signed doubleword/quadword integers.
VCVTTPH2UDQ/QQ: Convert with truncation packed FP16 values to unsigned doubleword/quadword integers.
VCVTTPH2UW/W: Convert packed FP16 values to unsigned/signed word integers.
INSTRUCTION SET SUMMARY

VCVTTSH2SI/USI Convert with truncation low FP16 value to a signed/unsigned integer.
VCVTUDQ2PH Convert packed unsigned doubleword integers to packed FP16 values.
VCVTUQ2PH Convert packed unsigned quadword integers to packed FP16 values.
VCVTUSI2SH Convert unsigned doubleword integer to an FP16 value.
VCVTUW2PH Convert packed unsigned word integers to FP16 values.
VCVTW2PH Convert packed signed word integers to FP16 values.
VDIVPH/SH Divide packed/scalar FP16 values.
VF[C]MADDCPH Complex multiply and accumulate FP16 values.
VF[C]MADDCSH Complex multiply and accumulate scalar FP16 values.
VF[C]MULCPH Complex multiply FP16 values.
VF[C]MULCSH Complex multiply scalar FP16 values.
VF[C]MULCSSH Complex multiply scalar FP16 values.
VF[,N]MADD[132,213,231]PH Fused multiply-add of packed FP16 values.
VFMSUBADD[132,213,231]PH Fused multiply-alternating subtract/add of packed FP16 values.
VF[,N]MSUB[132,213,231]PH Fused multiply-subtract of packed FP16 values.
VF[,]N]MSUB[132,213,231]SH Fused multiply-subtract of scalar FP16 values.
VFPCLASSPH/SH Test types of packed/scalar FP16 values.
VGETEXPPH/SH Convert exponents of packed/scalar FP16 values to FP16 values.
VGETMANTPH/SH Extract FP16 vector of normalized mantissas from FP16 vector/scalar.
VMAXPH/PS Return maximum of packed/scalar FP16 values.
VMINPH/PS Return minimum of packed/scalar FP16 values.
VMOVSH Move scalar FP16 value.
VMOVVW Move word.
VMPULPH/SH Multiply packed/scalar FP16 values.
VRCPPH/SH Compute reciprocals of packed/scalar FP16 values.
VREDUCEPH/SH Perform reduction transformation on packed/scalar FP16 values.
VRNSCALEPH/SH Round packed/scalar FP16 values to include a given number of fraction bits.
VRSQRTPH/SH Compute reciprocals of square roots of packed/scalar FP16 values.
VSCALEPH/SH Scale packed/scalar FP16 values with FP16 values.
VSRQTPH/SH Compute square root of packed/scalar FP16 values.
VSUBPH/SH Subtract packed/scalar FP16 values.
VUCOMISH Unordered compare scalar FP16 values and set EFLAGS.

5.20 SYSTEM INSTRUCTIONS

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

CLAC Clear AC Flag in EFLAGS register.
STAC Set AC Flag in EFLAGS register.
LGDT Load global descriptor table (GDT) register.
SGDT Store global descriptor table (GDT) register.
LLDT Load local descriptor table (LDT) register.
SLDT Store local descriptor table (LDT) register.
LTR Load task register.
STR Store task register.
LIDT  Load interrupt descriptor table (IDT) register.
SIDT  Store interrupt descriptor table (IDT) register.
MOV   Load and store control registers.
LMSW  Load machine status word.
SMSW  Store machine status word.
CLTS  Clear the task-switched flag.
ARPL  Adjust requested privilege level.
LAR   Load access rights.
LSL   Load segment limit.
VERR  Verify segment for reading.
VERW  Verify segment for writing.
MOV   Load and store debug registers.
INVD  Invalidate cache, no writeback.
WBINVD Invalidate cache, with writeback.
INVLP  Invalidate TLB Entry.
INVP  Invalidate Process-Context Identifier.
LOCK (prefix) Perform atomic access to memory (can be applied to a number of general purpose instructions that provide memory source/destination access).
HLT   Halt processor.
RSM   Return from system management mode (SMM).
RDMSR Read model-specific register.
WRMSR Write model-specific register.
RDPMC Read performance monitoring counters.
RDTSC Read time stamp counter.
RDTSCP Read time stamp counter and processor ID.
SYSENC Fast System Call, transfers to a flat protected mode kernel at CPL = 0.
SYSEXIT Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE  Save processor extended states to memory.
XSAVEN Save processor extended states with compaction to memory.
XSAVEOPT Save processor extended states to memory, optimized.
XSAVES Save processor supervisor-mode extended states to memory.
XRSTOR Restore processor extended states from memory.
XRSTORS Restore processor supervisor-mode extended states from memory.
XGETBV Reads the state of an extended control register.
XSETBV Writes the state of an extended control register.
RDFSBSAE Reads from FS base address at any privilege level.
RDGSBSAE Reads from GS base address at any privilege level.
WRDFSBSAE Writes to FS base address at any privilege level.
WRGSBSAE Writes to GS base address at any privilege level.

5.21 64-BIT MODE INSTRUCTIONS

The following instructions are introduced in 64-bit mode. This mode is a sub-mode of IA-32e mode.
CDQE  Convert doubleword to quadword.
CMPSQ Compare string operands.
CMPXCHG16B Compare RDX:RAX with m128.
INSTRUCTION SET SUMMARY

LODSQ Load qword at address (R)SI into RAX.
MOVSQ Move qword from address (R)SI to (R)DI.
MOVZX (64-bits) Move bytes/words to doublewords/quadwords, zero-extension.
STOSQ Store RAX at address RDI.
SWAPGS Exchanges current GS base register value with value in MSR address C0000102H.
SYSCALL Fast call to privilege level 0 system procedures.
SYSRET Return from fast system call.

5.22 VIRTUAL-MACHINE EXTENSIONS

The behavior of the VMCS-maintenance instructions is summarized below:

VMPTRLD Takes a single 64-bit source operand in memory. It makes the referenced VMCS active and current.
VMPTRST Takes a single 64-bit destination operand that is in memory. Current-VMCS pointer is stored into the destination operand.
VMCLEAR Takes a single 64-bit operand in memory. The instruction sets the launch state of the VMCS referenced by the operand to "clear", renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region.
VMREAD Reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand.
VMWRITE Writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand.

The behavior of the VMX management instructions is summarized below:
VMLAUNCH Launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMRESUME Resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMXOFF Causes the processor to leave VMX operation.
VMXON Takes a single 64-bit source operand in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

The behavior of the VMX-specific TLB-management instructions is summarized below:
INVEPT Invalidate cached Extended Page Table (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.
INVVPID Invalidate cached mappings of address translation based on the Virtual Processor ID (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:
VMCALL Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
VMFUNC Allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

5.23 SAFER MODE EXTENSIONS

The behavior of the GETSEC instruction leaves of the Safer Mode Extensions (SMX) are summarized below:
GETSEC[CAPABILITIES]Returns the available leaf functions of the GETSEC instruction.
INSTRUCTION SET SUMMARY

GETSEC[ENTERACCS] Loads an authenticated code chipset module and enters authenticated code execution mode.
GETSEC[EXITAC] Exits authenticated code execution mode.
GETSEC[ENTER] Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.
GETSEC[EXIT] Exits the MLE.
GETSEC[PARAMETERS] Returns SMX related parameter information.
GETSEC[SMCRTL] SMX mode control.
GETSEC[WAKEUP] Wakes up sleeping logical processors inside an MLE.

5.24 INTEL® MEMORY PROTECTION EXTENSIONS

Intel Memory Protection Extensions (Intel MPX) provides a set of instructions to enable software to add robust bounds checking capability to memory references. Details of Intel MPX are described in Appendix E, "Intel® Memory Protection Extensions."

BNDMK Create a LowerBound and an UpperBound in a register.
BNDCQ Check the address of a memory reference against a LowerBound.
BNDCU Check the address of a memory reference against an UpperBound in 1's compliment form.
BNDCN Check the address of a memory reference against an UpperBound not in 1's compliment form.
BNDMOV Copy or load from memory of the LowerBound and UpperBound to a register.
BNDMOV Store to memory of the LowerBound and UpperBound from a register.
BNDLDX Load bounds using address translation.
BNDSTX Store bounds using address translation.

5.25 INTEL® SOFTWARE GUARD EXTENSIONS

Intel Software Guard Extensions (Intel SGX) provide two sets of instruction leaf functions to enable application software to instantiate a protected container, referred to as an enclave. The enclave instructions are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Details of Intel SGX are described in Chapter 34 through Chapter 40 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D.

The first implementation of Intel SGX is also referred to as SGX1, it is introduced with the 6th Generation Intel Core Processors. The leaf functions supported in SGX1 are shown in Table 5-3.

<table>
<thead>
<tr>
<th>Supervisor Instruction</th>
<th>Description</th>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLS[EADD]</td>
<td>Add a page</td>
<td>ENCLU[EENTER]</td>
<td>Enter an Enclave</td>
</tr>
<tr>
<td>ENCLS[EQUEUE]</td>
<td>Block an EPC page</td>
<td>ENCLU[EEXIT]</td>
<td>Exit an Enclave</td>
</tr>
<tr>
<td>ENCLS[ECREATE]</td>
<td>Create an enclave</td>
<td>ENCLU[EGETKEY]</td>
<td>Create a cryptographic key</td>
</tr>
<tr>
<td>ENCLS[EDBGRD]</td>
<td>Read data by debugger</td>
<td>ENCLU[EREPORT]</td>
<td>Create a cryptographic report</td>
</tr>
<tr>
<td>ENCLS[EDBGWR]</td>
<td>Write data by debugger</td>
<td>ENCLU[ERESUME]</td>
<td>Re-enter an Enclave</td>
</tr>
<tr>
<td>ENCLS[EEXTEND]</td>
<td>Extend EPC page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EINIT]</td>
<td>Initialize an enclave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ELDB]</td>
<td>Load an EPC page as blocked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ELDU]</td>
<td>Load an EPC page as unblocked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EPA]</td>
<td>Add version array</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.26 SHADOW STACK MANAGEMENT INSTRUCTIONS

Shadow stack management instructions allow the program and run-time to perform operations like recovering from control protection faults, shadow stack switching, etc. The following instructions are provided.

- **CLRSSBSY**: Clear busy bit in a supervisor shadow stack token.
- **INCSSP**: Increment the shadow stack pointer (SSP).
- **RDSSP**: Read shadow stack point (SSP).
- **RSTORSSP**: Restore a shadow stack pointer (SSP).
- **SAVEPREVSSP**: Save previous shadow stack pointer (SSP).
- **SETSSBSY**: Set busy bit in a supervisor shadow stack token.
- **WRSS**: Write to a shadow stack.
- **WRUSS**: Write to a user mode shadow stack.

5.27 CONTROL TRANSFER TERMINATING INSTRUCTIONS

- **ENDBR32**: Terminate an Indirect Branch in 32-bit and Compatibility Mode.
- **ENDBR64**: Terminate an Indirect Branch in 64-bit Mode.

5.28 INTEL® AMX INSTRUCTIONS

- **LDTILECFG**: Load tile configuration.
- **STTILECFG**: Store tile configuration.
- **TDPBF16PS**: Dot product of BF16 tiles accumulated into packed single precision tile.
- **TDPBSSD**: Dot product of signed bytes with dword accumulation.
- **TDPBSUD**: Dot product of signed/unsigned bytes with dword accumulation.
- **TDPBUSD**: Dot product of unsigned/signed bytes with dword accumulation.
- **TDPBUUD**: Dot product of unsigned bytes with dword accumulation.
- **TILELOADD**: Load data into tile.
- **TILELOADDT1**: Load data into tile with hint to optimize data caching.
- **TILERELEASE**: Release tile.
- **TILESTORED**: Store tile.
- **TILEZERO**: Zero tile.

5.29 USER INTERRUPT INSTRUCTIONS

- **CLUI**: Clear user interrupt flag.
- **SENDUIPI**: Send user interprocessor interrupt.
- **STUI**: Set user interrupt flag.
- **TESTUI**: Determine user interrupt flag.

### Table 5-3. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

<table>
<thead>
<tr>
<th>Supervisor Instruction</th>
<th>Description</th>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLS[EREMOVE]</td>
<td>Remove a page from EPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ETRACK]</td>
<td>Activate EBLOCK checks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EWB]</td>
<td>Write back/invalidate an EPC page</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
UIRET  User-interrupt return.

5.30  **ENQUEUE STORE INSTRUCTIONS**

ENQCMD  Enqueue command.
ENQCMDS  Enqueue command supervisor.
This chapter describes the facilities in the Intel 64 and IA-32 architectures for executing calls to procedures or subroutines. It also describes how interrupts and exceptions are handled from the perspective of an application programmer.

### 6.1 Procedure Call Types

The processor supports procedure calls in the following two different ways:

- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions.

Both of these procedure call mechanisms use the procedure stack, commonly referred to simply as "the stack," to save the state of the calling procedure, pass parameters to the called procedure, and store local variables for the currently executing procedure.

The processor’s facilities for handling interrupts and exceptions are similar to those used by the CALL and RET instructions.

Processors that support Control-Flow Enforcement Technology (CET) support an additional stack referred to as "the shadow stack". The CALL instruction, when shadow stacks are enabled, additionally saves the state of the calling procedure on the shadow stack; and the RET instruction restores the state of the calling procedure if the state on the stack and the shadow stack match.

### 6.2 Stacks

The stack (see Figure 6-1) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. When using the flat memory model, the stack can be located anywhere in the linear address space for the program. A stack can be up to 4 GBytes long, the maximum size of a segment.

Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows down in memory (towards lesser addresses) when items are pushed on the stack and shrinks up (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory.

When a system sets up many stacks, only one stack—the current stack—is available at a time. The current stack is the one contained in the segment referenced by the SS register.
The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

### 6.2.1 Setting Up a Stack

To set a stack and establish it as the current stack, the program or operating system/executive must do the following:

1. Establish a stack segment.
2. Load the segment selector for the stack segment into the SS register using a MOV, POP, or LSS instruction.
3. Load the stack pointer for the stack into the ESP register using a MOV, POP, or LSS instruction. The LSS instruction can be used to load the SS and ESP registers in one operation.


### 6.2.2 Stack Alignment

The stack pointer for a stack segment should be aligned on 16-bit (word) or 32-bit (double-word) boundaries, depending on the width of the stack segment. The D flag in the segment descriptor for the current code segment sets the stack-segment width (see "Segment Descriptors" in Chapter 3, “ Protected-Mode Memory Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). The PUSH and POP instructions use the D flag to determine how much to decrement or increment the stack pointer on a push or pop operation, respectively. When the stack width is 16 bits, the stack pointer is incremented or decremented in 16-bit increments; when the width is 32 bits, the stack pointer is incremented or decremented in 32-bit increments. Pushing a 16-bit value onto a 32-bit wide stack can result in stack misaligned (that is, the stack pointer is not aligned on a double-
word boundary). One exception to this rule is when the contents of a segment register (a 16-bit segment selector) are pushed onto a 32-bit wide stack. Here, the processor automatically aligns the stack pointer to the next 32-bit boundary.

The processor does not check stack pointer alignment. It is the responsibility of the programs, tasks, and system procedures running on the processor to maintain proper alignment of stack pointers. Misaligning a stack pointer can cause serious performance degradation and in some instances program failures.

6.2.3 Address-Size Attributes for Stack Accesses

Instructions that use the stack implicitly (such as the PUSH and POP instructions) have two address-size attributes each of either 16 or 32 bits. This is because they always have the implicit address of the top of the stack, and they may also have an explicit memory address (for example, PUSH Array1[EBX]). The attribute of the explicit address is determined by the D flag of the current code segment and the presence or absence of the 67H address-size prefix.

The address-size attribute of the top of the stack determines whether SP or ESP is used for the stack access. Stack operations with an address-size attribute of 16 use the 16-bit SP stack pointer register and can use a maximum stack address of FFFFH; stack operations with an address-size attribute of 32 bits use the 32-bit ESP register and can use a maximum address of FFFFFFFFH. The default address-size attribute for data segments used as stacks is controlled by the B flag of the segment's descriptor. When this flag is clear, the default address-size attribute is 16; when the flag is set, the address-size attribute is 32.

6.2.4 Procedure Linking Information

The processor provides two pointers for linking of procedures: the stack-frame base pointer and the return instruction pointer. When used in conjunction with a standard software procedure-call technique, these pointers permit reliable and coherent linking of procedures.

6.2.4.1 Stack-Frame Base Pointer

The stack is typically divided into frames. Each stack frame can then contain local variables, parameters to be passed to another procedure, and procedure linking information. The stack-frame base pointer (contained in the EBP register) identifies a fixed reference point within the stack frame for the called procedure. To use the stack-frame base pointer, the called procedure typically copies the contents of the ESP register into the EBP register prior to pushing any local variables on the stack. The stack-frame base pointer then permits easy access to data structures passed on the stack, to the return instruction pointer, and to local variables added to the stack by the called procedure.

Like the ESP register, the EBP register automatically points to an address in the current stack segment (that is, the segment specified by the current contents of the SS register).

6.2.4.2 Return Instruction Pointer

Prior to branching to the first instruction of the called procedure, the CALL instruction pushes the address in the EIP register onto the current stack. This address is then called the return-instruction pointer and it points to the instruction where execution of the calling procedure should resume following a return from the called procedure. Upon returning from a called procedure, the RET instruction pops the return-instruction pointer from the stack back into the EIP register. Execution of the calling procedure then resumes.

The processor does not keep track of the location of the return-instruction pointer. It is thus up to the programmer to ensure that stack pointer is pointing to the return-instruction pointer on the stack, prior to issuing a RET instruction. A common way to reset the stack pointer to the point to the return-instruction pointer is to move the contents of the EBP register into the ESP register. If the EBP register is loaded with the stack pointer immediately following a procedure call, it should point to the return instruction pointer on the stack.

The processor does not require that the return instruction pointer point back to the calling procedure. Prior to executing the RET instruction, the return instruction pointer can be manipulated in software to point to any address.
in the current code segment (near return) or another code segment (far return). Performing such an operation, however, should be undertaken very cautiously, using only well defined code entry points.

6.2.5 Stack Behavior in 64-Bit Mode

In 64-bit mode, address calculations that reference SS segments are treated as if the segment base is zero. Fields (base, limit, and attribute) in segment descriptor registers are ignored. SS DPL is modified such that it is always equal to CPL. This will be true even if it is the only field in the SS descriptor that is modified.

Registers E(SP), E(IP) and E(BP) are promoted to 64-bits and are re-named RSP, RIP, and RBP respectively. Some forms of segment load instructions are invalid (for example, LDS, POP ES).

PUSH/POP instructions increment/decrement the stack using a 64-bit width. When the contents of a segment register is pushed onto 64-bit stack, the pointer is automatically aligned to 64 bits (as with a stack that has a 32-bit width).

6.3 SHADOW STACKS

A shadow stack is a second stack used exclusively for control transfer operations. This stack is separate from the procedure stack. The shadow stack is not used to store data, hence is not explicitly writeable by software. Writes to the shadow stack are restricted to control transfer instructions and shadow stack management instructions. Shadow stacks can be enabled separately for privilege level 3 (user mode) or privilege levels less than 3 (supervisor mode).

Shadow stacks are active only in protected mode with paging enabled. Shadow stacks cannot be enabled for a program executing in virtual 8086 mode.

Processors that support shadow stacks have an architectural register called the shadow stack pointer (SSP) that points to the current top of the shadow stack. The SSP cannot be directly encoded as a source, destination, or memory operand in instructions. The width of the shadow stack is 32-bit in 32-bit/compatibility mode, and is 64-bit in 64-bit mode. The address-size attribute of the shadow stack is likewise 32-bit in 32-bit/compatibility mode, and 64-bit in 64-bit mode.

The size of the shadow stack pushes and pops for far CALL and call to interrupt/exception handlers is fixed at 64 bits, and the processor uses 8-byte, zero padded stores for these pushes in 32-bit/compatibility modes.

6.4 CALLING PROCEDURES USING CALL AND RET

The CALL instruction allows control transfers to procedures within the current code segment (near call) and in a different code segment (far call). Near calls usually provide access to local procedures within the currently running program or task. Far calls are usually used to access operating system procedures or procedures in a different task. See "CALL—Call Procedure" in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for a detailed description of the CALL instruction.

The RET instruction also allows near and far returns to match the near and far versions of the CALL instruction. In addition, the RET instruction allows a program to increment the stack pointer on a return to release parameters from the stack. The number of bytes released from the stack is determined by an optional argument (n) to the RET instruction. See "RET—Return from Procedure" in Chapter 4, "Instruction Set Reference, M-U,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for a detailed description of the RET instruction.

6.4.1 Near CALL and RET Operation

When executing a near call, the processor does the following (see Figure 6-2):

1. Pushes the current value of the EIP register on the stack.
   - If shadow stack is enabled and the displacement value is not 0, pushes the current value of the EIP register on the shadow stack.
2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:
1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
   If shadow stack is enabled, pops the top-of-stack (the return instruction pointer) value from the shadow stack and if it’s not the same as the return instruction pointer popped from the stack, then the processor causes a control protection exception with error code NEAR-RET (#CP(NEAR-RET)).
2. If the RET instruction has an optional \( n \) argument, increments the stack pointer by the number of bytes specified with the \( n \) operand to release parameters from the stack.
3. Resumes execution of the calling procedure.

### 6.4.2 Far CALL and RET Operation

When executing a far call, the processor performs these actions (see Figure 6-2):
1. Pushes the current value of the CS register on the stack.
   If shadow stack is enabled:
   a. Temporarily saves the current value of the SSP register internally and aligns the SSP to the next 8 byte boundary.
   b. Pushes the current value of the CS register on the shadow stack.
   c. Pushes the current value of LIP (CS.base + EIP) on the shadow stack.
   d. Pushes the internally saved value of the SSP register on the shadow stack.
2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

When executing a far return, the processor does the following:
1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
   If shadow stack is enabled:
   a. Causes a control protection exception (#CP(FAR-RET/IRET)) if the SSP is not aligned to 8 bytes.
   b.Compares the values on the shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack, and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.
   c. Pops the top-of-stack value (the SSP of the procedure being returned to) from shadow stack into the SSP register.
3. If the RET instruction has an optional \( n \) argument, increments the stack pointer by the number of bytes specified with the \( n \) operand to release parameters from the stack.
4. Resumes execution of the calling procedure.
Figure 6-2. Stack on Near and Far Calls

Note: On a near or far return, parameters are released from the stack based on the optional \( n \) operand in the RET \( n \) instruction.

Figure 6-3. Shadow Stack on Near and Far Calls

Note: There are no parameters on the shadow stack. RET and RET \( n \) operate identically on the shadow stack.
6.4.3 Parameter Passing

Parameters can be passed between procedures in any of three ways: through general-purpose registers, in an argument list, or on the stack.

6.4.3.1 Passing Parameters Through the General-Purpose Registers

The processor does not save the state of the general-purpose registers on procedure calls. A calling procedure can thus pass up to six parameters to the called procedure by copying the parameters into any of these registers (except the ESP and EBP registers) prior to executing the CALL instruction. The called procedure can likewise pass parameters back to the calling procedure through general-purpose registers.

6.4.3.2 Passing Parameters on the Stack

To pass a large number of parameters to the called procedure, the parameters can be placed on the stack, in the stack frame for the calling procedure. Here, it is useful to use the stack-frame base pointer (in the EBP register) to make a frame boundary for easy access to the parameters.

The stack can also be used to pass parameters back from the called procedure to the calling procedure.

6.4.3.3 Passing Parameters in an Argument List

An alternate method of passing a larger number of parameters (or a data structure) to the called procedure is to place the parameters in an argument list in one of the data segments in memory. A pointer to the argument list can then be passed to the called procedure through a general-purpose register or the stack. Parameters can also be passed back to the calling procedure in this same manner.

6.4.4 Saving Procedure State Information

The processor does not save the contents of the general-purpose registers, segment registers, or the EFLAGS register on a procedure call. A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return. These values can be saved on the stack or in memory in one of the data segments.

The PUSHA and POPA instructions facilitate saving and restoring the contents of the general-purpose registers. PUSHA pushes the values in all the general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (the value prior to executing the PUSHA instruction), EBP, ESI, and EDI. The POPA instruction pops all the register values saved with a PUSHA instruction (except the ESP value) from the stack to their respective registers.

If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former values before executing a return to the calling procedure.

If a calling procedure needs to maintain the state of the EFLAGS register, it can save and restore all or part of the register using the PUSHF/PUSHFD and POPF/POPFD instructions. The PUSHF instruction pushes the lower word of the EFLAGS register on the stack, while the PUSHFD instruction pushes the entire register. The POPF instruction pops a word from the stack into the lower word of the EFLAGS register, while the POPFD instruction pops a double word from the stack into the register.

6.4.5 Calls to Other Privilege Levels

The IA-32 architecture’s protection mechanism recognizes four privilege levels, numbered from 0 to 3, where a greater number mean less privilege. The reason to use privilege levels is to improve the reliability of operating systems. For example, Figure 6-4 shows how privilege levels can be interpreted as rings of protection.
In this example, the highest privilege level 0 (at the center of the diagram) is used for segments that contain the most critical code modules in the system, usually the kernel of an operating system. The outer rings (with progressively lower privileges) are used for segments that contain code modules for less critical software.

Code modules in lower privilege segments can only access modules operating at higher privilege segments by means of a tightly controlled and protected interface called a gate. Attempts to access higher privilege segments without going through a protection gate and without having sufficient access rights causes a general-protection exception (#GP) to be generated.

If an operating system or executive uses this multilevel protection mechanism, a call to a procedure that is in a more privileged protection level than the calling procedure is handled in a similar manner as a far call (see Section 6.4.2, "Far CALL and RET Operation"). The differences are as follows:

- The segment selector provided in the CALL instruction references a special data structure called a call gate descriptor. Among other things, the call gate descriptor provides the following:
  - Access rights information.
  - The segment selector for the code segment of the called procedure.
  - An offset into the code segment (that is, the instruction pointer for the called procedure).
- The processor switches to a new stack to execute the called procedure. Each privilege level has its own stack. The segment selector and stack pointer for the privilege level 3 stack are stored in the SS and ESP registers, respectively, and are automatically saved when a call to a more privileged level occurs. The segment selectors and stack pointers for the privilege level 2, 1, and 0 stacks are stored in a system segment called the task state segment (TSS).

The use of a call gate and the TSS during a stack switch are transparent to the calling procedure, except when a general-protection exception is raised.

### 6.4.6 CALL and RET Operation Between Privilege Levels

When making a call to a more privileged protection level, the processor does the following (see Figure 6-5):

1. Performs an access rights check (privilege check).
2. Temporarily saves (internally) the current contents of the SS, ESP, CS, and EIP registers.
Figure 6-5. Stack Switch on a Call to a Different Privilege Level

Note: On a return, parameters are released on both stacks based on the optional n operand in the RET n instruction.
3. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.

4. Pushes the temporarily saved SS and ESP values for the calling procedure’s stack onto the new stack.

5. Copies the parameters from the calling procedure’s stack to the new stack. A value in the call gate descriptor determines how many parameters to copy to the new stack.

6. Pushes the temporarily saved CS and EIP values for the calling procedure to the new stack.

   If shadow stack is enabled at the privilege level of the calling procedure, then the processor temporarily saves the SSP of the calling procedure internally. If the calling procedure is at privilege level 3, the SSP of the calling procedure is also saved into the IA32_PL3_SSP MSR.

---

Figure 6-6. Shadow Stack Switch on a Call to a Different Privilege Level
If shadow stack is enabled at the privilege level of the called procedure, then the SSP for the called procedure is obtained from one of the MSRs listed below, depending on the target privilege level. The SSP obtained is then verified to ensure it points to a valid supervisor shadow stack that is not currently active by verifying a supervisor shadow stack token at the address pointed to by the SSP. The operations performed to verify and acquire the supervisor shadow stack token by making it busy are as described in Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

- IA32 PL2 SSP if transitioning to ring 2.
- IA32 PL1 SSP if transitioning to ring 1.
- IA32 PL0 SSP if transitioning to ring 0.

If shadow stack is enabled at the privilege level of the called procedure and the calling procedure was not at privilege level 3, then the processor pushes the temporarily saved CS, LIP (CS.base + EIP), and SSP of the calling procedure to the new shadow stack.

7. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively.

8. Begins execution of the called procedure at the new privilege level.

When executing a return from the privileged procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the call.
   - If shadow stack is enabled at the current privilege level:
     - Causes a control protection exception (#CP(FAR-RET/IRET)) if SSP is not aligned to 8 bytes.
     - If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode):
       - Compares the values on shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.
       - Temporarily saves the top-of-stack value (the SSP of the procedure being returned to) internally.
     - If a busy supervisor shadow stack token is present at address SSP+24, then marks the token free using operations described in Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
     - If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode), restores the SSP register from the internally saved value.
     - If the privilege level of the procedure being returned to is 3 (returning to user mode) and shadow stack is enabled at privilege level 3, then restores the SSP register with value of IA32 PL3 SSP MSR.
3. If the RET instruction has an optional n argument, increments the stack pointer by the number of bytes specified with the n operand to release parameters from the stack. If the call gate descriptor specifies that one or more parameters be copied from one stack to the other, a RET n instruction must be used to release the parameters from both stacks. Here, the n operand specifies the number of bytes occupied on each stack by the parameters. On a return, the processor increments ESP by n for each stack to step over (effectively remove) these parameters from the stacks.
4. Restores the SS and ESP registers to their values prior to the call, which causes a switch back to the stack of the calling procedure.
5. If the RET instruction has an optional n argument, increments the stack pointer by the number of bytes specified with the n operand to release parameters from the stack (see explanation in step 3).
6. Resumes execution of the calling procedure.

See Chapter 5, ”Protection,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for detailed information on calls to privileged levels and the call gate descriptor.

1. If any of these pushes leads to an exception or a VM exit, the supervisor shadow-stack token remains busy.
6.4.7 Branch Functions in 64-Bit Mode

The 64-bit extensions expand branching mechanisms to accommodate branches in 64-bit linear-address space. These are:

- Near-branch semantics are redefined in 64-bit mode.
- In 64-bit mode and compatibility mode, 64-bit call-gate descriptors for far calls are available.

In 64-bit mode, the operand size for all near branches (CALL, RET, JCC, JCXZ, JMP, and LOOP) is forced to 64 bits. These instructions update the 64-bit RIP without the need for a REX operand-size prefix.

The following aspects of near branches are controlled by the effective operand size:

- Truncation of the size of the instruction pointer.
- Size of a stack pop or push, due to a CALL or RET.
- Size of a stack-pointer increment or decrement, due to a CALL or RET.
- Indirect-branch operand size.

In 64-bit mode, all of the above actions are forced to 64 bits regardless of operand size prefixes (operand size prefixes are silently ignored). However, the displacement field for relative branches is still limited to 32 bits and the address size for near branches is not forced in 64-bit mode.

Address sizes affect the size of RCX used for JCXZ and LOOP; they also impact the address calculation for memory indirect branches. Such addresses are 64 bits by default; but they can be overridden to 32 bits by an address size prefix.

Software typically uses far branches to change privilege levels. The legacy IA-32 architecture provides the call-gate mechanism to allow software to branch from one privilege level to another, although call gates can also be used for branches that do not change privilege levels. When call gates are used, the selector portion of the direct or indirect pointer references a gate descriptor (the offset in the instruction is ignored). The offset to the destination’s code segment is taken from the call-gate descriptor.

64-bit mode redefines the type value of a 32-bit call-gate descriptor type to a 64-bit call gate descriptor and expands the size of the 64-bit descriptor to hold a 64-bit offset. The 64-bit mode call-gate descriptor allows far branches that reference any location in the supported linear-address space. These call gates also hold the target code selector (CS), allowing changes to privilege level and default size as a result of the gate transition.

Because immediates are generally specified up to 32 bits, the only way to specify a full 64-bit absolute RIP in 64-bit mode is with an indirect branch. For this reason, direct far branches are eliminated from the instruction set in 64-bit mode.

64-bit mode also expands the semantics of the SYSENTER and SYSEXIT instructions so that the instructions operate within a 64-bit memory space. The mode also introduces two new instructions: SYSCALL and SYSRET (which are valid only in 64-bit mode). For details, see “SYSENTER—Fast System Call,” “SYSEXIT—Fast Return from Fast System Call,” “SYSCALL—Fast System Call,” and “SYSRET—Return From Fast System Call” in Chapter 4, “Instruction Set Reference, M-U,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

6.5 INTERRUPTS AND EXCEPTIONS

The processor provides two mechanisms for interrupting program execution, interrupts, and exceptions:

- An **interrupt** is an asynchronous event that is typically triggered by an I/O device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction. The IA-32 architecture specifies three classes of exceptions: faults, traps, and aborts.

The processor responds to interrupts and exceptions in essentially the same way. When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition. The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT). When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task.
The operating system, executive, and/or device drivers normally handle interrupts and exceptions independently from application programs or tasks. Application programs can, however, access the interrupt and exception handlers incorporated in an operating system or executive through assembly-language calls. The remainder of this section gives a brief overview of the processor’s interrupt and exception handling mechanism. See Chapter 6, “Interrupt and Exception Handling,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a description of this mechanism.

The IA-32 Architecture defines 18 predefined interrupts and exceptions and 224 user-defined interrupts, which are associated with entries in the IDT. Each interrupt and exception in the IDT is identified with a number, called a vector. Table 6-1 lists the interrupts and exceptions with entries in the IDT and their respective vectors. Vectors 0 through 8, 10 through 14, and 16 through 19 are the predefined interrupts and exceptions; vectors 32 through 255 are for software-defined interrupts, which are for either software interrupts or maskable hardware interrupts.

Note that the processor defines several additional interrupts that do not point to entries in the IDT; the most notable of these interrupts is the SMI interrupt. See Chapter 6, “Interrupt and Exception Handling,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information about the interrupts and exceptions.

When the processor detects an interrupt or exception, it does one of the following things:

- Executes an implicit call to a handler procedure.
- Executes an implicit call to a handler task.

### 6.5.1 Call and Return Operation for Interrupt or Exception Handling Procedures

A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see Section 6.4.6, "CALL and RET Operation Between Privilege Levels"). Here, the vector references one of two kinds of gates in the IDT: an interrupt gate or a trap gate. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information
- The segment selector for the code segment that contains the handler procedure
- An offset into the code segment to the first instruction of the handler procedure

The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

### Table 6-1. Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Vector</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and DIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td>#BP</td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#ZF</td>
<td>Breakpoint</td>
<td>INT3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (Undefined Opcode)</td>
<td>UD instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.¹</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
</tbody>
</table>

¹ Floating-point instruction.¹
If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler’s privilege level.

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 6-7):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
   - If shadow stack is enabled:
     a. Temporarily saves the current value of the SSP register internally.
     b. Pushes the current value of the CS register on the shadow stack.
     c. Pushes the current value of LIP (CS.base + EIP) on the shadow stack.
     d. Pushes the temporarily saved SSP value on the shadow stack.
   - Pushes an error code (if appropriate) on the stack.
2. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
3. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
4. Begins execution of the handler procedure.

### Table 6-1. Exceptions and Interrupts (Contd.)

<table>
<thead>
<tr>
<th>Vector</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.²</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.³</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction⁴</td>
</tr>
<tr>
<td>20</td>
<td>#VE</td>
<td>Virtualization Exception</td>
<td>EPT violations⁵</td>
</tr>
<tr>
<td>21</td>
<td>#CP</td>
<td>Control Protection Exception</td>
<td>The RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at the target of an indirect call or jump.</td>
</tr>
<tr>
<td>22-31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT n instruction.</td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:
1. IA-32 processors after the Intel386 processor do not generate this exception.
2. This exception was introduced in the Intel486 processor.
3. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
4. This exception was introduced in the Pentium III processor.
5. This exception can occur only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.
Figure 6-7. Stack Usage on Transfers to Interrupt and Exception Handling Routines
Figure 6-8. Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines
If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.
2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure’s stack onto the new stack.
   - If shadow stack is enabled at the privilege level of the interrupted procedure, then the processor temporarily saves the SSP of the interrupted procedure internally. If the interrupted procedure is at privilege level 3, the SSP of the interrupted procedure is also saved into the IA32_PL3_SSP MSR.
   - If shadow stack is enabled at the privilege level being called, then the SSP for the called privilege level is obtained from one of the MSRs listed below, depending on the target privilege level. The SSP obtained is then verified to ensure it points to a valid supervisor shadow stack that is not currently active by verifying a supervisor shadow stack token at the address pointed to by the SSP. The operations performed to verify and acquire the supervisor shadow stack token by making it busy are as described in Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
     - IA32_PL2_SSP if transitioning to ring 2.
     - IA32_PL1_SSP if transitioning to ring 1.
     - IA32_PL0_SSP if transitioning to ring 0.
   - If shadow stack is enabled at the privilege level being called and the interrupted procedure was not at privilege level 3, then the processor pushes the temporarily saved CS, LIP (CS.base + EIP), and SSP of the interrupted procedure to the new shadow stack.
4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure. When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.
   - If shadow stack is enabled:
     a. Compares the values on the shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack, and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.
     b. Pops the top-of-stack value (the SSP prior to the interrupt or exception) from the shadow stack into the SSP register.
2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately.
4. Resumes execution of the interrupted procedure.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.

1. If any of these pushes leads to an exception or a VM exit, the supervisor shadow-stack token remains busy.
If shadow stack is enabled at the current privilege level:

- If SSP is not aligned to 8 bytes, then causes a control protection exception (#CP(FAR-RET/IRET)).
- If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode):
  - Compares the values on the shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack, and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.
  - Temporarily saves the top-of-stack value (the SSP of the procedure being returned to) internally.
- If a busy supervisor shadow stack token is present at address SSP+24, then marks the token free using operations described in Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
- If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode), restores the SSP register from the internally saved value.
- If the privilege level of the procedure being returned to is 3 (returning to user mode) and shadow stack is enabled at privilege level 3, then restores the SSP register with the value of the IA32_PL3_SSP MSR.

4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.

5. Resumes execution of the interrupted procedure.

6.5.2 Calls to Interrupt or Exception Handler Tasks

Interrupt and exception handler routines can also be executed in a separate task. Here, an interrupt or exception causes a task switch to a handler task. The handler task is given its own address space and (optionally) can execute at a higher protection level than application programs or tasks.

The switch to the handler task is accomplished with an implicit task call that references a **task gate descriptor**. The task gate provides access to the address space for the handler task. As part of the task switch, the processor saves complete state information for the interrupted program or task. Upon returning from the handler task, the state of the interrupted program or task is restored and execution continues. See Chapter 6, “Interrupt and Exception Handling,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on handling interrupts and exceptions through handler tasks.

6.5.3 Interrupt and Exception Handling in Real-Address Mode

When operating in real-address mode, the processor responds to an interrupt or exception with an implicit far call to an interrupt or exception handler. The processor uses the interrupt or exception vector as an index into an interrupt table. The interrupt table contains instruction pointers to the interrupt and exception handler procedures.

The processor saves the state of the EFLAGS register, the EIP register, the CS register, and an optional error code on the stack before switching to the handler procedure.

A return from the interrupt or exception handler is carried out with the IRET instruction.

See Chapter 21, “8086 Emulation,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for more information on handling interrupts and exceptions in real-address mode.

6.5.4 **INT n**, INTO, INT3, INT1, and BOUND Instructions

The INT n, INTO, INT3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT n instruction (opcode CD) uses a vector as an argument, which allows a program to call any interrupt handler.

The INTO instruction (opcode CE) explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception. An overflow exception can only be raised explicitly in either of the following ways:
• Execute the INTO instruction.
• Test the OF flag and execute the INT n instruction with an argument of 4 (the vector of the overflow exception) if the flag is set.

Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT3 instruction (opcode CC) explicitly calls the breakpoint exception (#BP) handler. Similarly, the INT1 instruction (opcode F1) explicitly calls the debug exception (#DB) handler.¹

The BOUND instruction explicitly calls the BOUND-range exceeded exception (#BR) handler if an operand is found to be not within predefined boundaries in memory. This instruction is provided for checking references to arrays and other data structures. Like the overflow exception, the BOUND-range exceeded exception can only be raised explicitly with the BOUND instruction or the INT n instruction with an argument of 5 (the vector of the bounds-check exception). The processor does not implicitly perform bounds checks and raise the BOUND-range exceeded exception.

### 6.5.5 Handling Floating-Point Exceptions

When operating on individual or packed floating-point values, the IA-32 architecture supports a set of six floating-point exceptions. These exceptions can be generated during operations performed by the x87 FPU instructions or by SSE/SSE2/SSE3 instructions. When an x87 FPU instruction (including the FISTTP instruction in SSE3) generates one or more of these exceptions, it in turn generates floating-point error exception (#MF); when an SSE/SSE2/SSE3 instruction generates a floating-point exception, it in turn generates SIMD floating-point exception (#XM).

See the following sections for further descriptions of the floating-point exceptions, how they are generated, and how they are handled:

- Section 4.9.1, “Floating-Point Exception Conditions,” and Section 4.9.3, “Typical Actions of a Floating-Point Exception Handler.”
- Section 8.4, “x87 FPU Floating-Point Exception Handling,” and Section 8.5, “x87 FPU Floating-Point Exception Conditions.”
- Section 11.5.1, “SIMD Floating-Point Exceptions.”
- Interrupt Behavior.

### 6.5.6 Interrupt and Exception Behavior in 64-Bit Mode

64-bit extensions expand the legacy IA-32 interrupt-processing and exception-processing mechanism to allow support for 64-bit operating systems and applications. Changes include:

- All interrupt handlers pointed to by the IDT are 64-bit code (does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits. The processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy environments, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism and a new interrupt shadow stack-switch mechanism.
- The alignment of interrupt stack frame is different.

---

¹ Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.
PROCEDURE CALLS, INTERRUPTS, AND EXCEPTIONS

6.6 PROCEDURE CALLS FOR BLOCK-STRUCTURED LANGUAGES

The IA-32 architecture supports an alternate method of performing procedure calls with the ENTER (enter procedure) and LEAVE (leave procedure) instructions. These instructions automatically create and release, respectively, stack frames for called procedures. The stack frames have predefined spaces for local variables and the necessary pointers to allow coherent returns from called procedures. They also allow scope rules to be implemented so that procedures can access their own local variables and some number of other variables located in other stack frames.

ENTER and LEAVE offer two benefits:

- They provide machine-language support for implementing block-structured languages, such as C and Pascal.
- They simplify procedure entry and exit in compiler-generated code.

6.6.1 ENTER Instruction

The ENTER instruction creates a stack frame compatible with the scope rules typically used in block-structured languages. In block-structured languages, the scope of a procedure is the set of variables to which it has access. The rules for scope vary among languages. They may be based on the nesting of procedures, the division of the program into separately compiled files, or some other modularization scheme.

ENTER has two operands. The first specifies the number of bytes to be reserved on the stack for dynamic storage for the procedure being called. Dynamic storage is the memory allocated for variables created when the procedure is called, also known as automatic variables. The second parameter is the lexical nesting level (from 0 to 31) of the procedure. The nesting level is the depth of a procedure in a hierarchy of procedure calls. The lexical level is unrelated to either the protection privilege level or to the I/O privilege level of the currently running program or task.

ENTER, in the following example, allocates 2 Kbytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame for this procedure:

```
ENTER 2048,3
```

The lexical nesting level determines the number of stack frame pointers to copy into the new stack frame from the preceding frame. A stack frame pointer is a doubleword used to access the variables of a procedure. The set of stack frame pointers used by a procedure to access the variables of other procedures is called the display. The first doubleword in the display is a pointer to the previous stack frame. This pointer is used by a LEAVE instruction to undo the effect of an ENTER instruction by discarding the current stack frame.

After the ENTER instruction creates the display for a procedure, it allocates the dynamic local variables for the procedure by decrementing the contents of the ESP register by the number of bytes specified in the first parameter. This new value in the ESP register serves as the initial top-of-stack for all PUSH and POP operations within the procedure.

To allow a procedure to address its display, the ENTER instruction leaves the EBP register pointing to the first doubleword in the display. Because stacks grow down, this is actually the doubleword with the highest address in the display. Data manipulation instructions that specify the EBP register as a base register automatically address locations within the stack segment instead of the data segment.

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. The non-nested form pushes the contents of the EBP register on the stack, copies the contents of the ESP register into the EBP register, and subtracts the first operand from the contents of the ESP register to allocate dynamic storage. The non-nested form differs from the nested form in that no stack frame pointers are copied. The nested form of the ENTER instruction occurs when the second parameter (lexical level) is not zero.

The following pseudo code shows the formal definition of the ENTER instruction. STORAGE is the number of bytes of dynamic storage to allocate for local variables, and LEVEL is the lexical nesting level.
The main procedure (in which all other procedures are nested) operates at the highest lexical level, level 1. The first procedure it calls operates at the next deeper lexical level, level 2. A level 2 procedure can access the variables of the main program, which are at fixed locations specified by the compiler. In the case of level 1, the ENTER instruction allocates only the requested dynamic storage on the stack because there is no previous display to copy.

A procedure that calls another procedure at a lower lexical level gives the called procedure access to the variables of the caller. The ENTER instruction provides this access by placing a pointer to the calling procedure’s stack frame in the display.

A procedure that calls another procedure at the same lexical level should not give access to its variables. In this case, the ENTER instruction copies only that part of the display from the calling procedure which refers to previously nested procedures operating at higher lexical levels. The new stack frame does not include the pointer for addressing the calling procedure’s stack frame.

The ENTER instruction treats a re-entrant procedure as a call to a procedure at the same lexical level. In this case, each succeeding iteration of the re-entrant procedure can address only its own variables and the variables of the procedures within which it is nested. A re-entrant procedure always can address its own variables; it does not require pointers to the stack frames of previous iterations.

By copying only the stack frame pointers of procedures at higher lexical levels, the ENTER instruction makes certain that procedures access only those variables of higher lexical levels, not those at parallel lexical levels (see Figure 6-9).

---

**Figure 6-9. Nested Procedures**

Block-structured languages can use the lexical levels defined by ENTER to control access to the variables of nested procedures. In Figure 6-9, for example, if procedure A calls procedure B which, in turn, calls procedure C, then procedure C will have access to the variables of the MAIN procedure and procedure A, but not those of procedure B because they are at the same lexical level. The following definition describes the access to variables for the nested procedures in Figure 6-9.

1. MAIN has variables at fixed locations.
2. Procedure A can access only the variables of MAIN.
3. Procedure B can access only the variables of procedure A and MAIN. Procedure B cannot access the variables of procedure C or procedure D.

4. Procedure C can access only the variables of procedure A and MAIN. Procedure C cannot access the variables of procedure B or procedure D.

5. Procedure D can access the variables of procedure C, procedure A, and MAIN. Procedure D cannot access the variables of procedure B.

In Figure 6-10, an ENTER instruction at the beginning of the MAIN procedure creates three doublewords of dynamic storage for MAIN, but copies no pointers from other stack frames. The first doubleword in the display holds a copy of the last value in the EBP register before the ENTER instruction was executed. The second doubleword holds a copy of the contents of the EBP register following the ENTER instruction. After the instruction is executed, the EBP register points to the first doubleword pushed on the stack, and the ESP register points to the last doubleword in the stack frame.

When MAIN calls procedure A, the ENTER instruction creates a new display (see Figure 6-11). The first doubleword is the last value held in MAIN’s EBP register. The second doubleword is a pointer to MAIN’s stack frame which is copied from the second doubleword in MAIN’s display. This happens to be another copy of the last value held in MAIN’s EBP register. Procedure A can access variables in MAIN because MAIN is at level 1.

Therefore the base address for the dynamic storage used in MAIN is the current address in the EBP register, plus four bytes to account for the saved contents of MAIN’s EBP register. All dynamic variables for MAIN are at fixed, positive offsets from this value.
When procedure A calls procedure B, the ENTER instruction creates a new display (see Figure 6-12). The first doubleword holds a copy of the last value in procedure A’s EBP register. The second and third doublewords are copies of the two stack frame pointers in procedure A’s display. Procedure B can access variables in procedure A and MAIN by using the stack frame pointers in its display.

When procedure B calls procedure C, the ENTER instruction creates a new display for procedure C (see Figure 6-13). The first doubleword holds a copy of the last value in procedure B’s EBP register. This is used by the LEAVE instruction to restore procedure B’s stack frame. The second and third doublewords are copies of the two stack frame pointers in procedure A’s display. If procedure C were at the next deeper lexical level from procedure B, a fourth doubleword would be copied, which would be the stack frame pointer to procedure B’s local variables.

Note that procedure B and procedure C are at the same level, so procedure C is not intended to access procedure B’s variables. This does not mean that procedure C is completely isolated from procedure B; procedure C is called by procedure B, so the pointer to the returning stack frame is a pointer to procedure B’s stack frame. In addition, procedure B can pass parameters to procedure C either on the stack or through variables global to both procedures (that is, variables in the scope of both procedures).

![Figure 6-12. Stack Frame After Entering Procedure B](image-url)
6.6.2 LEAVE Instruction

The LEAVE instruction, which does not have any operands, reverses the action of the previous ENTER instruction. The LEAVE instruction copies the contents of the EBP register into the ESP register to release all stack space allocated to the procedure. Then it restores the old value of the EBP register from the stack. This simultaneously restores the ESP register to its original value. A subsequent RET instruction then can remove any arguments and the return address pushed on the stack by the calling program for use by the procedure.
General-purpose (GP) instructions are a subset of the IA-32 instructions that represent the fundamental instruction set for the Intel IA-32 processors. These instructions were introduced into the IA-32 architecture with the first IA-32 processors (the Intel 8086 and 8088). Additional instructions were added to the general-purpose instruction set in subsequent families of IA-32 processors (the Intel 286, Intel386, Intel486, Pentium, Pentium Pro, and Pentium II processors).

Intel 64 architecture further extends the capability of most general-purpose instructions so that they are able to handle 64-bit data in 64-bit mode. A small number of general-purpose instructions (still supported in non-64-bit modes) are not supported in 64-bit mode.

General-purpose instructions perform basic data movement, memory addressing, arithmetic and logical, program flow control, input/output, and string operations on a set of integer, pointer, and BCD data types. This chapter provides an overview of the general-purpose instructions. See the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D, for detailed descriptions of individual instructions.

7.1 PROGRAMMING ENVIRONMENT FOR GP INSTRUCTIONS

The programming environment for the general-purpose instructions consists of the set of registers and address space. The environment includes the following items:

- **General-purpose registers** — Eight 32-bit general-purpose registers (see Section 3.4.1, "General-Purpose Registers") are used in non-64-bit modes to address operands in memory. These registers are referenced by the names EAX, EBX, ECX, EDX, EBP, ESI EDI, and ESP.

- **Segment registers** — The six 16-bit segment registers contain segment pointers for use in accessing memory (see Section 3.4.2, "Segment Registers"). These registers are referenced by the names CS, DS, SS, ES, FS, and GS.

- **EFLAGS register** — This 32-bit register (see Section 3.4.3, "EFLAGS Register") is used to provide status and control for basic arithmetic, compare, and system operations.

- **EIP register** — This 32-bit register contains the current instruction pointer (see Section 3.5, "Instruction Pointer").

General-purpose instructions operate on the following data types. The width of valid data types is dependent on processor mode (see Chapter 4):

- Bytes, words, doublewords.
- Signed and unsigned byte, word, doubleword integers.
- Near and far pointers.
- Bit fields.
- BCD integers.

7.2 PROGRAMMING ENVIRONMENT FOR GP INSTRUCTIONS IN 64-BIT MODE

The programming environment for the general-purpose instructions in 64-bit mode is similar to that described in Section 7.1.

- **General-purpose registers** — In 64-bit mode, sixteen general-purpose registers available. These include the eight GPRs described in Section 7.1 and eight new GPRs (R8D-R15D). R8D-R15D are available by using a REX prefix. All sixteen GPRs can be promoted to 64 bits. The 64-bit registers are referenced as RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8-R15 (see Section 3.4.1.1, "General-Purpose Registers in 64-Bit Mode"). Promotion to 64-bit operand requires REX prefix encodings.
7.3 SUMMARY OF GP INSTRUCTIONS

General purpose instructions are divided into the following subgroups:

- Data transfer.
- Binary arithmetic.
- Decimal arithmetic.
- Logical.
- Shift and rotate.
- Bit and byte.
- Control transfer.
- String.
- I/O.
- Enter and Leave.
- Flag control.
- Segment register.
- Miscellaneous.

Each sub-group of general-purpose instructions is discussed in the context of non-64-bit mode operation first. Changes in 64-bit mode beyond those affected by the use of the REX prefixes are discussed in separate sub-sections within each subgroup. For a simple list of general-purpose instructions by subgroup, see Chapter 5.

7.3.1 Data Transfer Instructions

The data transfer instructions move bytes, words, doublewords, or quadwords both between memory and the processor's registers and between registers. For the purpose of this discussion, these instructions are divided into subordinate subgroups that provide for:

- General data movement.

---

Segment registers — In 64-bit mode, segmentation is available but it is set up uniquely (see Section 3.4.2.1, “Segment Registers in 64-Bit Mode”).

Flags and Status register — When the processor is running in 64-bit mode, EFLAGS becomes the 64-bit RFLAGS register (see Section 3.4.3, “EFLAGS Register”).

Instruction Pointer register — In 64-bit mode, the EIP register becomes the 64-bit RIP register (see Section 3.5.1, “Instruction Pointer in 64-Bit Mode”).

General-purpose instructions operate on the following data types in 64-bit mode. The width of valid data types is dependent on default operand size, address size, or a prefix that overrides the default size:

- Bytes, words, doublewords, quadwords.
- Signed and unsigned byte, word, doubleword, quadword integers.
- Near and far pointers.
- Bit fields.

See also:

- Chapter 3, “Basic Execution Environment,” for more information about IA-32e modes.
- Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, for a complete listing of all instructions. This information documents the behavior of individual instructions in the 64-bit mode context.
• Exchange.
• Stack manipulation.
• Type conversion.

7.3.1.1 General Data Movement Instructions

Move instructions — The MOV (move) and CMOVcc (conditional move) instructions transfer data between memory and registers or between registers.

The MOV instruction performs basic load and store operations between memory and the processor’s registers and data movement operations between registers. It handles data transfers along the paths listed in Table 7-1. (See “MOV—Move to/from Control Registers” and “MOV—Move to/from Debug Registers” in Chapter 4, “Instruction Set Reference, M-U,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for information on moving data to and from the control and debug registers.)

The MOV instruction cannot move data from one memory location to another or from one segment register to another segment register. Memory-to-memory moves are performed with the MOVS (string move) instruction (see Section 7.3.9, "String Operations").

Conditional move instructions — The CMOVcc instructions are a group of instructions that check the state of the status flags in the EFLAGS register and perform a move operation if the flags are in a specified state. These instructions can be used to move a 16-bit or 32-bit value from memory to a general-purpose register or from one general-purpose register to another. The flag state being tested is specified with a condition code (cc) associated with the instruction. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

Table 7-1. Move Instruction Operations

<table>
<thead>
<tr>
<th>Type of Data Movement</th>
<th>Source → Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>From memory to a register</td>
<td>Memory location → General-purpose register</td>
</tr>
<tr>
<td>From memory to a register</td>
<td>Memory location → Segment register</td>
</tr>
<tr>
<td>From a register to memory</td>
<td>General-purpose register → Memory location</td>
</tr>
<tr>
<td>From a register to memory</td>
<td>Segment register → Memory location</td>
</tr>
<tr>
<td>Between registers</td>
<td>General-purpose register → General-purpose register</td>
</tr>
<tr>
<td>Between registers</td>
<td>General-purpose register → Segment register</td>
</tr>
<tr>
<td>Between registers</td>
<td>Segment register → General-purpose register</td>
</tr>
<tr>
<td>Between registers</td>
<td>General-purpose register → Control register</td>
</tr>
<tr>
<td>Between registers</td>
<td>Control register → General-purpose register</td>
</tr>
<tr>
<td>Between registers</td>
<td>General-purpose register → Debug register</td>
</tr>
<tr>
<td>Between registers</td>
<td>Debug register → General-purpose register</td>
</tr>
<tr>
<td>Immediate data to a register</td>
<td>Immediate → General-purpose register</td>
</tr>
<tr>
<td>Immediate data to memory</td>
<td>Immediate → Memory location</td>
</tr>
</tbody>
</table>

Table 7-2 shows mnemonics for CMOVcc instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letters “CMOV” to form the mnemonics for CMOVcc instructions. The instructions listed in Table 7-2 as pairs (for example, CMOVA/CMOVNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

CMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF statements and the possibility of branch mispredictions by the processor.

These conditional move instructions are supported in the P6 family, Pentium 4, and Intel Xeon processors. Software can check if CMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction.
7.3.1.2 Exchange Instructions

The exchange instructions swap the contents of one or more operands and, in some cases, perform additional operations such as asserting the LOCK signal or modifying flags in the EFLAGS register.

The XCHG (exchange) instruction swaps the contents of two operands. This instruction takes the place of three MOV instructions and does not require a temporary location to save the contents of one operand location while the other is being loaded. When a memory operand is used with the XCHG instruction, the processor’s LOCK signal is automatically asserted. This instruction is thus useful for implementing semaphores or similar data structures for process synchronization. See "Bus Locking" in Chapter 9, “Multiple-Processor Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on bus locking.

The BSWAP (byte swap) instruction reverses the byte order in a 32-bit register operand. Bit positions 0 through 7 are exchanged with 24 through 31, and bit positions 8 through 15 are exchanged with 16 through 23. Executing this instruction twice in a row leaves the register with the same value as before. The BSWAP instruction is useful for converting between “big-endian” and “little-endian” data formats. This instruction also speeds execution of decimal arithmetic. (The XCHG instruction can be used to swap the bytes in a word.)

The XADD (exchange and add) instruction swaps two operands and then stores the sum of the two operands in the destination operand. The status flags in the EFLAGS register indicate the result of the addition. This instruction can be combined with the LOCK prefix (see “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A) in a multiprocessing system to allow multiple processors to execute one DO loop.

The CMPXCHG (compare and exchange) and CMPXCHG8B (compare and exchange 8 bytes) instructions are used to synchronize operations in systems that use multiple processors. The CMPXCHG instruction requires three operands: a source operand in a register, another source operand in the EAX register, and a destination operand. If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register). Otherwise, the original

### Table 7-2. Conditional Move Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Status Flag States</th>
<th>Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned Conditional Moves</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMOVA/CMOVNBE</td>
<td>(CF or ZF) = 0</td>
<td>Above/not below or equal</td>
</tr>
<tr>
<td>CMOVAE/CMOVNB</td>
<td>CF = 0</td>
<td>Above or equal/not below</td>
</tr>
<tr>
<td>CMOVNC</td>
<td>CF = 0</td>
<td>Not carry</td>
</tr>
<tr>
<td>CMOVVB/CMOVNAE</td>
<td>CF = 1</td>
<td>Below/not above or equal</td>
</tr>
<tr>
<td>CMOVC</td>
<td>CF = 1</td>
<td>Carry</td>
</tr>
<tr>
<td>CMOVBE/CMOVNA</td>
<td>(CF or ZF) = 1</td>
<td>Below or equal/not above</td>
</tr>
<tr>
<td>CMOVE/CMOVZ</td>
<td>ZF = 1</td>
<td>Equal/zero</td>
</tr>
<tr>
<td>CMOVNE/CMOVNZ</td>
<td>ZF = 0</td>
<td>Not equal/not zero</td>
</tr>
<tr>
<td>CMOVVP/CMOVPE</td>
<td>PF = 1</td>
<td>Parity/parity even</td>
</tr>
<tr>
<td>CMOVNP/CMOVPO</td>
<td>PF = 0</td>
<td>Not parity/parity odd</td>
</tr>
<tr>
<td><strong>Signed Conditional Moves</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMOVGE/CMOVNL</td>
<td>(SF xor OF) = 0</td>
<td>Greater or equal/not less</td>
</tr>
<tr>
<td>CMOVL/CMOVNGE</td>
<td>(SF xor OF) = 1</td>
<td>Less/not greater or equal</td>
</tr>
<tr>
<td>CMOVL/CMOVNG</td>
<td>((SF xor OF) or ZF) = 1</td>
<td>Less or equal/not greater</td>
</tr>
<tr>
<td>CMOVO</td>
<td>OF = 1</td>
<td>Overflow</td>
</tr>
<tr>
<td>CMOVNO</td>
<td>OF = 0</td>
<td>Not overflow</td>
</tr>
<tr>
<td>CMOVS</td>
<td>SF = 1</td>
<td>Sign (negative)</td>
</tr>
<tr>
<td>CMOVNS</td>
<td>SF = 0</td>
<td>Not sign (non-negative)</td>
</tr>
</tbody>
</table>
value of the destination operand is loaded in the EAX register. The status flags in the EFLAGS register reflect the result that would have been obtained by subtracting the destination operand from the value in the EAX register.

The CMPXCHG instruction is commonly used for testing and modifying semaphores. It checks to see if a semaphore is free. If the semaphore is free, it is marked allocated; otherwise it gets the ID of the current owner. This is all done in one uninterruptible operation. In a single-processor system, the CMPXCHG instruction eliminates the need to switch to protection level 0 (to disable interrupts) before executing multiple instructions to test and modify a semaphore.

For multiple processor systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically. (See “Locked Atomic Operations” in Chapter 9, "Multiple-Processor Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on atomic operations.)

The CMPXCHG8B instruction also requires three operands: a 64-bit value in EDX:EAX, a 64-bit value in ECX:EBX, and a destination operand in memory. The instruction compares the 64-bit value in the EDX:EAX registers with the destination operand. If they are equal, the 64-bit value in the ECX:EBX registers is stored in the destination operand. If the EDX:EAX registers and the destination are not equal, the destination is loaded in the EDX:EAX registers. The CMPXCHG8B instruction can be combined with the LOCK prefix to perform the operation atomically.

7.3.1.3  Exchange Instructions in 64-Bit Mode

The CMPXCHG16B instruction is available in 64-bit mode only. It is an extension of the functionality provided by CMPXCHG8B that operates on 128-bits of data.

7.3.1.4  Stack Manipulation Instructions

The PUSH, POP, PUSHA (push all registers), and POPA (pop all registers) instructions move data to and from the stack. The PUSH instruction decrements the stack pointer (contained in the ESP register), then copies the source operand to the top of stack (see Figure 7-1). It operates on memory operands, immediate operands, and register operands (including segment registers). The PUSH instruction is commonly used to place parameters on the stack before calling a procedure. It can also be used to reserve space on the stack for temporary variables.

![Figure 7-1. Operation of the PUSH Instruction](image)

The PUSHA instruction saves the contents of the eight general-purpose registers on the stack (see Figure 7-2). This instruction simplifies procedure calls by reducing the number of instructions required to save the contents of the general-purpose registers. The registers are pushed on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI.
The POP instruction copies the word or doubleword at the current top of stack (indicated by the ESP register) to the location specified with the destination operand. It then increments the ESP register to point to the new top of stack (see Figure 7-3). The destination operand may specify a general-purpose register, a segment register, or a memory location.

The POPA instruction reverses the effect of the PUSHA instruction. It pops the top eight words or doublewords from the top of the stack into the general-purpose registers, except for the ESP register (see Figure 7-4). If the operand-size attribute is 32, the doublewords on the stack are transferred to the registers in the following order: EDI, ESI, EBP, ignore doubleword, EBX, EDX, ECX, and EAX. The ESP register is restored by the action of popping the stack. If the operand-size attribute is 16, the words on the stack are transferred to the registers in the following order: DI, SI, BP, ignore word, BX, DX, CX, and AX.

Figure 7-2. Operation of the PUSHA Instruction

Figure 7-3. Operation of the POP Instruction
7.3.1.5 Stack Manipulation Instructions in 64-Bit Mode

In 64-bit mode, the stack pointer size is 64 bits and cannot be overridden by an instruction prefix. In implicit stack references, address-size overrides are ignored. Pushes and pops of 32-bit values on the stack are not possible in 64-bit mode. 16-bit pushes and pops are supported by using the 66H operand-size prefix. PUSHA, PUSHAD, POPA, and POPAD are not supported.

7.3.1.6 Type Conversion Instructions

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into quadwords. These instructions are especially useful for converting integers to larger integer formats, because they perform sign extension (see Figure 7-5).

Two kinds of type conversion instructions are provided: simple conversion and move and convert.

Simple conversion — The CBW (convert byte to word), CWDE (convert word to doubleword extended), CWD (convert word to doubleword), and CDQ (convert doubleword to quadword) instructions perform sign extension to double the size of the source operand.

The CBW instruction copies the sign (bit 7) of the byte in the AL register into every bit position of the upper byte of the AX register. The CWDE instruction copies the sign (bit 15) of the word in the AX register into every bit position of the high word of the EAX register.

The CWD instruction copies the sign (bit 15) of the word in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the doubleword in the EAX register into every bit position in the EDX register. The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.
Move with sign or zero extension — The MOVSX (move with sign extension) and MOVZX (move with zero extension) instructions move the source operand into a register then perform the sign extension.

The MOVSX instruction extends an 8-bit value to a 16-bit value or an 8-bit or 16-bit value to a 32-bit value by sign extending the source operand, as shown in Figure 7-5. The MOVZX instruction extends an 8-bit value to a 16-bit value or an 8-bit or 16-bit value to a 32-bit value by zero extending the source operand.

7.3.1.7 Type Conversion Instructions in 64-Bit Mode

The MOVXSD instruction operates on 64-bit data. It sign-extends a 32-bit value to 64 bits. This instruction is not encodable in non-64-bit modes.

7.3.2 Binary Arithmetic Instructions

Binary arithmetic instructions operate on 8-, 16-, and 32-bit numeric data encoded as signed or unsigned binary integers. The binary arithmetic instructions may also be used in algorithms that operate on decimal (BCD) values. For the purpose of this discussion, these instructions are divided into subordinate subgroups of instructions that:
- Add and subtract.
- Increment and decrement.
- Compare and change signs.
- Multiply and divide.

7.3.2.1 Addition and Subtraction Instructions

The ADD (add integers), ADC (add integers with carry), SUB (subtract integers), and SBB (subtract integers with borrow) instructions perform addition and subtraction operations on signed or unsigned integer operands.

The ADD instruction computes the sum of two integer operands.

The ADC instruction computes the sum of two integer operands, plus 1 if the CF flag is set. This instruction is used to propagate a carry when adding numbers in stages.

The SUB instruction computes the difference of two integer operands.

The SBB instruction computes the difference of two integer operands, minus 1 if the CF flag is set. This instruction is used to propagate a borrow when subtracting numbers in stages.

7.3.2.2 Increment and Decrement Instructions

The INC (increment) and DEC (decrement) instructions add 1 to or subtract 1 from an unsigned integer operand, respectively. A primary use of these instructions is for implementing counters.

7.3.2.3 Increment and Decrement Instructions in 64-Bit Mode

The INC and DEC instructions are supported in 64-bit mode. However, some forms of INC and DEC (the register operand being encoded using register extension field in the MOD R/M byte) are not encodable in 64-bit mode because the opcodes are treated as REX prefixes.

7.3.2.4 Comparison and Sign Change Instructions

The CMP (compare) instruction computes the difference between two integer operands and updates the OF, SF, ZF, AF, PF, and CF flags according to the result. The source operands are not modified, nor is the result saved. The CMP instruction is commonly used in conjunction with a Jcc (jump) or SETcc (byte set on condition) instruction, with the latter instructions performing an action based on the result of a CMP instruction.

The NEG (negate) instruction subtracts a signed integer operand from zero. The effect of the NEG instruction is to change the sign of a two's complement operand while keeping its magnitude.
7.3.2.5 Multiplication and Division Instructions

The processor provides two multiply instructions, MUL (unsigned multiply) and IMUL (signed multiply), and two divide instructions, DIV (unsigned divide) and IDIV (signed divide).

The MUL instruction multiplies two unsigned integer operands. The result is computed to twice the size of the source operands (for example, if word operands are being multiplied, the result is a doubleword).

The IMUL instruction multiplies two signed integer operands. The result is computed to twice the size of the source operands; however, in some cases the result is truncated to the size of the source operands (see “IMUL—Signed Multiply” in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

The DIV instruction divides one unsigned operand by another unsigned operand and returns a quotient and a remainder.

The IDIV instruction is identical to the DIV instruction, except that IDIV performs a signed division.

7.3.3 Decimal Arithmetic Instructions

Decimal arithmetic can be performed by combining the binary arithmetic instructions ADD, SUB, MUL, and DIV (discussed in Section 7.3.2, “Binary Arithmetic Instructions”) with the decimal arithmetic instructions. The decimal arithmetic instructions are provided to carry out the following operations:

- To adjust the results of a previous binary arithmetic operation to produce a valid BCD result.
- To adjust the operands of a subsequent binary arithmetic operation so that the operation will produce a valid BCD result.

These instructions operate on both packed and unpacked BCD values. For the purpose of this discussion, the decimal arithmetic instructions are divided into subordinate subgroups of instructions that provide:

- Packed BCD adjustments.
- Unpacked BCD adjustments.

7.3.3.1 Packed BCD Adjustment Instructions

The DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) instructions adjust the results of operations performed on packed BCD integers (see Section 4.7, “BCD and Packed BCD Integers”). Adding two packed BCD values requires two instructions: an ADD instruction followed by a DAA instruction. The ADD instruction adds (binary addition) the two values and stores the result in the AL register. The DAA instruction then adjusts the value in the AL register to obtain a valid, 2-digit, packed BCD value and sets the CF flag if a decimal carry occurred as the result of the addition.

Likewise, subtracting one packed BCD value from another requires a SUB instruction followed by a DAS instruction. The SUB instruction subtracts (binary subtraction) one BCD value from another and stores the result in the AL register. The DAS instruction then adjusts the value in the AL register to obtain a valid, 2-digit, packed BCD value and sets the CF flag if a decimal borrow occurred as the result of the subtraction.

7.3.3.2 Unpacked BCD Adjustment Instructions

The AAA (ASCII adjust after addition), AAS (ASCII adjust after subtraction), AAM (ASCII adjust after multiplication), and AAD (ASCII adjust before division) instructions adjust the results of arithmetic operations performed on unpacked BCD values (see Section 4.7, “BCD and Packed BCD Integers”). All these instructions assume that the value to be adjusted is stored in the AL register or, in one instance, the AL and AH registers.

The AAA instruction adjusts the contents of the AL register following the addition of two unpacked BCD values. It converts the binary value in the AL register into a decimal value and stores the result in the AL register in unpacked BCD format (the decimal number is stored in the lower 4 bits of the register and the upper 4 bits are cleared). If a decimal carry occurred as a result of the addition, the CF flag is set and the contents of the AH register are incremented by 1.
The AAS instruction adjusts the contents of the AL register following the subtraction of two unpacked BCD values. Here again, a binary value is converted into an unpacked BCD value. If a borrow was required to complete the decimal subtract, the CF flag is set and the contents of the AH register are decremented by 1.

The AAM instruction adjusts the contents of the AL register following a multiplication of two unpacked BCD values. It converts the binary value in the AL register into a decimal value and stores the least significant digit of the result in the AL register (in unpacked BCD format) and the most significant digit, if there is one, in the AH register (also in unpacked BCD format).

The AAD instruction adjusts a two-digit BCD value so that when the value is divided with the DIV instruction, a valid unpacked BCD result is obtained. The instruction converts the BCD value in registers AH (most significant digit) and AL (least significant digit) into a binary value and stores the result in register AL. When the value in AL is divided by an unpacked BCD value, the quotient and remainder will be automatically encoded in unpacked BCD format.

### 7.3.4 Decimal Arithmetic Instructions in 64-Bit Mode

Decimal arithmetic instructions are not supported in 64-bit mode, they are either invalid or not encodable.

### 7.3.5 Logical Instructions

The logical instructions AND, OR, XOR (exclusive or), and NOT perform the standard Boolean operations for which they are named. The AND, OR, and XOR instructions require two operands; the NOT instruction operates on a single operand.

### 7.3.6 Shift and Rotate Instructions

The shift and rotate instructions rearrange the bits within an operand. For the purpose of this discussion, these instructions are further divided into subordinate subgroups of instructions that:

- Shift bits
- Double-shift bits (move them between operands)
- Rotate bits

#### 7.3.6.1 Shift Instructions

The SAL (shift arithmetic left), SHL (shift logical left), SAR (shift arithmetic right), SHR (shift logical right) instructions perform an arithmetic or logical shift of the bits in a byte, word, or doubleword.

The SAL and SHL instructions perform the same operation (see Figure 7-6). They shift the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand.
The SHR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-7). As with the SHL/SAL instruction, the empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand.

The SAR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-8). This instruction differs from the SHR instruction in that it preserves the sign of the source operand by clearing empty bit positions if the operand is positive or setting the empty bits if the operand is negative. Again, the CF flag is loaded with the last bit shifted out of the operand.

The SAR and SHR instructions can also be used to perform division by powers of 2 (see “SAL/SAR/SHL/SHR—Shift Instructions” in Chapter 4, “Instruction Set Reference, M-U,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B).
7.3.6.2 Double-Shift Instructions

The SHLD (shift left double) and SHRD (shift right double) instructions shift a specified number of bits from one operand to another (see Figure 7-9). They are provided to facilitate operations on unaligned bit strings. They can also be used to implement a variety of bit string move operations.

The SHLD instruction shifts the bits in the destination operand to the left and fills the empty bit positions (in the destination operand) with bits shifted out of the source operand. The destination and source operands must be the same length (either words or doublewords). The shift count can range from 0 to 31 bits. The result of this shift operation is stored in the destination operand, and the source operand is not modified. The CF flag is loaded with the last bit shifted out of the destination operand.

The SHRD instruction operates the same as the SHLD instruction except bits are shifted to the right in the destination operand, with the empty bit positions filled with bits shifted out of the source operand.
7.3.6.3 Rotate Instructions

The ROL (rotate left), ROR (rotate right), RCL (rotate through carry left) and RCR (rotate through carry right) instructions rotate the bits in the destination operand out of one end and back through the other end (see Figure 7-10). Unlike a shift, no bits are lost during a rotation. The rotate count can range from 0 to 31.

![Figure 7-10. ROL, ROR, RCL, and RCR Instruction Operations](image)

The ROL instruction rotates the bits in the operand to the left (toward more significant bit locations). The ROR instruction rotates the operand right (toward less significant bit locations).

The RCL instruction rotates the bits in the operand to the left, through the CF flag. This instruction treats the CF flag as a one-bit extension on the upper end of the operand. Each bit that exits from the most significant bit location of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the least significant bit location of the operand.

The RCR instruction rotates the bits in the operand to the right through the CF flag.

For all the rotate instructions, the CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand. The value of this flag can then be tested by a conditional jump instruction (JC or JNC).

7.3.7 Bit and Byte Instructions

These instructions operate on bit or byte strings. For the purpose of this discussion, they are further divided into subordinate subgroups that:

- Test and modify a single bit.
- Scan a bit string.
- Set a byte given conditions.
- Test operands and report results.
7.3.7.1 Bit Test and Modify Instructions

The bit test and modify instructions (see Table 7-3) operate on a single bit, which can be in an operand. The location of the bit is specified as an offset from the least significant bit of the operand. When the processor identifies the bit to be tested and modified, it first loads the CF flag with the current value of the bit. Then it assigns a new value to the selected bit, as determined by the modify operation for the instruction.

Table 7-3. Bit Test and Modify Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect on CF Flag</th>
<th>Effect on Selected Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT (Bit Test)</td>
<td>CF flag ← Selected Bit</td>
<td>No effect</td>
</tr>
<tr>
<td>BTS (Bit Test and Set)</td>
<td>CF flag ← Selected Bit</td>
<td>Selected Bit ← 1</td>
</tr>
<tr>
<td>BTR (Bit Test and Reset)</td>
<td>CF flag ← Selected Bit</td>
<td>Selected Bit ← 0</td>
</tr>
<tr>
<td>BTC (Bit Test and Complement)</td>
<td>CF flag ← Selected Bit</td>
<td>Selected Bit ← (\text{NOT} ) (Selected Bit)</td>
</tr>
</tbody>
</table>

7.3.7.2 Bit Scan Instructions

The BSF (bit scan forward) and BSR (bit scan reverse) instructions scan a bit string in a source operand for a set bit and store the bit index of the first set bit found in a destination register. The bit index is the offset from the least significant bit (bit 0) in the bit string to the first set bit. The BSF instruction scans the source operand low-to-high (from bit 0 of the source operand toward the most significant bit); the BSR instruction scans high-to-low (from the most significant bit toward the least significant bit).

7.3.7.3 Byte Set on Condition Instructions

The SET\(cc\) (set byte on condition) instructions set a destination-operand byte to 0 or 1, depending on the state of selected status flags (CF, OF, SF, ZF, and PF) in the EFLAGS register. The suffix \(cc\) added to the SET mnemonic determines the condition being tested for.

For example, the SETO instruction tests for overflow. If the OF flag is set, the destination byte is set to 1; if OF is clear, the destination byte is cleared to 0. Appendix B, “EFLAGS Condition Codes,” lists the conditions it is possible to test for with this instruction.

7.3.7.4 Test Instruction

The TEST instruction performs a logical AND of two operands and sets the SF, ZF, and PF flags according to the results. The flags can then be tested by the conditional jump or loop instructions or the SET\(cc\) instructions. The TEST instruction differs from the AND instruction in that it does not alter either of the operands.

7.3.8 Control Transfer Instructions

The processor provides both conditional and unconditional control transfer instructions to direct the flow of program execution. Conditional transfers are taken only for specified states of the status flags in the EFLAGS register. Unconditional control transfers are always executed.

For the purpose of this discussion, these instructions are further divided into subordinate subgroups that process:

- Unconditional transfers.
- Conditional transfers.
- Software interrupts.

7.3.8.1 Unconditional Transfer Instructions

The JMP, CALL, RET, INT, and IRET instructions transfer program control to another location (destination address) in the instruction stream. The destination can be within the same code segment (near transfer) or in a different code segment (far transfer).
**Jump instruction** — The JMP (jump) instruction unconditionally transfers program control to a destination instruction. The transfer is one-way; that is, a return address is not saved. A destination operand specifies the address (the instruction pointer) of the destination instruction. The address can be a relative address or an absolute address.

A relative address is a displacement (offset) with respect to the address in the EIP register. The destination address (a near pointer) is formed by adding the displacement to the address in the EIP register. The displacement is specified with a signed integer, allowing jumps either forward or backward in the instruction stream.

An absolute address is a offset from address 0 of a segment. It can be specified in either of the following ways:

- **An address in a general-purpose register** — This address is treated as a near pointer, which is copied into the EIP register. Program execution then continues at the new address within the current code segment.
- **An address specified using the standard addressing modes of the processor** — Here, the address can be a near pointer or a far pointer. If the address is for a near pointer, the address is translated into an offset and copied into the EIP register. If the address is for a far pointer, the address is translated into a segment selector (which is copied into the CS register) and an offset (which is copied into the EIP register).

In protected mode, the JMP instruction also allows jumps to a call gate, a task gate, and a task-state segment.

**Call and return instructions** — The CALL (call procedure) and RET (return from procedure) instructions allow a jump from one procedure (or subroutine) to another and a subsequent jump back (return) to the calling procedure.

The CALL instruction transfers program control from the current (or calling) procedure to another procedure (the called procedure). To allow a subsequent return to the calling procedure, the CALL instruction saves the current contents of the EIP register on the stack before jumping to the called procedure. The EIP register (prior to transferring program control) contains the address of the instruction following the CALL instruction. When this address is pushed on the stack, it is referred to as the return instruction pointer or return address.

The address of the called procedure (the address of the first instruction in the procedure being jumped to) is specified in a CALL instruction the same way as it is in a JMP instruction (see “Jump instruction” on page 7-15). The address can be specified as a relative address or an absolute address. If an absolute address is specified, it can be either a near or a far pointer.

The RET instruction transfers program control from the procedure currently being executed (the called procedure) back to the procedure that called it (the calling procedure). Transfer of control is accomplished by copying the return instruction pointer from the stack into the EIP register. Program execution then continues with the instruction pointed to by the EIP register.

The RET instruction has an optional operand, the value of which is added to the contents of the ESP register as part of the return operation. This operand allows the stack pointer to be incremented to remove parameters from the stack that were pushed on the stack by the calling procedure.

See Section 6.4, “Calling Procedures Using CALL and RET,” for more information on the mechanics of making procedure calls with the CALL and RET instructions.

**Return from interrupt instruction** — When the processor services an interrupt, it performs an implicit call to an interrupt-handling procedure. The IRET (return from interrupt) instruction returns program control from an interrupt handler to the interrupted procedure (that is, the procedure that was executing when the interrupt occurred). The IRET instruction performs a similar operation to the RET instruction (see "Call and return instructions" on page 7-15) except that it also restores the EFLAGS register from the stack. The contents of the EFLAGS register are automatically stored on the stack along with the return instruction pointer when the processor services an interrupt.

### 7.3.8.2 Conditional Transfer Instructions

The conditional transfer instructions execute jumps or loops that transfer program control to another instruction in the instruction stream if specified conditions are met. The conditions for control transfer are specified with a set of condition codes that define various states of the status flags (CF, ZF, OF, PF, and SF) in the EFLAGS register.

**Conditional jump instructions** — The Jcc (conditional) jump instructions transfer program control to a destination instruction if the conditions specified with the condition code (cc) associated with the instruction are satisfied (see Table 7-4). If the condition is not satisfied, execution continues with the instruction following the Jcc instruction. As with the JMP instruction, the transfer is one-way; that is, a return address is not saved.
The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The Jcc instructions do not support far transfers; however, far transfers can be accomplished with a combination of a Jcc and a JMP instruction (see "Jcc—Jump if Condition Is Met" in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

Table 7-4 shows the mnemonics for the Jcc instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter “J” to form the mnemonic for a Jcc instruction. The instructions are divided into two groups: unsigned and signed conditional jumps. These groups correspond to the results of operations performed on unsigned and signed integers respectively. Those instructions listed as pairs (for example, JA/JNBE) are alternate names for the same instruction. Assemblers provide alternate names to make it easier to read program listings.

The JCXZ and JECXZ instructions test the CX and ECX registers, respectively, instead of one or more status flags. See “Jump if zero instructions” on page 7-17 for more information about these instructions.

**Loop instructions** — The LOOP, LOOPE (loop while equal), LOOPZ (loop while zero), LOOPNE (loop while not equal), and LOOPNZ (loop while not zero) instructions are conditional jump instructions that use the value of the ECX register as a count for the number of times to execute a loop. All the loop instructions decrement the count in the ECX register each time they are executed and terminate a loop when zero is reached. The LOOP, LOOPZ, LOOPNE, and LOOPNZ instructions also accept the ZF flag as a condition for terminating the loop before the count reaches zero.

The LOOP instruction decrements the contents of the ECX register (or the CX register, if the address-size attribute is 16), then tests the register for the loop-termination condition. If the count in the ECX register is non-zero, program control is transferred to the instruction address specified by the destination operand. The destination
operand is a relative address (that is, an offset relative to the contents of the EIP register), and it generally points
to the first instruction in the block of code that is to be executed in the loop. When the count in the ECX register
reaches zero, program control is transferred to the instruction immediately following the LOOP instruction,
which terminates the loop. If the count in the ECX register is zero when the LOOP instruction is first executed, the
register is pre-decremented to FFFFFFFFH, causing the loop to be executed $2^{32}$ times.

The LOOPE and LOOPZ instructions perform the same operation (they are mnemonics for the same instruction).
These instructions operate the same as the LOOP instruction, except that they also test the ZF flag.

If the count in the ECX register is not zero and the ZF flag is set, program control is transferred to the destination
operand. When the count reaches zero or the ZF flag is clear, the loop is terminated by transferring program control
to the instruction immediately following the LOOPE/LOOPZ instruction.

The LOOPNE and LOOPNZ instructions (mnemonics for the same instruction) operate the same as the
LOOPE/LOOPZ instructions, except that they terminate the loop if the ZF flag is set.

Jump if zero instructions — The JECXZ (jump if ECX zero) instruction jumps to the location specified in the desti-
nation operand if the ECX register contains the value zero. This instruction can be used in combination with a loop
instruction (LOOP, LOOPE, LOOPZ, LOOPNE, or LOOPNZ) to test the ECX register prior to beginning a loop. As
described in “Loop instructions” on page 7-16, the loop instructions decrement the contents of the ECX register
before testing for zero. If the value in the ECX register is zero initially, it will be decremented to FFFFFFFFH on the
first loop instruction, causing the loop to be executed $2^{32}$ times. To prevent this problem, a JECXZ instruction can
be inserted at the beginning of the code block for the loop, causing a jump out of the loop if the ECX register count
is initially zero. When used with repeated string scan and compare instructions, the JECXZ instruction can deter-
mine whether the loop terminated because the count reached zero or because the scan or compare conditions were
satisfied.

The JCXZ (jump if CX is zero) instruction operates the same as the JECXZ instruction when the 16-bit address-size
attribute is used. Here, the CX register is tested for zero.

7.3.8.3 Control Transfer Instructions in 64-Bit Mode

In 64-bit mode, the operand size for all near branches (CALL, RET, JCC, JCXZ, JMP, and LOOP) is forced to 64 bits.
The listed instructions update the 64-bit RIP without need for a REX operand-size prefix.

Near branches in the following operations are forced to 64-bits (regardless of operand size prefixes):

- Truncation of the size of the instruction pointer.
- Size of a stack pop or push, due to CALL or RET.
- Size of a stack-pointer increment or decrement, due to CALL or RET.
- Indirect-branch operand size.

Note that the displacement field for relative branches is still limited to 32 bits and the address size for near
branches is not forced.

Address size determines the register size (CX/ECX/RCX) used for JCXZ and LOOP. It also impacts the address
calculation for memory indirect branches. Addresses size is 64 bits by default, although it can be over-ridden to 32
bits (using a prefix).

7.3.8.4 Software Interrupt Instructions

The INT $n$ (software interrupt), INTO (interrupt on overflow), and BOUND (detect value out of range) instructions
allow a program to explicitly raise a specified interrupt or exception, which in turn causes the handler routine for
the interrupt or exception to be called.

The INT $n$ instruction can raise any of the processor’s interrupts or exceptions by encoding the vector of the inter-
rupt or exception in the instruction. This instruction can be used to support software generated interrupts or to test
the operation of interrupt and exception handlers.

The IRET (return from interrupt) instruction returns program control from an interrupt handler to the interrupted
procedure. The IRET instruction performs a similar operation to the RET instruction.
The CALL (call procedure) and RET (return from procedure) instructions allow a jump from one procedure to another and a subsequent return to the calling procedure. EFLAGS register contents are automatically stored on the stack along with the return instruction pointer when the processor services an interrupt.

The INTO instruction raises the overflow exception if the OF flag is set. If the flag is clear, execution continues without raising the exception. This instruction allows software to access the overflow exception handler explicitly to check for overflow conditions.

The BOUND instruction compares a signed value against upper and lower bounds, and raises the “BOUND range exceeded” exception if the value is less than the lower bound or greater than the upper bound. This instruction is useful for operations such as checking an array index to make sure it falls within the range defined for the array.

### 7.3.8.5 Software Interrupt Instructions in 64-Bit Mode and Compatibility Mode

In 64-bit mode, the stack size is 8 bytes wide. IRET must pop 8-byte items off the stack. SS:RSP pops unconditionally. BOUND is not supported.

In compatibility mode, SS:RSP is popped only if the CPL changes.

### 7.3.9 String Operations

The GP instructions includes a set of **string instructions** that are designed to access large data structures; these are introduced in Section 7.3.9.1. Section 7.3.9.2 describes how REP prefixes can be used with these instructions to perform more complex **repeated string operations**. Certain processors optimize repeated string operations with **fast-string operation**, as described in Section 7.3.9.3. Section 7.3.9.4 explains how string operations can be used in 64-bit mode.

#### 7.3.9.1 String Instructions

The MOVS (Move String), CMPS (Compare string), SCAS (Scan string), LODS (Load string), and STOS (Store string) instructions permit large data structures, such as alphanumeric character strings, to be moved and examined in memory. These instructions operate on individual elements in a string, which can be a byte, word, or doubleword. The string elements to be operated on are identified with the ESI (source string element) and EDI (destination string element) registers. Both of these registers contain absolute addresses (offsets into a segment) that point to a string element.

By default, the ESI register addresses the segment identified with the DS segment register. A segment-override prefix allows the ESI register to be associated with the CS, SS, ES, FS, or GS segment register. The EDI register addresses the segment identified with the ES segment register; no segment override is allowed for the EDI register. The use of two different segment registers in the string instructions permits operations to be performed on strings located in different segments. Or by associating the ESI register with the ES segment register, both the source and destination strings can be located in the same segment. (This latter condition can also be achieved by loading the DS and ES segment registers with the same segment selector and allowing the ESI register to default to the DS register.)

The MOVS instruction moves the string element addressed by the ESI register to the location addressed by the EDI register. The assembler recognizes three "short forms" of this instruction, which specify the size of the string to be moved: MOVSB (move byte string), MOVSW (move word string), and MOVSD (move doubleword string).

The CMPS instruction subtracts the destination string element from the source string element and updates the status flags (CF, ZF, OF, SF, PF, and AF) in the EFLAGS register according to the results. Neither string element is written back to memory. The assembler recognizes three "short forms" of the CMPS instruction: CMPSB (compare byte strings), CMPSW (compare word strings), and CMPSD (compare doubleword strings).

The SCAS instruction subtracts the destination string element from the contents of the EAX, AX, or AL register (depending on operand length) and updates the status flags according to the results. The string element and register contents are not modified. The following "short forms" of the SCAS instruction specify the operand length: SCASB (scan byte string), SCASW (scan word string), and SCASD (scan doubleword string).

The LODS instruction loads the source string element identified by the ESI register into the EAX register (for a doubleword string), the AX register (for a word string), or the AL register (for a byte string). The "short forms" for
this instruction are LODSB (load byte string), LODSW (load word string), and LODSD (load doubleword string). This instruction is usually used in a loop, where other instructions process each element of the string after they are loaded into the target register.

The STOS instruction stores the source string element from the EAX (doubleword string), AX (word string), or AL (byte string) register into the memory location identified with the EDI register. The “short forms” for this instruction are STOSB (store byte string), STOSW (store word string), and STOSD (store doubleword string). This instruction is also normally used in a loop. Here a string is commonly loaded into the register with a LODS instruction, operated on by other instructions, and then stored again in memory with a STOS instruction.

The I/O instructions (see Section 7.3.10, “I/O Instructions”) also perform operations on strings in memory.

7.3.9.2 Repeated String Operations

Each of the string instructions described in Section 7.3.9.1 perform one iteration of a string operation. To operate on strings longer than a doubleword, the string instructions can be combined with a repeat prefix (REP) to create a repeating instruction or be placed in a loop.

When used in string instructions, the ESI and EDI registers are automatically incremented or decremented after each iteration of an instruction to point to the next element (byte, word, or doubleword) in the string. String operations can thus begin at higher addresses and work toward lower ones, or they can begin at lower addresses and work toward higher ones. The DF flag in the EFLAGS register controls whether the registers are incremented (DF = 0) or decremented (DF = 1). The STD and CLD instructions set and clear this flag, respectively.

The following repeat prefixes can be used in conjunction with a count in the ECX register to cause a string instruction to repeat:

- **REP** — Repeat while the ECX register not zero.
- **REPE/REPZ** — Repeat while the ECX register not zero and the ZF flag is set.
- **REPNE/REPNZ** — Repeat while the ECX register not zero and the ZF flag is clear.

When a string instruction has a repeat prefix, the operation executes until one of the termination conditions specified by the prefix is satisfied. The REPE/REPZ and REPNE/REPNZ prefixes are used only with the CMPS and SCAS instructions. Also, note that a REP STOS instruction is the fastest way to initialize a large block of memory.

7.3.9.3 Fast-String Operation

To improve performance, more recent processors support modifications to the processor’s operation during the string store operations initiated with the MOV$S, MOV$B, STOS, and STOSB instructions. This optimized operation, called fast-string operation, is used when the execution of one of those instructions meets certain initial conditions (see below). Instructions using fast-string operation effectively operate on the string in groups that may include multiple elements of the native data size (byte, word, doubleword, or quadword). With fast-string operation, the processor recognizes interrupts and data breakpoints only on boundaries between these groups. Fast-string operation is used only if the source and destination addresses both use either the WB or WC memory types.

The initial conditions for fast-string operation are implementation-specific and may vary with the native string size. Examples of parameters that may impact the use of fast-string operation include the following:

- The alignment indicated in the EDI and ESI alignment registers.
- The address order of the string operation.
- The value of the initial operation counter (ECX).
- The difference between the source and destination addresses.

**NOTE**

Initial conditions for fast-string operation in future Intel 64 or IA-32 processor families may differ from above. The Intel® 64 and IA-32 Architectures Optimization Reference Manual may contain model-specific information.
Software can disable fast-string operation by clearing the fast-string-enable bit (bit 0) of IA32_MISC_ENABLE MSR. However, Intel recommends that system software always enable fast-string operation.

When fast-string operation is enabled (because IA32_MISC_ENABLE[0] = 1), some processors may further enhance the operation of the REP MOVSB and REP STOSB instructions. A processor supports these enhancements if CPUID.(EAX=07H, ECX=0H):EBX[bit 9] is 1. The Intel® 64 and IA-32 Architectures Optimization Reference Manual may include model-specific recommendations for use of these enhancements.

The stores produced by fast-string operation may appear to execute out of order. Software dependent upon sequential store ordering should not use string operations for the entire data structure to be stored. Data and semaphores should be separated. Order-dependent code should write to a discrete semaphore variable after any string operations to allow correctly ordered data to be seen by all processors. Atomicity of load and store operations is guaranteed only for native data elements of the string with native data size, and only if they are included in a single cache line. See Section 9.2.4, “Fast-String Operation and Out-of-Order Stores,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

### 7.3.9.4 String Operations in 64-Bit Mode

The behavior of MOVS (Move String), CMPS (Compare string), SCAS (Scan string), LODS (Load string), and STOS (Store string) instructions in 64-bit mode is similar to their behavior in non-64-bit modes, with the following differences:

- The source operand is specified by RSI or DS:ESI, depending on the address size attribute of the operation.
- The destination operand is specified by RDI or DS:EDI, depending on the address size attribute of the operation.
- Operation on 64-bit data is supported by using the REX.W prefix.

When using REP prefixes for string operations in 64-bit mode, the repeat count is specified by RCX or ECX (depending on the address size attribute of the operation). The default address size is 64 bits.

### 7.3.10 I/O Instructions

The IN (input from port to register), INS (input from port to string), OUT (output from register to port), and OUTS (output string to port) instructions move data between the processor’s I/O ports and either a register or memory.

The register I/O instructions (IN and OUT) move data between an I/O port and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The I/O port being read or written to is specified with an immediate operand or an address in the DX register.

The block I/O instructions (INS and OUTS) move blocks of data (strings) between an I/O port and memory. These instructions operate similar to the string instructions (see Section 7.3.9, “String Operations”). The ESI and EDI registers are used to specify string elements in memory and the repeat prefix (REP) is used to repeat the instructions to implement block moves. The assembler recognizes the following alternate mnemonics for these instructions: INSB (input byte), INSW (input word), and INSĐ (input doubleword), and OUTSB (output byte), OUTSW (output word), and OUTSD (output doubleword).

The INS and OUTS instructions use an address in the DX register to specify the I/O port to be read or written to.

### 7.3.11 I/O Instructions in 64-Bit Mode

For I/O instructions to and from memory, the differences in 64-bit mode are:

- The source operand is specified by RSI or DS:ESI, depending on the address size attribute of the operation.
- The destination operand is specified by RDI or DS:EDI, depending on the address size attribute of the operation.
- Operation on 64-bit data is not encodable and REX prefixes are silently ignored.
7.3.12 Enter and Leave Instructions

The ENTER and LEAVE instructions provide machine-language support for procedure calls in block-structured languages, such as C and Pascal. These instructions and the call and return mechanism that they support are described in detail in Section 6.6, “Procedure Calls for Block-Structured Languages.”

7.3.13 Flag Control (EFLAG) Instructions

The Flag Control (EFLAG) instructions allow the state of selected flags in the EFLAGS register to be read or modified. For the purpose of this discussion, these instructions are further divided into subordinate subgroups of instructions that manipulate:

- Carry and direction flags.
- The EFLAGS register.
- Interrupt flags.

7.3.13.1 Carry and Direction Flag Instructions

The STC (set carry flag), CLC (clear carry flag), and CMC (complement carry flag) instructions allow the CF flag in the EFLAGS register to be modified directly. They are typically used to initialize the CF flag to a known state before an instruction that uses the flag in an operation is executed. They are also used in conjunction with the rotate-with-carry instructions (RCL and RCR).

The STD (set direction flag) and CLD (clear direction flag) instructions allow the DF flag in the EFLAGS register to be modified directly. The DF flag determines the direction in which index registers ESI and EDI are stepped when executing string processing instructions. If the DF flag is clear, the index registers are incremented after each iteration of a string instruction; if the DF flag is set, the registers are decremented.

7.3.13.2 EFLAGS Transfer Instructions

The EFLAGS transfer instructions allow groups of flags in the EFLAGS register to be copied to a register or memory or be loaded from a register or memory.

The LAHF (load AH from flags) and SAHF (store AH into flags) instructions operate on five of the EFLAGS status flags (SF, ZF, AF, PF, and CF). The LAHF instruction copies the status flags to bits 7, 6, 4, 2, and 0 of the AH register, respectively. The contents of the remaining bits in the register (bits 5, 3, and 1) are unaffected, and the contents of the EFLAGS register remain unchanged. The SAHF instruction copies bits 7, 6, 4, 2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags, respectively in the EFLAGS register.

The PUSHF (push flags), PUSHFD (push flags double), POPF (pop flags), and POPFD (pop flags double) instructions copy the flags in the EFLAGS register to and from the stack. The PUSHF instruction pushes the lower word of the EFLAGS register onto the stack (see Figure 7-11). The PUSHFD instruction pushes the entire EFLAGS register onto the stack (with the RF and VM flags read as clear).

![Figure 7-11. Flags Affected by the PUSHF, POPF, PUSHFD, and POPFD Instructions](image-url)

The POPF instruction pops a word from the stack into the EFLAGS register. Only bits 11, 10, 8, 7, 6, 4, 2, and 0 of the EFLAGS register are affected with all uses of this instruction. If the current privilege level (CPL) of the current
code segment is 0 (most privileged), the IOPL bits (bits 13 and 12) also are affected. If the I/O privilege level (IOPL) is greater than or equal to the CPL, numerically, the IF flag (bit 9) also is affected.

The POPFD instruction pops a doubleword into the EFLAGS register. This instruction can change the state of the AC bit (bit 18) and the ID bit (bit 21), as well as the bits affected by a POPF instruction. The restrictions for changing the IOPL bits and the IF flag that were given for the POPF instruction also apply to the POPFD instruction.

### 7.3.13.3 Interrupt Flag Instructions

The STI (set interrupt flag) and CLI (clear interrupt flag) instructions allow the interrupt IF flag in the EFLAGS register to be modified directly. The IF flag controls the servicing of hardware-generated interrupts (those received at the processor’s INTR pin). If the IF flag is set, the processor services hardware interrupts; if the IF flag is clear, hardware interrupts are masked.

The ability to execute these instructions depends on the operating mode of the processor and the current privilege level (CPL) of the program or task attempting to execute these instructions.

### 7.3.14 Flag Control (RFLAG) Instructions in 64-Bit Mode

In 64-bit mode, the LAHF and SAHF instructions are supported if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1. PUSHF and POPF behave the same in 64-bit mode as in non-64-bit mode. PUSHFD always pushes 64-bit RFLAGS onto the stack (with the RF and VM flags read as clear). POPFD always pops a 64-bit value from the top of the stack and loads the lower 32 bits into RFLAGS. It then zero extends the upper bits of RFLAGS.

### 7.3.15 Segment Register Instructions

The processor provides a variety of instructions that address the segment registers of the processor directly. These instructions are only used when an operating system or executive is using the segmented or the real-address mode memory model.

For the purpose of this discussion, these instructions are divided into subordinate subgroups of instructions that allow:
- Segment-register load and store.
- Far control transfers.
- Software interrupt calls.
- Handling of far pointers.

#### 7.3.15.1 Segment-Register Load and Store Instructions

The MOV instruction (introduced in Section 7.3.1.1, “General Data Movement Instructions”) and the PUSH and POP instructions (introduced in Section 7.3.1.4, “Stack Manipulation Instructions”) can transfer 16-bit segment selectors to and from segment registers (DS, ES, FS, GS, and SS). The transfers are always made to or from a segment register and a general-purpose register or memory. Transfers between segment registers are not supported.

The POP and MOV instructions cannot place a value in the CS register. Only the far control-transfer versions of the JMP, CALL, and RET instructions (see Section 7.3.15.2, “Far Control Transfer Instructions”) affect the CS register directly.

#### 7.3.15.2 Far Control Transfer Instructions

The JMP and CALL instructions (see Section 7.3.8, “Control Transfer Instructions”) both accept a far pointer as a destination to transfer program control to a segment other than the segment currently being pointed to by the CS register. When a far call is made with the CALL instruction, the current values of the EIP and CS registers are both pushed on the stack.

The RET instruction (see “Call and return instructions” on page 7-15) can be used to execute a far return. Here, program control is transferred from a code segment that contains a called procedure back to the code segment that
The RET instruction restores the values of the CS and EIP registers for the calling procedure from the stack.

### 7.3.15.3 Software Interrupt Instructions

The software interrupt instructions INT, INTO, and IRET (see Section 7.3.8.4, “Software Interrupt Instructions”) can also call and return from interrupt and exception handler procedures that are located in a code segment other than the current code segment. With these instructions, however, the switching of code segments is handled transparently from the application program.

### 7.3.15.4 Load Far Pointer Instructions

The load far pointer instructions LDS (load far pointer using DS), LES (load far pointer using ES), LFS (load far pointer using FS), LGS (load far pointer using GS), and LSS (load far pointer using SS) load a far pointer from memory into a segment register and a general-purpose general register. The segment selector part of the far pointer is loaded into the selected segment register and the offset is loaded into the selected general-purpose register.

### 7.3.16 Miscellaneous Instructions

The following instructions perform operations that are of interest to applications programmers. For the purpose of this discussion, these instructions are further divided into subordinate subgroups of instructions that provide for:

- Address computations.
- Table lookup.
- Processor identification.
- NOP and undefined instruction entry.

#### 7.3.16.1 Address Computation Instruction

The LEA (load effective address) instruction computes the effective address in memory (offset within a segment) of a source operand and places it in a general-purpose register. This instruction can interpret any of the processor’s addressing modes and can perform any indexing or scaling that may be needed. It is especially useful for initializing the ESI or EDI registers before the execution of string instructions or for initializing the EBX register before an XLAT instruction.

#### 7.3.16.2 Table Lookup Instructions

The XLAT and XLATB (table lookup) instructions replace the contents of the AL register with a byte read from a translation table in memory. The initial value in the AL register is interpreted as an unsigned index into the translation table. This index is added to the contents of the EBX register (which contains the base address of the table) to calculate the address of the table entry. These instructions are used for applications such as converting character codes from one alphabet into another (for example, an ASCII code could be used to look up its EBCDIC equivalent in a table).

#### 7.3.16.3 Processor Identification Instruction

The CPUID (processor identification) instruction returns information about the processor on which the instruction is executed.

#### 7.3.16.4 No-Operation and Undefined Instructions

The NOP (no operation) instruction increments the EIP register to point at the next instruction, but affects nothing else.
The UD (undefined) instruction generates an invalid opcode exception. Intel reserves the opcode for this instruction for this function. The instruction is provided to allow software to test an invalid opcode exception handler.

### 7.3.17 Random Number Generator Instructions

The instructions for generating random numbers to comply with NIST SP800-90A, SP800-90B, and SP800-90C standards are described in this section.

#### 7.3.17.1 RDRAND

The RDRAND instruction returns a random number. All Intel processors that support the RDRAND instruction indicate the availability of the RDRAND instruction via reporting CPUID.01H:ECX.RDRAND[bit 30] = 1.

RDRAND returns random numbers that are supplied by a cryptographically secure, deterministic random bit generator DRBG. The DRBG is designed to meet the NIST SP 800-90A standard. The DRBG is re-seeded frequently from an on-chip non-deterministic entropy source to guarantee data returned by RDRAND is statistically uniform, non-periodic and non-deterministic.

In order for the hardware design to meet its security goals, the random number generator continuously tests itself and the random data it is generating. Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDRAND instruction will return no data instead of bad data.

Under heavy load, with multiple cores executing RDRAND in parallel, it is possible, though unlikely, for the demand of random numbers by software processes/threads to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDRAND instruction returning no data transitorily. The RDRAND instruction indicates the occurrence of this rare situation by clearing the CF flag.

The RDRAND instruction returns with the carry flag set (CF = 1) to indicate valid data is returned. It is recommended that software using the RDRAND instruction to get random numbers retry for a limited number of iterations while RDRAND returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows. A retry limit should be employed to prevent a hard failure in the RNG (expected to be extremely rare) leading to a busy loop in software.

The intrinsic primitive for RDRAND is defined to address software’s need for the common cases (CF = 1) and the rare situations (CF = 0). The intrinsic primitive returns a value that reflects the value of the carry flag returned by the underlying RDRAND instruction. The example below illustrates the recommended usage of an RDRAND intrinsic in a utility function, a loop to fetch a 64 bit random value with a retry count limit of 10. A C implementation might be written as follows:

```c
#define SUCCESS 1
#define RETRY_LIMIT_EXCEEDED 0
#define RETRY_LIMIT 10

int get_random_64( unsigned __int 64 * arand) {
    int i;
    for ( i = 0; i < RETRY_LIMIT; i++) {
        if(_rdrand64_step(arand)) return SUCCESS;
    }
    return RETRY_LIMIT_EXCEEDED;
}
```

#### 7.3.17.2 RDSEED

The RDSEED instruction returns a random number. All Intel processors that support the RDSEED instruction indicate the availability of the RDSEED instruction via reporting CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 1.
RDSEED returns random numbers that are supplied by a cryptographically secure, enhanced non-deterministic random bit generator (Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP800-90C standards.

In order for the hardware design to meet its security goals, the random number generator continuously tests itself and the random data it is generating. Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDSEED instruction will return no data instead of bad data.

Under heavy load, with multiple cores executing RDSEED in parallel, it is possible for the demand of random numbers by software processes/threads to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDSEED instruction returning no data transitorily. The RDSEED instruction indicates the occurrence of this situation by clearing the CF flag.

The RDSEED instruction returns with the carry flag set (CF = 1) to indicate valid data is returned. It is recommended that software using the RDSEED instruction to get random numbers retry for a limited number of iterations while RDSEED returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows. A retry limit should be employed to prevent a hard failure in the NRBG (expected to be extremely rare) leading to a busy loop in software.

The intrinsic primitive for RDSEED is defined to address software’s need for the common cases (CF = 1) and the rare situations (CF = 0). The intrinsic primitive returns a value that reflects the value of the carry flag returned by the underlying RDSEED instruction.
PROGRAMMING WITH GENERAL-PURPOSE INSTRUCTIONS
The x87 Floating-Point Unit (FPU) provides high-performance floating-point processing capabilities for use in graphics processing, scientific, engineering, and business applications. It supports the floating-point, integer, and packed BCD integer data types and the floating-point processing algorithms and exception handling architecture defined in the IEEE Standard 754 for Floating-Point Arithmetic.

This chapter describes the x87 FPU execution environment and instruction set. It also provides exception handling information that is specific to the x87 FPU. Refer to the following chapters or sections of chapters for additional information about x87 FPU instructions and floating-point operations:

- Section 4.2.2, "Floating-Point Data Types," Section 4.2.1.2, "Signed Integers," and Section 4.7, "BCD and Packed BCD Integers," describe the floating-point, integer, and BCD data types.
- Section 4.9, "Overview of Floating-Point Exceptions," Section 4.9.1, "Floating-Point Exception Conditions," and Section 4.9.2, "Floating-Point Exception Priority," give an overview of the floating-point exceptions that the x87 FPU can detect and report.

8.1 X87 FPU EXECUTION ENVIRONMENT

The x87 FPU represents a separate execution environment within the IA-32 architecture (see Figure 8-1). This execution environment consists of eight data registers (called the x87 FPU data registers) and the following special-purpose registers:

• Status register.
• Control register.
• Tag word register.
• Last instruction pointer register.
• Last data (operand) pointer register.
• Opcode register.

These registers are described in the following sections.

The x87 FPU executes instructions from the processor’s normal instruction stream. The state of the x87 FPU is independent from the state of the basic execution environment and from the state of SSE/SSE2/SSE3 extensions. However, the x87 FPU and Intel MMX technology share state because the MMX registers are aliased to the x87 FPU data registers. Therefore, when writing code that uses x87 FPU and MMX instructions, the programmer must explicitly manage the x87 FPU and MMX state (see Section 9.5, "Compatibility with x87 FPU Architecture").

8.1.1 x87 FPU in 64-Bit Mode and Compatibility Mode

In compatibility mode and 64-bit mode, x87 FPU instructions function like they do in protected mode. Memory operands are specified using the ModR/M, SIB encoding that is described in Section 3.7.5, "Specifying an Offset."

8.1.2 x87 FPU Data Registers

The x87 FPU data registers (shown in Figure 8-1) consist of eight 80-bit registers. Values are stored in these registers in the double extended precision floating-point format shown in Figure 4-3. When floating-point, integer, or packed BCD integer values are loaded from memory into any of the x87 FPU data registers, the values are automatically converted into double extended precision floating-point format (if they are not already in that format). When computation results are subsequently transferred back into memory from any of the x87 FPU registers, the
results can be left in the double extended precision floating-point format or converted back into a shorter floating-point format, an integer format, or the packed BCD integer format. (See Section 8.2, “x87 FPU Data Types,” for a description of the data types operated on by the x87 FPU.)

The x87 FPU instructions treat the eight x87 FPU data registers as a register stack (see Figure 8-2). All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP (stack TOP) field in the x87 FPU status word. Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory and then increment TOP by one. (For the x87 FPU, a load operation is equivalent to a push and a store operation is equivalent to a pop.) Note that load and store operations are also available that do not push and pop the stack.

If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception indicates when wraparound might cause an unsaved value to be overwritten (see Section 8.5.1.1, “Stack Overflow or Underflow Exception (#IS)”).

Many floating-point instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP. Assemblers support these
register addressing modes, using the expression ST(0), or simply ST, to represent the current stack top and ST(i) to specify the \(i\)th register from TOP in the stack \((0 \leq i \leq 7)\). For example, if TOP contains 011B (register 3 is the top of the stack), the following instruction would add the contents of two registers in the stack (registers 3 and 5):

\[
FADD ST, ST(2);
\]

Figure 8-3 shows an example of how the stack structure of the x87 FPU registers and instructions are typically used to perform a series of computations. Here, a two-dimensional dot product is computed, as follows:

1. The first instruction (FLD value1) decrements the stack register pointer (TOP) and loads the value 5.6 from memory into ST(0). The result of this operation is shown in snapshot (a).
2. The second instruction multiplies the value in ST(0) by the value 2.4 from memory and stores the result in ST(0), shown in snapshot (b).
3. The third instruction decrements TOP and loads the value 3.8 in ST(0).
4. The fourth instruction multiplies the value in ST(0) by the value 10.3 from memory and stores the result in ST(0), shown in snapshot (c).
5. The fifth instruction adds the value and the value in ST(1) and stores the result in ST(0), shown in snapshot (d).

\[
\text{Computation} \\
\text{Dot Product} = (5.6 \times 2.4) + (3.8 \times 10.3)
\]

\[
\text{Code:} \\
\text{FLD value1 ;(a) value1 = 5.6} \\
\text{FMUL value2 ;(b) value2 = 2.4} \\
\text{FLD value3 ; value3 = 3.8} \\
\text{FMUL value4 ;(c)value4 = 10.3} \\
\text{FADD ST(1) ;(d)}
\]

The style of programming demonstrated in this example is supported by the floating-point instruction set. In cases where the stack structure causes computation bottlenecks, the FXCH (exchange x87 FPU register contents) instruction can be used to streamline a computation.

### 8.1.2.1 Parameter Passing With the x87 FPU Register Stack

Like the general-purpose registers, the contents of the x87 FPU data registers are unaffected by procedure calls, or in other words, the values are maintained across procedure boundaries. A calling procedure can thus use the x87 FPU data registers (as well as the procedure stack) for passing parameter between procedures. The called procedure can reference parameters passed through the register stack using the current stack register pointer (TOP) and the ST(0) and ST(i) nomenclature. It is also common practice for a called procedure to leave a return value or result in register ST(0) when returning execution to the calling procedure or program.
When mixing MMX and x87 FPU instructions in the procedures or code sequences, the programmer is responsible for maintaining the integrity of parameters being passed in the x87 FPU data registers. If an MMX instruction is executed before the parameters in the x87 FPU data registers have been passed to another procedure, the parameters may be lost (see Section 9.5, "Compatibility with x87 FPU Architecture").

8.1.3 x87 FPU Status Register

The 16-bit x87 FPU status register (see Figure 8-4) indicates the current state of the x87 FPU. The flags in the x87 FPU status register include the FPU busy flag, top-of-stack (TOP) pointer, condition code flags, exception summary status flag, stack fault flag, and exception flags. The x87 FPU sets the flags in this register to show the results of operations.

Figure 8-4. x87 FPU Status Word

The contents of the x87 FPU status register (referred to as the x87 FPU status word) can be stored in memory using the FSTSW/FNSTSW, FSTENV/FNSTENV, FSAVE/FNSAVE, and FXSAVE instructions. It can also be stored in the AX register of the integer unit, using the FSTSW/FNSTSW instructions.

8.1.3.1 Top of Stack (TOP) Pointer

A pointer to the x87 FPU data register that is currently at the top of the x87 FPU register stack is contained in bits 11 through 13 of the x87 FPU status word. This pointer, which is commonly referred to as TOP (for top-of-stack), is a binary value from 0 to 7. See Section 8.1.2, "x87 FPU Data Registers," for more information about the TOP pointer.

8.1.3.2 Condition Code Flags

The four condition code flags (C0 through C3) indicate the results of floating-point comparison and arithmetic operations. Table 8-1 summarizes the manner in which the floating-point instructions set the condition code flags. These condition code bits are used principally for conditional branching and for storage of information used in exception handling (see Section 8.1.4, "Branching and Conditional Moves on Condition Codes").

As shown in Table 8-1, the C1 condition code flag is used for a variety of functions. When both the IE and SF flags in the x87 FPU status word are set, indicating a stack overflow or underflow exception (#IS), the C1 flag distinguishes between overflow (C1 = 1) and underflow (C1 = 0). When the PE flag in the status word is set, indicating an inexact (rounded) result, the C1 flag is set to 1 if the last rounding by the instruction was upward. The FXAM instruction sets C1 to the sign of the value being examined.
The C2 condition code flag is used by the FPREM and FPREM1 instructions to indicate an incomplete reduction (or partial remainder). When a successful reduction has been completed, the C0, C3, and C1 condition code flags are set to the three least-significant bits of the quotient (Q2, Q1, and Q0, respectively). See “FPREM1—Partial Remainder” in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for more information on how these instructions use the condition code flags.

The FPTAN, FSIN, FCOS, and FSINCOS instructions set the C2 flag to 1 to indicate that the source operand is beyond the allowable range of ±2^63 and clear the C2 flag if the source operand is within the allowable range.

Where the state of the condition code flags are listed as undefined in Table 8-1, do not rely on any specific value in these flags.

### 8.1.3.3 x87 FPU Floating-Point Exception Flags

The six x87 FPU floating-point exception flags (bits 0 through 5) of the x87 FPU status word indicate that one or more floating-point exceptions have been detected since the bits were last cleared. The individual exception flags (IE, DE, ZE, OE, UE, and PE) are described in detail in Section 8.4, “x87 FPU Floating-Point Exception Handling.” Each of the exception flags can be masked by an exception mask bit in the x87 FPU control word (see Section 8.1.5, “x87 FPU Control Word”). The exception summary status flag (ES, bit 7) is set when any of the unmasked exception flags are set. When the ES flag is set, the x87 FPU exception handler is invoked, using one of the techniques described in Section 8.7, “Handling x87 FPU Exceptions in Software.” (Note that if an exception flag is masked, the x87 FPU will still set the appropriate flag if the associated exception occurs, but it will not set the ES flag.)

The exception flags are “sticky” bits (once set, they remain set until explicitly cleared). They can be cleared by executing the FCLEX/FNCLEX (clear exceptions) instructions, by reinitializing the x87 FPU with the FINIT/FNINIT or FSAVE/FNSAVE instructions, or by overwriting the flags with an FRSTOR or FLDENV instruction.

The B-bit (bit 15) is included for 8087 compatibility only. It reflects the contents of the ES flag.

### Table 8-1. Condition Code Interpretation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C0</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCOM, FCOMP, FCOMPP, FICOM,</td>
<td>Result</td>
<td>Operands</td>
<td>0 or #IS</td>
<td></td>
</tr>
<tr>
<td>FICOMP, FTST, FUCOM, FUCOMP</td>
<td>of Comparison</td>
<td>not comparable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOMI, FCOMIP, FUCOMI, FUCOM</td>
<td>Undefined</td>
<td>(These instructions set the status flags in the EFLAGS register.)</td>
<td>#IS</td>
<td></td>
</tr>
<tr>
<td>FXAM</td>
<td>Q2</td>
<td>Q1</td>
<td>0 = reduction complete</td>
<td>Q0 or #IS</td>
</tr>
<tr>
<td>FPREM, FPREM1</td>
<td></td>
<td></td>
<td>1 = reduction incomplete</td>
<td></td>
</tr>
<tr>
<td>F2XM1, FADD, FADDP, FBSTP,</td>
<td>Undefined</td>
<td>Roundup or #IS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMOVcc, FIADD, FDIV, FDIVP,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV, FDIVR, FIDIV, FIDIVR,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIDIV, FIMUL, FIST, FISTP,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISUB, FISUBR, FMUL, FMULP,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPATAN, FRNDINT, FSQRT, FST,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP, FSUB, FSUBP, FSUBR,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSQRT, FYL2X, FYL2XP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOS, FSIN, FSINCOS, FPTAN</td>
<td>Undefined</td>
<td>0 = source operand within range</td>
<td>Roundup or #IS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = source operand out of range</td>
<td>(Undefined if C2 = 1)</td>
</tr>
<tr>
<td>FABS, FBLD, FCHS, FDECPSTP,</td>
<td>Undefined</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD, Load Constants, FSTP</td>
<td></td>
<td></td>
<td></td>
<td>(ext. prec.), FXCH,</td>
</tr>
<tr>
<td>FEXTRACT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vol. 1 8-5
8.1.3.4 Stack Fault Flag

The stack fault flag (bit 6 of the x87 FPU status word) indicates that stack overflow or stack underflow has occurred with data in the x87 FPU data register stack. The x87 FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition.

When this flag is set, the condition code flag C1 indicates the nature of the fault: overflow (C1 = 1) and underflow (C1 = 0). The SF flag is a “sticky” flag, meaning that after it is set, the processor does not clear it until it is explicitly instructed to do so (for example, by an FINIT/FNINIT, FCLEX/FNCLEX, or FSAVE/FNSAVE instruction).

See Section 8.1.7, “x87 FPU Tag Word,” for more information on x87 FPU stack faults.

8.1.4 Branching and Conditional Moves on Condition Codes

The x87 FPU (beginning with the P6 family processors) supports two mechanisms for branching and performing conditional moves according to comparisons of two floating-point values. These mechanisms are referred to here as the “old mechanism” and the “new mechanism.”

The old mechanism is available in the x87 FPU prior to the P6 family processors and in P6 family processors. This mechanism uses the floating-point compare instructions (FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, and FICOMP) to compare two floating-point values and set the condition code flags (C0 through C3) according to the results. The contents of the condition code flags are then copied into the status flags of the EFLAGS register using a two step process (see Figure 8-5):

1. The FSTSW AX instruction moves the x87 FPU status word into the AX register.
2. The SAHF instruction copies the upper 8 bits of the AX register, which includes the condition code flags, into the lower 8 bits of the EFLAGS register.

When the condition code flags have been loaded into the EFLAGS register, conditional jumps or conditional moves can be performed based on the new settings of the status flags in the EFLAGS register.

Table 8-1. Condition Code Interpretation (Contd.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLENV, FRSTOR</td>
<td>Each bit loaded from memory</td>
</tr>
<tr>
<td>FFREE, FLDCW, FCLEX/FNCLEX, FNOP, FSTCW/FNSTCW, FSTENV/FNSTENV, FSTSW/FNSTSW,</td>
<td>Undefined</td>
</tr>
<tr>
<td>FINIT/FNINIT, FSAVE/FNSAVE</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

---

8-6 Vol. 1
The new mechanism is available beginning with the P6 family processors. Using this mechanism, the new floating-point compare and set EFLAGS instructions (FCOMI, FCOMIP, FUCOMI, and FUCOMIP) compare two floating-point values and set the ZF, PF, and CF flags in the EFLAGS register directly. A single instruction thus replaces the three instructions required by the old mechanism.

Note also that the FCMOVcc instructions (also new in the P6 family processors) allow conditional moves of floating-point values (values in the x87 FPU data registers) based on the setting of the status flags (ZF, PF, and CF) in the EFLAGS register. These instructions eliminate the need for an IF statement to perform conditional moves of floating-point values.

8.1.5 x87 FPU Control Word

The 16-bit x87 FPU control word (see Figure 8-6) controls the precision of the x87 FPU and rounding method used. It also contains the x87 FPU floating-point exception mask bits. The control word is cached in the x87 FPU control register. The contents of this register can be loaded with the FLDCW instruction and stored in memory with the FSTCW/FNSTCW instructions.

![Figure 8-6. x87 FPU Control Word](image)

When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU control word is set to 037FH, which masks all floating-point exceptions, sets rounding to nearest, and sets the x87 FPU precision to 64 bits.

8.1.5.1 x87 FPU Floating-Point Exception Mask Bits

The exception-flag mask bits (bits 0 through 5 of the x87 FPU control word) mask the 6 floating-point exception flags in the x87 FPU status word. When one of these mask bits is set, its corresponding x87 FPU floating-point exception is blocked from being generated.

8.1.5.2 Precision Control Field

The precision-control (PC) field (bits 8 and 9 of the x87 FPU control word) determines the precision (64, 53, or 24 bits) of floating-point calculations made by the x87 FPU (see Table 8-2). The default precision is double extended precision, which uses the full 64-bit significand available with the double extended precision floating-point format of the x87 FPU data registers. This setting is best suited for most applications, because it allows applications to take full advantage of the maximum precision available with the x87 FPU data registers.
The double precision and single precision settings reduce the size of the significand to 53 bits and 24 bits, respectively. These settings are provided to support IEEE Standard 754 and to provide compatibility with the specifications of certain existing programming languages. Using these settings nullifies the advantages of the double extended precision floating-point format’s 64-bit significand length. When reduced precision is specified, the rounding of the significand value clears the unused bits on the right to zeros.

The precision-control bits only affect the results of the following floating-point instructions: FADD, FADDP, FIADD, FSUB, FSUBP, FISUB, FSUBR, FSUBRP, FISUBR, FMUL, FMULP, FIMUL, FDIV, FDIVP, FIDIV, FDIVR, FDIVRP, FIDIVR, and FSQRT.

8.1.5.3 Rounding Control Field

The rounding-control (RC) field of the x87 FPU control register (bits 10 and 11) controls how the results of x87 FPU floating-point instructions are rounded. See Section 4.8.4, “Rounding,” for a discussion of rounding of floating-point values; See Section 4.8.4.1, ”Rounding Control (RC) Fields,” for the encodings of the RC field.

8.1.6 Infinity Control Flag

The infinity control flag (bit 12 of the x87 FPU control word) is provided for compatibility with the Intel 287 Math Coprocessor; it is not meaningful for later version x87 FPU coprocessors or IA-32 processors. See Section 4.8.3.3, “Signed Infinities,” for information on how the x87 FPUs handle infinity values.

8.1.7 x87 FPU Tag Word

The 16-bit tag word (see Figure 8-7) indicates the contents of each the 8 registers in the x87 FPU data-register stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number, zero, or a special floating-point number (NaN, infinity, denormal, or unsupported format), or whether it is empty. The x87 FPU tag word is cached in the x87 FPU in the x87 FPU tag word register. When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU tag word is set to FFFFH, which marks all the x87 FPU data registers as empty.

<table>
<thead>
<tr>
<th>Precision</th>
<th>PC Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision (24 bits)</td>
<td>00B</td>
</tr>
<tr>
<td>Reserved</td>
<td>01B</td>
</tr>
<tr>
<td>Double Precision (53 bits)</td>
<td>10B</td>
</tr>
<tr>
<td>Double Extended Precision (64 bits)</td>
<td>11B</td>
</tr>
</tbody>
</table>

The double precision and single precision settings reduce the size of the significand to 53 bits and 24 bits, respectively. These settings are provided to support IEEE Standard 754 and to provide compatibility with the specifications of certain existing programming languages. Using these settings nullifies the advantages of the double extended precision floating-point format’s 64-bit significand length. When reduced precision is specified, the rounding of the significand value clears the unused bits on the right to zeros.

The precision-control bits only affect the results of the following floating-point instructions: FADD, FADDP, FIADD, FSUB, FSUBP, FISUB, FSUBR, FSUBRP, FISUBR, FMUL, FMULP, FIMUL, FDIV, FDIVP, FIDIV, FDIVR, FDIVRP, FIDIVR, and FSQRT.

8.1.5.3 Rounding Control Field

The rounding-control (RC) field of the x87 FPU control register (bits 10 and 11) controls how the results of x87 FPU floating-point instructions are rounded. See Section 4.8.4, “Rounding,” for a discussion of rounding of floating-point values; See Section 4.8.4.1, ”Rounding Control (RC) Fields,” for the encodings of the RC field.

8.1.6 Infinity Control Flag

The infinity control flag (bit 12 of the x87 FPU control word) is provided for compatibility with the Intel 287 Math Coprocessor; it is not meaningful for later version x87 FPU coprocessors or IA-32 processors. See Section 4.8.3.3, “Signed Infinities,” for information on how the x87 FPUs handle infinity values.

8.1.7 x87 FPU Tag Word

The 16-bit tag word (see Figure 8-7) indicates the contents of each the 8 registers in the x87 FPU data-register stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number, zero, or a special floating-point number (NaN, infinity, denormal, or unsupported format), or whether it is empty. The x87 FPU tag word is cached in the x87 FPU in the x87 FPU tag word register. When the x87 FPU is initialized with either an FINIT/FNINIT or FSAVE/FNSAVE instruction, the x87 FPU tag word is set to FFFFH, which marks all the x87 FPU data registers as empty.
The x87 FPU uses the tag values to detect stack overflow and underflow conditions (see Section 8.5.1.1, "Stack Overflow or Underflow Exception (#IS)").

Application programs and exception handlers can use this tag information to check the contents of an x87 FPU data register without performing complex decoding of the actual data in the register. To read the tag register, it must be stored in memory using either the FSTENV/FNSTENV or FSAVE/FNSAVE instructions. The location of the tag word in memory after being saved with one of these instructions is shown in Figures 8-9 through 8-12.

Software cannot directly load or modify the tags in the tag register. The FLDENV and FRSTOR instructions load an image of the tag register into the x87 FPU; however, the x87 FPU uses those tag values only to determine if the data registers are empty (11B) or non-empty (00B, 01B, or 10B).

If the tag register image indicates that a data register is empty, the tag in the tag register for that data register is marked empty (11B); if the tag register image indicates that the data register is non-empty, the x87 FPU reads the actual value in the data register and sets the tag for the register accordingly. This action prevents a program from setting the values in the tag register to incorrectly represent the actual contents of non-empty data registers.

### 8.1.8 x87 FPU Instruction and Data (Operand) Pointers

The x87 FPU stores pointers to the instruction and data (operand) for the last non-control instruction executed. These are the x87 FPU instruction pointer and x87 FPU data (operand) pointers; software can save these pointers to provide state information for exception handlers. The pointers are illustrated in Figure 8-1 (the figure illustrates the pointers as used outside 64-bit mode; see below).

Note that the value in the x87 FPU data pointer is always a pointer to a memory operand. If the last non-control instruction that was executed did not have a memory operand, the value in the data pointer is undefined (reserved). If CPUID.(EAX=07H, ECX=0H):EBX[bit 6] = 1, the data pointer is updated only for x87 non-control instructions that incur unmasked x87 exceptions.

The contents of the x87 FPU instruction and data pointers remain unchanged when any of the following instructions are executed: FCLEX/FNCLEX, FLDCW, FSTCW/FNSTCW, FSTSW/FNSTSW, FSTENV/FNSTENV, FLDENV, and WAIT/FWAIT.

For all the x87 FPUs and Numeric Processor Extensions (NPXs) except the 8087, the x87 FPU instruction pointer points to any prefixes that preceded the instruction. For the 8087, the x87 FPU instruction pointer points only to the actual opcode.

The x87 FPU instruction and data pointers each consists of an offset and a segment selector:

- The x87 FPU Instruction Pointer Offset (FIP) comprises 64 bits on processors that support IA-32e mode; on other processors, it offset comprises 32 bits.
- The x87 FPU Instruction Pointer Selector (FCS) comprises 16 bits.
- The x87 FPU Data Pointer Offset (FDP) comprises 64 bits on processors that support IA-32e mode; on other processors, it offset comprises 32 bits.
- The x87 FPU Data Pointer Selector (FDS) comprises 16 bits.

The pointers are accessed by the FINIT/FNINIT, FLDENV, FRSTOR, FSAVE/FNSAVE, FSTENV/FNSTENV, FXRSTOR, FXSAVE, XRSTOR, XSAVE, and XSAVEOPT instructions as follows:

- FINIT/FNINIT. Each instruction clears FIP, FCS, FDP, and FDS.
- FLDENV, FRSTOR. These instructions use the memory formats given in Figures 8-9 through 8-12:
  - For each of FIP and FDP, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
  - If CR0.PE = 1, each instruction loads FCS and FDS from memory; otherwise, it clears them.
- FSAVE/FNSAVE, FSTENV/FNSTENV. These instructions use the memory formats given in Figures 8-9 through 8-12.
  - Each instruction saves the lower 32 bits of each FIP and FDP into memory. the upper 32 bits are not saved.
  - If CR0.PE = 1, each instruction saves FCS and FDS into memory. If CPUID.(EAX=07H, ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS; it saves each as 0000H.
PROGRAMMING WITH THE X87 FPU

— After saving these data into memory, FSAVE/FNSAVE clears FIP, FCS, FDP, and FDS.

• FXRSTOR, XRSTOR. These instructions load data from a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-43, 3-46, and 3-47 in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

— Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
  • For each of FIP and FDP, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
  • Each instruction loads FCS and FDS from memory.

— In 64-bit mode with REX.W = 1, the instructions operate as follows:
  • Each instruction loads FIP and FDP from memory.
  • Each instruction clears FCS and FDS.

• FXSAVE, XSAVE, and XSAVEOPT. These instructions store data into a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-43, 3-46, and 3-47 in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

— Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
  • Each instruction saves the lower 32 bits of each of FIP and FDP into memory. The upper 32 bits are not saved.
  • Each instruction saves FCS and FDS into memory. If CPUID.(EAX=07H, ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS; it saves each as 0000H.

— In 64-bit mode with REX.W = 1, each instruction saves FIP and FDP into memory. FCS and FDS are not saved.

8.1.9 Last Instruction Opcode

The x87 FPU stores in the 11-bit x87 FPU opcode register (FOP) the opcode of the last x87 non-control instruction executed that incurred an unmasked x87 exception. (This information provides state information for exception handlers.) Only the first and second opcode bytes (after all prefixes) are stored in the x87 FPU opcode register. Figure 8-8 shows the encoding of these two bytes. Since the upper 5 bits of the first opcode byte are the same for all floating-point opcodes (11011B), only the lower 3 bits of this byte are stored in the opcode register.

8.1.9.1 Fopcode Compatibility Sub-mode

Some Pentium 4 and Intel Xeon processors provide program control over the value stored into FOP. Here, bit 2 of the IA32_MISC_ENABLE MSR enables (set) or disables (clear) the fopcode compatibility mode.

If fopcode compatibility mode is enabled, FOP is defined as it had been in previous IA-32 implementations, as the opcode of the last x87 non-control instruction executed (even if that instruction did not incur an unmasked x87 exception).
The fopcode compatibility mode should be enabled only when x87 FPU floating-point exception handlers are designed to use the fopcode to analyze program performance or restart a program after an exception has been handled.

More recent Intel 64 processors do not support fopcode compatibility mode and do not allow software to set bit 2 of the IA32_MISC_ENABLE MSR.

### 8.1.10 Saving the x87 FPU State with FSTENV/FNSTENV and FSAVE/FNSAVE

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions store x87 FPU state information in memory for use by exception handlers and other system and application software. The FSTENV/FNSTENV instruction saves the contents of the status, control, tag, x87 FPU instruction pointer, x87 FPU data pointer, and opcode registers. The FSAVE/FNSAVE instruction stores that information plus the contents of the x87 FPU data registers. Note that the FSAVE/FNSAVE instruction also initializes the x87 FPU to default values (just as the FINIT/FNINIT instruction does) after it has saved the original state of the x87 FPU.

The manner in which this information is stored in memory depends on the operating mode of the processor (protected mode or real-address mode) and on the operand-size attribute in effect (32-bit or 16-bit). See Figures 8-9 through 8-12. In virtual-8086 mode or SMM, the real-address mode formats shown in Figure 8-12 is used. See Chapter 32, “System Management Mode,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for information on using the x87 FPU while in SMM.

The FLDENV and FRSTOR instructions allow x87 FPU state information to be loaded from memory into the x87 FPU. Here, the FLDENV instruction loads only the status, control, tag, x87 FPU instruction pointer, x87 FPU data pointer, and opcode registers, and the FRSTOR instruction loads all the x87 FPU registers, including the x87 FPU stack registers.

![Figure 8-8. Contents of x87 FPU Opcode Registers](image)

The 32-bit protected mode format is shown in Figure 8-9. For instructions that also store x87 FPU data registers, the eight 80-bit registers (R0-R7) follow the above structure in sequence.

![Figure 8-9. Protected Mode x87 FPU State Image in Memory, 32-Bit Format](image)
8.1.11 Saving the x87 FPU State with FXSAVE

The FXSAVE and FXRSTOR instructions save and restore, respectively, the x87 FPU state along with the state of the XMM registers and the MXCSR register. Using the FXSAVE instruction to save the x87 FPU state has two benefits: (1) FXSAVE executes faster than FSAVE, and (2) FXSAVE saves the entire x87 FPU, MMX, and XMM state in one operation. See Section 10.5, “FXSAVE and FXRSTOR Instructions,” for additional information about these instructions.
8.2 X87 FPU DATA TYPES

The x87 FPU recognizes and operates on the following seven data types (see Figures 8-13): single precision floating-point, double precision floating-point, double extended precision floating-point, signed word integer, signed doubleword integer, signed quadword integer, and packed BCD decimal integers.

For detailed information about these data types, see Section 4.2.2, “Floating-Point Data Types,” Section 4.2.1.2, “Signed Integers,” and Section 4.7, “BCD and Packed BCD Integers.”

With the exception of the 80-bit double extended precision floating-point format, all of these data types exist in memory only. When they are loaded into x87 FPU data registers, they are converted into double extended precision floating-point format and operated on in that format.

Denormal values are also supported in each of the floating-point types, as required by IEEE Standard 754. When a denormal number in single precision or double precision floating-point format is used as a source operand and the denormal exception is masked, the x87 FPU automatically normalizes the number when it is converted to double extended precision format.

When stored in memory, the least significant byte of an x87 FPU data-type value is stored at the initial address specified for the value. Successive bytes from the value are then stored in successively higher addresses in memory. The floating-point instructions load and store memory operands using only the initial address of the operand.

As a general rule, values should be stored in memory in double precision format. This format provides sufficient range and precision to return correct results with a minimum of programmer attention. The single precision format is useful for debugging algorithms, because rounding problems will manifest themselves more quickly in this format. The double extended precision format is normally reserved for holding intermediate results in the x87 FPU registers and constants. Its extra length is designed to shield final results from the effects of rounding and overflow/underflow in intermediate calculations. However, when an application requires the maximum range and precision of the x87 FPU (for data storage, computations, and results), values can be stored in memory in double extended precision format.
8.2.1 Indefinites

For each x87 FPU data type, one unique encoding is reserved for representing the special value indefinite. The x87 FPU produces indefinite values as responses to some masked floating-point invalid-operation exceptions. See Tables 4-1, 4-3, and 4-4 for the encoding of the integer indefinite, QNaN floating-point indefinite, and packed BCD integer indefinite, respectively.

The binary integer encoding 100..00B represents either of two things, depending on the circumstances of its use:

- The largest negative number supported by the format (–2^{15}, –2^{31}, or –2^{63}).
- The integer indefinite value.

If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the x87 FPU interprets it as the largest negative number representable in the format being used. If the x87 FPU detects an invalid operation when storing an integer value in memory with an FIST/FISTP instruction and the invalid-operation exception is masked, the x87 FPU stores the integer indefinite encoding in the destination operand as a masked response to the exception. In situations where the origin of a value with this encoding may be ambiguous, the invalid-operation exception flag can be examined to see if the value was produced as a response to an exception.

8.2.2 Unsupported Double Extended Precision Floating-Point Encodings and Pseudo-Denormals

The double extended precision floating-point format permits many encodings that do not fall into any of the categories shown in Table 4-3. Table 8-3 shows these unsupported encodings. Some of these encodings were supported by the Intel 287 math coprocessor; however, most of them are not supported by the Intel 387 math coprocessor and later IA-32 processors. These encodings are no longer supported due to changes made in the final version of IEEE Standard 754 that eliminated these encodings.

Specifically, the categories of encodings formerly known as pseudo-NaNs, pseudo-infinities, and un-normal numbers are not supported and should not be used as operand values. The Intel 387 math coprocessor and later IA-32 processors generate an invalid-operation exception when these encodings are encountered as operands.

Beginning with the Intel 387 math coprocessor, the encodings formerly known as pseudo-denormal numbers are not generated by IA-32 processors. When encountered as operands, however, they are handled correctly, considering the biased exponent as 1 (and the unbiased exponent as -16382); that is, they are treated as denormals and a denormal exception is generated. Pseudo-denormal numbers should not be used as operand values. They are supported by current IA-32 processors (as described here) to support legacy code.

**Table 8-3. Unsupported Double Extended Precision Floating-Point Encodings and Pseudo-Denormals**

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Biased Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>11.11</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
<td>Fraction</td>
</tr>
<tr>
<td>Positive Pseudo-NaNs</td>
<td>0</td>
<td>11.11</td>
<td>0</td>
</tr>
<tr>
<td>Quiet</td>
<td>.</td>
<td>11.11</td>
<td>11.11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>11.11</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11.11</td>
<td>10.00</td>
</tr>
<tr>
<td>Signaling</td>
<td>0</td>
<td>11.11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>11.11</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11.11</td>
<td>01.11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>11.11</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11.11</td>
<td>00.01</td>
</tr>
<tr>
<td>Positive Floating-Point</td>
<td>0</td>
<td>11.11</td>
<td>0</td>
</tr>
<tr>
<td>Pseudo-infinity</td>
<td>0</td>
<td>11.10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>11.11</td>
<td>11.11</td>
</tr>
<tr>
<td>Unnormals</td>
<td>0</td>
<td>00.01</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>00.01</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00.00</td>
<td>00.00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>00.00</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00.00</td>
<td>00.00</td>
</tr>
<tr>
<td>Pseudo-denormals</td>
<td>.</td>
<td>00.00</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>00.00</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00.00</td>
<td>11.11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>00.00</td>
<td>.</td>
</tr>
</tbody>
</table>
8.3 X87 FPU INSTRUCTION SET

The floating-point instructions that the X87 FPU supports can be grouped into six functional categories:

- Data transfer instructions.
- Basic arithmetic instructions.
- Comparison instructions.
- Transcendental instructions.
- Load constant instructions.
- x87 FPU control instructions.

See Section 5.2, "x87 FPU Instructions," for a list of the floating-point instructions by category.

The following section briefly describes the instructions in each category. Detailed descriptions of the floating-point instructions are given in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

8.3.1 Escape (ESC) Instructions

All of the instructions in the X87 FPU instruction set fall into a class of instructions known as escape (ESC) instructions. All of these instructions have a common opcode format, where the first byte of the opcode is one of the numbers from D8H through DFH.

8.3.2 x87 FPU Instruction Operands

Most floating-point instructions require one or two operands, located on the X87 FPU data-register stack or in memory. (None of the floating-point instructions accept immediate operands.)

When an operand is located in a data register, it is referenced relative to the ST(0) register (the register at the top of the register stack), rather than by a physical register number. Often the ST(0) register is an implied operand.

Operands in memory can be referenced using the same operand addressing methods described in Section 3.7, "Operand Addressing."

8.3.3 Data Transfer Instructions

The data transfer instructions (see Table 8-4) perform the following operations:

<table>
<thead>
<tr>
<th>Negative Floating-Point</th>
<th>Pseudo-denormals</th>
<th>1</th>
<th>00..00</th>
<th>1</th>
<th>11..11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>00..00</td>
<td>1</td>
<td>00..00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unnormals</th>
<th></th>
<th>1</th>
<th>11..10</th>
<th>0</th>
<th>11..01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>00..01</td>
<td>1</td>
<td>00..00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pseudo-infinity</th>
<th></th>
<th>1</th>
<th>11..11</th>
<th>0</th>
<th>00..00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>11..11</td>
<td>1</td>
<td>11..11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Negative Pseudo-NaNs</th>
<th></th>
<th>1</th>
<th>11..11</th>
<th>0</th>
<th>01..11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signaling</td>
<td></td>
<td>.</td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>11..11</td>
<td>1</td>
<td>00..01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quiet</th>
<th></th>
<th>1</th>
<th>11..11</th>
<th>0</th>
<th>11..11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>.</td>
<td></td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>11..11</td>
<td>1</td>
<td>10..00</td>
</tr>
</tbody>
</table>

Table 8-3. Unsupported Double Extended Precision Floating-Point Encodings and Pseudo-Denormals (Contd.)
• Load a floating-point, integer, or packed BCD operand from memory into the ST(0) register.
• Store the value in an ST(0) register to memory in floating-point, integer, or packed BCD format.
• Move values between registers in the x87 FPU register stack.

The FLD (load floating-point) instruction pushes a floating-point operand from memory onto the top of the x87 FPU data-register stack. If the operand is in single precision or double precision floating-point format, it is automatically converted to double extended precision floating-point format. This instruction can also be used to push the value in a selected x87 FPU data register onto the top of the register stack.

The FILD (load integer) instruction converts an integer operand in memory into double extended precision floating-point format and pushes the value onto the top of the register stack. The FBLD (load packed decimal) instruction performs the same load operation for a packed BCD operand in memory.

Table 8-4. Data Transfer Instructions

<table>
<thead>
<tr>
<th>Floating-Point</th>
<th>Integer</th>
<th>Packed Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD</td>
<td>Load Floating-Point</td>
<td>FILD</td>
</tr>
<tr>
<td>FST</td>
<td>Store Floating-Point</td>
<td>FIST</td>
</tr>
<tr>
<td>FSTP</td>
<td>Store Floating-Point and Pop</td>
<td>FISTP</td>
</tr>
<tr>
<td>FXCH</td>
<td>Exchange Register Contents</td>
<td>FBSTP</td>
</tr>
<tr>
<td>FCMOVcc</td>
<td>Conditional Move</td>
<td>FBLD</td>
</tr>
</tbody>
</table>

The FST (store floating-point) and FIST (store integer) instructions store the value in register ST(0) in memory in the destination format (floating-point or integer, respectively). Again, the format conversion is carried out automatically.

The FSTP (store floating-point and pop), FISTP (store integer and pop), and FBSTP (store packed decimal and pop) instructions store the value in the ST(0) registers into memory in the destination format (floating-point, integer, or packed BCD), then performs a pop operation on the register stack. A pop operation causes the ST(0) register to be marked empty and the stack pointer (TOP) in the x87 FPU control work to be incremented by 1. The FSTP instruction can also be used to copy the value in the ST(0) register to another x87 FPU register [ST(i)].

The FXCH (exchange register contents) instruction exchanges the value in a selected register in the stack [ST(i)] with the value in ST(0).

The FCMOVcc (conditional move) instructions move the value in a selected register in the stack [ST(i)] to register ST(0) if a condition specified with a condition code (cc) is satisfied (see Table 8-5). The condition being tested for is represented by the status flags in the EFLAGS register. The condition code mnemonics are appended to the letters “FCMOV” to form the mnemonic for a FCMOVcc instruction.

Table 8-5. Floating-Point Conditional Move Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Status Flag States</th>
<th>Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMOVB</td>
<td>CF=1</td>
<td>Below</td>
</tr>
<tr>
<td>FCMOVNB</td>
<td>CF=0</td>
<td>Not below</td>
</tr>
<tr>
<td>FCMOVE</td>
<td>ZF=1</td>
<td>Equal</td>
</tr>
<tr>
<td>FCMOVNE</td>
<td>ZF=0</td>
<td>Not equal</td>
</tr>
</tbody>
</table>
Like the CMOV \( cc \) instructions, the FCMOV \( cc \) instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

Software can check if the FCMOV\( cc \) instructions are supported by checking the processor’s feature information with the CPUID instruction.

### 8.3.4 Load Constant Instructions

The following instructions push commonly used constants onto the top \([ST(0)]\) of the x87 FPU register stack:

- **FLDZ** Load \(+0.0\).
- **FLD1** Load \(+1.0\).
- **FLDPI** Load \(\pi\).
- **FLDL2T** Load \(\log_2 10\).
- **FLDL2E** Load \(\log_2 e\).
- **FLDLG2** Load \(\log_{10} 2\).
- **FLDLN2** Load \(\log_e 2\).

The constant values have full double extended precision floating-point precision (64 bits) and are accurate to approximately 19 decimal digits. They are stored internally in a format more precise than double extended precision floating-point. When loading the constant, the x87 FPU rounds the more precise internal constant according to the RC (rounding control) field of the x87 FPU control word. The inexact-result exception \(\#P\) is not generated as a result of this rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up. See Section 8.3.8, “Approximation of \(\pi\),” for information on the \(\pi\) constant.

### 8.3.5 Basic Arithmetic Instructions

The following floating-point instructions perform basic arithmetic operations on floating-point numbers. Where applicable, these instructions match IEEE Standard 754:

- **FADD/FADDP** Add floating-point.
- **FIADD** Add integer to floating-point.
- **FSUB/FSUBP** Subtract floating-point.
- **FISUB** Subtract integer from floating-point.
- **FSUBR/FSUBRP** Reverse subtract floating-point.
- **FISUBR** Reverse subtract floating-point from integer.
- **FMUL/FMULP** Multiply floating-point.
- **FIMUL** Multiply integer by floating-point.
- **FDIV/FDIVP** Divide floating-point.
- **FIDIV** Divide floating-point by integer.
- **FDIVR/FDIVRP** Reverse divide.
- **FIDIVR** Reverse divide integer by floating-point.
- **FABS** Absolute value.
- **FCHS** Change sign.

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Status Flag States</th>
<th>Condition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCMOVBE</td>
<td>CF=1 orZF=1</td>
<td>Below or equal</td>
</tr>
<tr>
<td>FCMOVNE</td>
<td>CF=0 orZF=0</td>
<td>Not below nor equal</td>
</tr>
<tr>
<td>FCMOVU</td>
<td>PF=1</td>
<td>Unordered</td>
</tr>
<tr>
<td>FCMOVNU</td>
<td>PF=0</td>
<td>Not unordered</td>
</tr>
</tbody>
</table>

Table 8-5. Floating-Point Conditional Move Instructions (Contd.)
PROGRAMMING WITH THE X87 FPU

FSQRT Square root.
FPREM Partial remainder.
FPREM1 IEEE partial remainder.
FRNDINT Round to integral value.
FXTRACT Extract exponent and significand.

The add, subtract, multiply, and divide instructions operate on the following types of operands:

- Two x87 FPU data registers.
- An x87 FPU data register and a floating-point or integer value in memory.

See Section 8.1.2, “x87 FPU Data Registers,” for a description of how operands are referenced on the data register stack.

Operands in memory can be in single precision floating-point, double precision floating-point, word-integer, or doubleword-integer format. They are converted to double extended precision floating-point format automatically.

Reverse versions of the subtract (FSUBR) and divide (FDIVR) instructions enable efficient coding. For example, the following options are available with the FSUB and FSUBR instructions for operating on values in a specified x87 FPU data register ST(i) and the ST(0) register:

**FSUB:**

\[
\begin{align*}
\text{ST}(0) & := \text{ST}(0) - \text{ST}(i) \\
\text{ST}(i) & := \text{ST}(i) - \text{ST}(0)
\end{align*}
\]

**FSUBR:**

\[
\begin{align*}
\text{ST}(0) & := \text{ST}(i) - \text{ST}(0) \\
\text{ST}(i) & := \text{ST}(0) - \text{ST}(i)
\end{align*}
\]

These instructions eliminate the need to exchange values between the ST(0) register and another x87 FPU register to perform a subtraction or division.

The pop versions of the add, subtract, multiply, and divide instructions offer the option of popping the x87 FPU register stack following the arithmetic operation. These instructions operate on values in the ST(i) and ST(0) registers, store the result in the ST(i) register, and pop the ST(0) register.

The FPREM instruction computes the remainder from the division of two operands in the manner used by the Intel 8087 and Intel 287 math coprocessors; the FPREM1 instruction computes the remainder in the manner specified in IEEE Standard 754.

The FSQRT instruction computes the square root of the source operand.

The FRNDINT instruction returns a floating-point value that is the integral value closest to the source value in the direction of the rounding mode specified in the RC field of the x87 FPU control word.

The FABS, FCHS, and FXTRACT instructions perform convenient arithmetic operations. The FABS instruction produces the absolute value of the source operand. The FCHS instruction changes the sign of the source operand. The FXTRACT instruction separates the source operand into its exponent and fraction and stores each value in a register in floating-point format.

### 8.3.6 Comparison and Classification Instructions

The following instructions compare or classify floating-point values:

**FCOM/FCOMP/FCOMPP** Compare floating-point and set x87 FPU condition code flags.

**FUCOM/FUCOMP/FUCOMPP** Unordered compare floating-point and set x87 FPU condition code flags.

**FICOM/FICOMP** Compare integer and set x87 FPU condition code flags.

**FCOMI/FCOMIP** Compare floating-point and set EFLAGS status flags.

**FUCOMI/FUCOMIP** Unordered compare floating-point and set EFLAGS status flags.

**FTST** Test (compare floating-point with 0.0).
FXAM Examine.

Comparison of floating-point values differ from comparison of integers because floating-point values have four (rather than three) mutually exclusive relationships: less than, equal, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN or in an unsupported format. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have less than, equal, or greater than relationships with other floating-point values.

The FCOM, FCOMP, and FCOMPP instructions compare the value in register ST(0) with a floating-point source operand and set the condition code flags (C0, C2, and C3) in the x87 FPU status word according to the results (see Table 8-6).

If an unordered condition is detected (one or both of the values are NaNs or in an undefined format), a floating-point invalid-operation exception is generated.

The pop versions of the instruction pop the x87 FPU register stack once or twice after the comparison operation is complete.

The FUCOM, FUCOMP, and FUCOMPP instructions operate the same as the FCOM, FCOMP, and FCOMPP instructions. The only difference is that with the FUCOM, FUCOMP, and FUCOMPP instructions, if an unordered condition is detected because one or both of the operands are QNaNs, the floating-point invalid-operation exception is not generated.

The FICOM and FICOMP instructions also operate the same as the FCOM and FCOMP instructions, except that the source operand is an integer value in memory. The integer value is automatically converted into a double extended precision floating-point value prior to making the comparison. The FICOMP instruction pops the x87 FPU register stack following the comparison operation.

The FTST instruction performs the same operation as the FCOM instruction, except that the value in register ST(0) is always compared with the value 0.0.

The FCOMI and FCOMIP instructions were introduced into the IA-32 architecture in the P6 family processors. They perform the same comparison as the FCOM and FCOMP instructions, except that they set the status flags (ZF, PF, and CF) in the EFLAGS register to indicate the results of the comparison (see Table 8-7) instead of the x87 FPU condition code flags. The FCOMI and FCOMIP instructions allow condition branch instructions (Jcc) to be executed directly from the results of their comparison.

Software can check if the FCOMI and FCOMIP instructions are supported by checking the processor’s feature information with the CPUID instruction.

The FUCOMI and FUCOMIP instructions operate the same as the FCOMI and FCOMIP instructions, except that they do not generate a floating-point invalid-operation exception if the unordered condition is the result of one or both of the operands being a QNaN. The FCOMIP and FUCOMIP instructions pop the x87 FPU register stack following the comparison operation.

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; Source Operand</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; Source Operand</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) = Source Operand</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8-6. Setting of x87 FPU Condition Code Flags for Floating-Point Number Comparisons

<table>
<thead>
<tr>
<th>Comparison Results</th>
<th>ZF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 &gt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST0 &lt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST0 = ST(i)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8-7. Setting of EFLAGS Status Flags for Floating-Point Number Comparisons
Programming with the x87 FPU

The FXAM instruction determines the classification of the floating-point value in the ST(0) register (that is, whether the value is zero, a denormal number, a normal finite number, ±∞, a NaN, or an unsupported format) or that the register is empty. It sets the x87 FPU condition code flags to indicate the classification (see “FXAM—Examine” in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). It also sets the C1 flag to indicate the sign of the value.

8.3.6.1 Branching on the x87 FPU Condition Codes

The processor does not offer any control-flow instructions that branch on the setting of the condition code flags (C0, C2, and C3) in the x87 FPU status word. To branch on the state of these flags, the x87 FPU status word must first be moved to the AX register in the integer unit. The FSTSW AX (store status word) instruction can be used for this purpose. When these flags are in the AX register, the TEST instruction can be used to control conditional branching as follows:

1. Check for an unordered result. Use the TEST instruction to compare the contents of the AX register with the constant 0400H (see Table 8-8). This operation will clear the ZF flag in the EFLAGS register if the condition code flags indicate an unordered result; otherwise, the ZF flag will be set. The JNZ instruction can then be used to transfer control (if necessary) to a procedure for handling unordered operands.

2. Check ordered comparison result. Use the constants given in Table 8-8 in the TEST instruction to test for a less than, equal to, or greater than result, then use the corresponding conditional branch instruction to transfer program control to the appropriate procedure or section of code.

If a program or procedure has been thoroughly tested and it incorporates periodic checks for QNaN results, then it is not necessary to check for the unordered result every time a comparison is made.

See Section 8.1.4, “Branching and Conditional Moves on Condition Codes,” for another technique for branching on x87 FPU condition codes.

Some non-comparison x87 FPU instructions update the condition code flags in the x87 FPU status word. To ensure that the status word is not altered inadvertently, store it immediately following a comparison operation.

8.3.7 Trigonometric Instructions

The following instructions perform four common trigonometric functions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSIN</td>
<td>Sine</td>
</tr>
<tr>
<td>FCOS</td>
<td>Cosine</td>
</tr>
<tr>
<td>FSINCOS</td>
<td>Sine and cosine</td>
</tr>
<tr>
<td>FPTAN</td>
<td>Tangent</td>
</tr>
<tr>
<td>FPATAN</td>
<td>Arctangent</td>
</tr>
</tbody>
</table>

These instructions operate on the top one or two registers of the x87 FPU register stack and they return their results to the stack. The source operands for the FSIN, FCOS, FSINCOS, and FPTAN instructions must be given in radians; the source operand for the FPATAN instruction is given in rectangular coordinate units.

The FSINCOS instruction returns both the sine and the cosine of a source operand value. It operates faster than executing the FSIN and FCOS instructions in succession.
The FPATAN instruction computes the arctangent of ST(1) divided by ST(0), returning a result in radians. It is useful for converting rectangular coordinates to polar coordinates.

See Section 8.3.8, “Approximation of Pi,” and Section 8.3.10, “Transcendental Instruction Accuracy,” for information regarding the accuracy of these instructions.

8.3.8 Approximation of Pi

When the argument (source operand) of a trigonometric function is within the domain of the function, the argument is automatically reduced by the appropriate multiple of 2π through the same reduction mechanism used by the FPREM and FPREM1 instructions. The internal value of π (3.1415926...) that the x87 FPU uses for argument reduction and other computations, denoted as Pi in the expression below. The numerical value of Pi can be written as:

$$\pi = 0.f \times 2^{22}$$

where the fraction f is expressed in binary form as:

$$f = C90FDAA2 \ 2168C234 \ C$$

(The spaces in the fraction above indicate 32-bit boundaries.)

The internal approximation Pi of the value π has a 66 significant bits. Since the exact value of π represented in binary has the next 3 bits equal to 0, it means that Pi is the value of π rounded to nearest-even to 68 bits, and also the value of π rounded toward zero (truncated) to 69 bits.

However, accuracy problems may arise because this relatively short finite approximation Pi of the number π is used for calculating the reduced argument of the trigonometric function approximations in the implementations of FSIN, FCOS, FSINCOS, and FPTAN. Alternately, this means that FSIN (x), FCOS (x), and FPTAN (x) are really approximating the mathematical functions sin (x * π / Pi), cos (x * π / Pi), and tan (x * π / Pi), and not exactly sin (x), cos (x), and tan (x). (Note that FSINCOS is the equivalent of FSIN and FCOS combined together). The period of sin (x * π / Pi) for example is 2* Pi, and not 2π.

See also Section 8.3.10, “Transcendental Instruction Accuracy,” for more information on the accuracy of these functions.

8.3.9 Logarithmic, Exponential, and Scale

The following instructions provide two different logarithmic functions, an exponential function and a scale function:

- FYL2X Logarithm.
- FYL2XP1 Logarithm epsilon.
- F2XM1 Exponential.
- FSSCALE Scale.

The FYL2X and FYL2XP1 instructions perform two different base 2 logarithmic operations. The FYL2X instruction computes (y * log2x). This operation permits the calculation of the log of any base using the following equation:

$$\log_b x = (1 / \log_2 b) \times \log_2 x$$

The FYL2XP1 instruction computes (y * log2(x + 1)). This operation provides optimum accuracy for values of x that are close to 0.

The F2XM1 instruction computes (2^x − 1). This instruction only operates on source values in the range −1.0 to +1.0. The FSSCALE instruction multiplies the source operand by a power of 2.

8.3.10 Transcendental Instruction Accuracy

New transcendental instruction algorithms were incorporated into the IA-32 architecture beginning with the Pentium processors. These new algorithms (used in transcendental instructions FSIN, FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) allow a higher level of accuracy than was possible in earlier IA-32 processors.
and x87 math coprocessors. The accuracy of these instructions is measured in terms of units in the last place (ulp). For a given argument \( x \), let \( f(x) \) and \( F(x) \) be the correct and computed (approximate) function values, respectively. The error in ulps is defined to be:

\[
\text{error} = \frac{|f(x) - F(x)|}{2^k - 63}
\]

where \( k \) is an integer such that:

\[ 1 \leq 2^{-k} f(x) < 2. \]

With the Pentium processor and later IA-32 processors, the worst case error on transcendental functions is less than 1 ulp when rounding to the nearest (even) and less than 1.5 ulps when rounding in other modes. The functions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

However, for FSIN, FCOS, FSINCOS, and FPTAN which approximate periodic trigonometric functions, the previous statement about maximum ulp errors is true only when these instructions are applied to reduced argument (see Section 8.3.8, ”Approximation of Pi”). This is due to the fact that only 66 significant bits are retained in the finite approximation \( \pi \) of the number \( \pi \) (3.1415926...), used internally for calculating the reduced argument in FSIN, FCOS, FSINCOS, and FPTAN. This approximation of \( \pi \) is not always sufficiently accurate for good argument reduction.

For single precision, the argument of FSIN, FCOS, FSINCOS, and FPTAN must exceed 200,000 radians in order for the error of the result to exceed 1 ulp when rounding to the nearest (even), or 1.5 ulps when rounding in other (directed) rounding modes.

For double and double-extended precision, the ulp errors will grow above these thresholds for arguments much smaller in magnitude. The ulp errors increase significantly when the argument approaches the value of \( \pi \) (or \( \pi/2 \)) for FSIN, and when it approaches \( \pi/2 \) (or \( \pi/2 \)) for FCOS, FSINCOS, and FPTAN.

For all three IEEE precisions supported (32-bit single precision, 64-bit double precision, and 80-bit double-extended precision), applying FSIN, FCOS, FSINCOS, or FPTAN to arguments larger than a certain value can lead to reduced arguments (calculated internally) that are inaccurate or even very inaccurate in some cases. This leads to equally inaccurate approximations of the corresponding mathematical functions. In particular, arguments that are close to certain values will lose significance when reduced, leading to increased relative (and ulp) errors in the results of FSIN, FCOS, FSINCOS, and FPTAN. These values are:

- Any non-zero multiple of \( \pi \) for FSIN.
- Any multiple of \( \pi \), plus \( \pi/2 \) for FCOS.
- Any non-zero multiple of \( \pi/2 \) for FSINCOS and FPTAN.

If the arguments passed to FSIN, FCOS, FSINCOS, and FPTAN are not close to these values then even the finite approximation \( \pi \) used internally for argument reduction will allow for results that have good accuracy.

Therefore, in order to avoid such errors it is recommended to perform accurate argument reduction in software, and to apply FSIN, FCOS, FSINCOS, and FPTAN to reduced arguments only. Regardless of the target precision (single, double, or double-extended), it is safe to reduce the argument to a value smaller in absolute value than about \( 3\pi/4 \) for FSIN, and smaller than about \( 3\pi/8 \) for FCOS, FSINCOS, and FPTAN.

The thresholds shown above are not exact. For example, accuracy measurements show that the double-extended precision result of FSIN will not have errors larger than 0.72 ulp for \( |x| < 2.82 \) (so \( |x| < 3\pi/4 \) will ensure good accuracy, as \( 3\pi/4 < 2.82 \)). On the same interval, double precision results from FSIN will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

Likewise, the double-extended precision result of FCOS will not have errors larger than 0.82 ulp for \( |x| < 1.31 \) (so \( |x| < 3\pi/8 \) will ensure good accuracy, as \( 3\pi/8 < 1.31 \)). On the same interval, double precision results from FCOS will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

FSINCOS behaves similarly to FSIN and FCOS, combined as a pair.
Finally, the double-extended precision result of FPTAN will not have errors larger than 0.78 ulp for \(|x| < 1.25\) (so \(|x| < 3\pi/8\) will ensure good accuracy, as \(3\pi/8 < 1.25\)). On the same interval, double precision results from FPTAN will have errors at most slightly larger than 0.5 ulp, and single precision results will be correctly rounded in the vast majority of cases.

A recommended alternative in order to avoid the accuracy issues that might be caused by FSIN, FCOS, FSINCOS, and FPTAN, is to use good quality mathematical library implementations of the sin, cos, sincos, and tan functions, for example those from the Intel® Math Library available in the Intel® Compiler.

The instructions FYL2X and FYL2XP1 are two operand instructions and are guaranteed to be within 1 ulp only when \(y\) equals 1. When \(y\) is not equal to 1, the maximum ulp error is always within 1.35 ulps in round to nearest mode. (For the two operand functions, monotonicity was proved by holding one of the operands constant.)

### 8.3.11 x87 FPU Control Instructions

The following instructions control the state and modes of operation of the x87 FPU. They also allow the status of the x87 FPU to be examined:

- **FINIT/FNINIT**: Initialize x87 FPU.
- **FLDCW**: Load x87 FPU control word.
- **FSTCW/FNSTCW**: Store x87 FPU control word.
- **FSTSW/FNSTSW**: Store x87 FPU status word.
- **FCLEX/FNCLEX**: Clear x87 FPU exception flags.
- **FLDENV**: Load x87 FPU environment.
- **FSTENV/FNSTENV**: Store x87 FPU environment.
- **FRSTOR**: Restore x87 FPU state.
- **FSAVE/FNSAVE**: Save x87 FPU state.
- **FINCSTP**: Increment x87 FPU register stack pointer.
- **FDECSTP**: Decrement x87 FPU register stack pointer.
- **FFREE**: Free x87 FPU register.
- **FNOP**: No operation.
- **WAIT/FWAIT**: Check for and handle pending unmasked x87 FPU exceptions.

The FINIT/FNINIT instructions initialize the x87 FPU and its internal registers to default values.

The FLDCW instructions loads the x87 FPU control word register with a value from memory. The FSTCW/FNSTCW and FSTSW/FNSTSW instructions store the x87 FPU control and status words, respectively, in memory (or for an FSTSW/FNSTSW instruction in a general-purpose register).

The FSTENV/FNSTENV and FSAVE/FNSAVE instructions save the x87 FPU environment and state, respectively, in memory. The x87 FPU environment includes all the x87 FPU’s control and status registers; the x87 FPU state includes the x87 FPU environment and the data registers in the x87 FPU register stack. (The FSAVE/FNSAVE instruction also initializes the x87 FPU to default values, like the FINIT/FNINIT instruction, after it saves the original state of the x87 FPU.)

The FLDENV and FRSTOR instructions load the x87 FPU environment and state, respectively, from memory into the x87 FPU. These instructions are commonly used when switching tasks or contexts.

The WAIT/FWAIT instructions are synchronization instructions. (They are actually mnemonics for the same opcode.) These instructions check the x87 FPU status word for pending unmasked x87 FPU exceptions. If any pending unmasked x87 FPU exceptions are found, they are handled before the processor resumes execution of the instructions (integer, floating-point, or system instruction) in the instruction stream. The WAIT/FWAIT instructions are provided to allow synchronization of instruction execution between the x87 FPU and the processor’s integer unit. See Section 8.6, "x87 FPU Exception Synchronization," for more information on the use of the WAIT/FWAIT instructions.
8.3.12 Waiting vs. Non-waiting Instructions

All of the x87 FPU instructions except a few special control instructions perform a wait operation (similar to the WAIT/FWAIT instructions), to check for and handle pending unmasked x87 FPU floating-point exceptions, before they perform their primary operation (such as adding two floating-point numbers). These instructions are called waiting instructions. Some of the x87 FPU control instructions, such as FSTSW/FNSTSW, have both a waiting and a non-waiting version. The waiting version (with the "F" prefix) executes a wait operation before it performs its primary operation; whereas, the non-waiting version (with the "FN" prefix) ignores pending unmasked exceptions.

Non-waiting instructions allow software to save the current x87 FPU state without first handling pending exceptions or to reset or reinitialize the x87 FPU without regard for pending exceptions.

NOTES

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for a non-waiting instruction to be interrupted prior to being executed to handle a pending x87 FPU exception.

When operating a P6 family, Pentium 4, or Intel Xeon processor in MS-DOS compatibility mode, non-waiting instructions can not be interrupted in this way.

8.3.13 Unsupported x87 FPU Instructions

The Intel 8087 instructions FENI and FDISI and the Intel 287 math coprocessor instruction FSETPM perform no function in the Intel 387 math coprocessor and later IA-32 processors. If these opcodes are detected in the instruction stream, the x87 FPU performs no specific operation and no internal x87 FPU states are affected.

8.4 X87 FPU FLOATING-POINT EXCEPTION HANDLING

The x87 FPU detects the six classes of exception conditions described in Section 4.9, "Overview of Floating-Point Exceptions":

- Invalid operation (#I), with two subclasses:
  - Stack overflow or underflow (#IS).
  - Invalid arithmetic operation (#IA).
- Denormalized operand (#D).
- Divide-by-zero (#Z).
- Numeric overflow (#O).
- Numeric underflow (#U).
- Inexact result (precision) (#P).

Each of the six exception classes has a corresponding flag bit in the x87 FPU status word and a mask bit in the x87 FPU control word (see Section 8.1.3, "x87 FPU Status Register," and Section 8.1.5, "x87 FPU Control Word," respectively). In addition, the exception summary (ES) flag in the status word indicates when one or more unmasked exceptions has been detected. The stack fault (SF) flag (also in the status word) distinguishes between the two types of invalid-operation exceptions.

The mask bits can be set with FLDCW, FRSTOR, or FXRSTOR; they can be read with either FSTCW/FNSTCW, FSAVE/FNSAVE, or FXSAVE. The flag bits can be read with the FSTSW/FNSTSW, FSAVE/FNSAVE, or FXSAVE instruction.

NOTE

Section 4.9.1, "Floating-Point Exception Conditions," provides a general overview of how the IA-32 processor detects and handles the various classes of floating-point exceptions. This information pertains to the x87 FPU as well as the Intel SSE, SSE2, and SSE3 instructions.
The following sections give specific information about how the x87 FPU handles floating-point exceptions that are unique to the x87 FPU.

8.4.1 Arithmetic vs. Non-arithmetic Instructions

When dealing with floating-point exceptions, it is useful to distinguish between arithmetic instructions and non-arithmetic instructions. Non-arithmetic instructions have no operands or do not make substantial changes to their operands. Arithmetic instructions do make significant changes to their operands; in particular, they make changes that could result in floating-point exceptions being signaled. Table 8-9 lists the non-arithmetic and arithmetic instructions. It should be noted that some non-arithmetic instructions can signal a floating-point stack (fault) exception, but this exception is not the result of an operation on an operand.

<table>
<thead>
<tr>
<th>Non-arithmetic Instructions</th>
<th>Arithmetic Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABS</td>
<td>F2XM1</td>
</tr>
<tr>
<td>FCHS</td>
<td>FADD/FADDP</td>
</tr>
<tr>
<td>FCLEX</td>
<td>FBLD</td>
</tr>
<tr>
<td>FDECASTP</td>
<td>FBSTP</td>
</tr>
<tr>
<td>FFREE</td>
<td>FCOM/FCOMP/FCOMPP</td>
</tr>
<tr>
<td>FINCSTP</td>
<td>FCOS</td>
</tr>
<tr>
<td>FINIT/FNINIT</td>
<td>FDIV/FDIVP/FDIVR/FDIVRP</td>
</tr>
<tr>
<td>FLD (register-to-register)</td>
<td>FIADD</td>
</tr>
<tr>
<td>FLD (extended format from memory)</td>
<td>FICOM/FICOMP</td>
</tr>
<tr>
<td>FLD constant</td>
<td>FIDIV/FIDIVR</td>
</tr>
<tr>
<td>FDCW</td>
<td>FILD</td>
</tr>
<tr>
<td>FLDENV</td>
<td>FIMUL</td>
</tr>
<tr>
<td>FNOP</td>
<td>FIST/FISTP¹</td>
</tr>
<tr>
<td>FRSTOR</td>
<td>FISUB/FISUBR</td>
</tr>
<tr>
<td>FSAVE/FNSAVE</td>
<td>FLD (single and double)</td>
</tr>
<tr>
<td>FST/FSTP (register-to-register)</td>
<td>FMUL/FMULP</td>
</tr>
<tr>
<td>FSTP (extended format to memory)</td>
<td>FPATAN</td>
</tr>
<tr>
<td>FSTCW/FNSTCW</td>
<td>FPREM/FPREM1</td>
</tr>
<tr>
<td>FSTENV/FNSTENV</td>
<td>FPTAN</td>
</tr>
<tr>
<td>FSTSW/FNSTSW</td>
<td>FRNDINT</td>
</tr>
<tr>
<td>WAIT/FWAIT</td>
<td>FSSCALE</td>
</tr>
<tr>
<td>FXAM</td>
<td>FSIN</td>
</tr>
<tr>
<td>FXCH</td>
<td>FSINCOS</td>
</tr>
<tr>
<td>FXTRACT</td>
<td>FSQRT</td>
</tr>
<tr>
<td>FST/FSTP (single and double)</td>
<td>FSUB/FSUBP/FSUBR/FSUBRP</td>
</tr>
<tr>
<td>FTST</td>
<td>FUCOM/FUCOMP/FUCOMPP</td>
</tr>
<tr>
<td>FXTRACT</td>
<td>FYL2X/FYL2XP1</td>
</tr>
</tbody>
</table>
The following sections describe the various conditions that cause a floating-point exception to be generated by the x87 FPU and the masked response of the x87 FPU when these conditions are detected. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, lists the floating-point exceptions that can be signaled for each floating-point instruction.

See Section 4.9.2, "Floating-Point Exception Priority," for a description of the rules for exception precedence when more than one floating-point exception condition is detected for an instruction.

### 8.5.1 Invalid Operation Exception

The floating-point invalid-operation exception occurs in response to two sub-classes of operations:

- Stack overflow or underflow (#IS).
- Invalid arithmetic operand (#IA).

The flag for this exception (IE) is bit 0 of the x87 FPU status word, and the mask bit (IM) is bit 0 of the x87 FPU control word. The stack fault flag (SF) of the x87 FPU status word indicates the type of operation that caused the exception. When the SF flag is set to 1, a stack operation has resulted in stack overflow or underflow; when the flag is cleared to 0, an arithmetic instruction has encountered an invalid operand. Note that the x87 FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition. As a result, the state of the SF flag can be 1 following an invalid-arithmetic-operation exception, if it was not cleared from the last time a stack overflow or underflow condition occurred. See Section 8.1.3.4, “Stack Fault Flag,” for more information about the SF flag.

#### 8.5.1.1 Stack Overflow or Underflow Exception (#IS)

The x87 FPU tag word keeps track of the contents of the registers in the x87 FPU register stack (see Section 8.1.7, "x87 FPU Tag Word"). It then uses this information to detect two different types of stack faults:

- **Stack overflow** — An instruction attempts to load a non-empty x87 FPU register from memory. A non-empty register is defined as a register containing a zero (tag value of 01), a valid value (tag value of 00), or a special value (tag value of 10).
- **Stack underflow** — An instruction references an empty x87 FPU register as a source operand, including attempting to write the contents of an empty register to memory. An empty register has a tag value of 11.

**NOTES**

The term stack overflow originates from the situation where the program has loaded (pushed) eight values from memory onto the x87 FPU register stack and the next value pushed on the stack causes a stack wraparound to a register that already contains a value.

The term stack underflow originates from the opposite situation. Here, a program has stored (popped) eight values from the x87 FPU register stack to memory and the next value popped from the stack causes stack wraparound to an empty register.

When the x87 FPU detects stack overflow or underflow, it sets the IE flag (bit 0) and the SF flag (bit 6) in the x87 FPU status word to 1. It then sets condition-code flag C1 (bit 9) in the x87 FPU status word to 1 if stack overflow occurred or to 0 if stack underflow occurred.
If the invalid-operation exception is masked, the x87 FPU returns the floating-point, integer, or packed decimal integer indefinite value to the destination operand, depending on the instruction being executed. This value overwrites the destination register or memory location specified by the instruction.

If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”) and the top-of-stack pointer (TOP) and source operands remain unchanged.

### 8.5.1.2 Invalid ArithmeticOperand Exception (#IA)

The x87 FPU is able to detect a variety of invalid arithmetic operations that can be coded in a program. These operations are listed in Table 8-10. (This list includes the invalid operations defined in IEEE Standard 754.)

When the x87 FPU detects an invalid arithmetic operand, it sets the IE flag (bit 0) in the x87 FPU status word to 1. If the invalid-operation exception is masked, the x87 FPU then returns an indefinite value or QNaN to the destination operand and/or sets the floating-point condition codes as shown in Table 8-10. If the invalid-operation exception is not masked, a software exception handler is invoked (see Section 8.7, “Handling x87 FPU Exceptions in Software”) and the top-of-stack pointer (TOP) and source operands remain unchanged.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Masked Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any arithmetic operation on an operand that is in an unsupported format.</td>
<td>Return the QNaN floating-point indefinite value to the destination operand.</td>
</tr>
<tr>
<td>Any arithmetic operation on a SNaN.</td>
<td>Return a QNaN to the destination operand (see Table 4-7).</td>
</tr>
<tr>
<td>Ordered compare and test operations: one or both operands are NaNs.</td>
<td>Set the condition code flags (C0, C2, and C3) in the x87 FPU status word or the CF, PF, and ZF flags in the EFLAGS register to 111B (not comparable).</td>
</tr>
<tr>
<td>Addition: operands are opposite-signed infinities. Subtraction: operands are like-signed infinities.</td>
<td>Return the QNaN floating-point indefinite value to the destination operand.</td>
</tr>
<tr>
<td>Division: $\infty$ by 0; 0 by $\infty$.</td>
<td>Return the QNaN floating-point indefinite value to the destination operand.</td>
</tr>
<tr>
<td>Remainder instructions FPREM, FPREM1: modulus (divisor) is 0 or dividend is $\infty$.</td>
<td>Return the QNaN floating-point indefinite; clear condition code flag C2 to 0.</td>
</tr>
<tr>
<td>Trigonometric instructions FCOS, FPTAN, FSIN, FSINCOS: source operand is $\infty$.</td>
<td>Return the QNaN floating-point indefinite; clear condition code flag C2 to 0.</td>
</tr>
<tr>
<td>FSQRT: negative operand (except FSQRT (–0) = –0); FYL2X: negative operand (except FYL2X (–0) = –$\infty$); FYL2XP1: operand more negative than –1.</td>
<td>Return the QNaN floating-point indefinite value to the destination operand.</td>
</tr>
<tr>
<td>FBSTP: Converted value cannot be represented in 18 decimal digits, or source value is an SNaN, QNaN, ±$\infty$, or in an unsupported format.</td>
<td>Store packed BCD integer indefinite value in the destination operand.</td>
</tr>
<tr>
<td>FIST/FISTP: Converted value exceeds representable integer range of the destination operand, or source value is an SNaN, QNaN, ±$\infty$, or in an unsupported format.</td>
<td>Store integer indefinite value in the destination operand.</td>
</tr>
<tr>
<td>FXCH: one or both registers are tagged empty.</td>
<td>Load empty registers with the QNaN floating-point indefinite value, then perform the exchange.</td>
</tr>
</tbody>
</table>

Normally, when one or both of the source operands is a QNaN (and neither is an SNaN or in an unsupported format), an invalid-operand exception is not generated. An exception to this rule is most of the compare instructions (such as the FCOM and FCOMI instructions) and the floating-point to integer conversion instructions
8.5.2 Denormal Operand Exception (#D)

The x87 FPU signals the denormal-operand exception under the following conditions:

- If an arithmetic instruction attempts to operate on a denormal operand (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers").
- If an attempt is made to load a denormal single precision or double precision floating-point value into an x87 FPU register. (If the denormal value being loaded is a double extended precision floating-point value, the denormal-operand exception is not reported.)

The flag (DE) for this exception is bit 1 of the x87 FPU status word, and the mask bit (DM) is bit 1 of the x87 FPU control word.

When a denormal-operand exception occurs and the exception is masked, the x87 FPU sets the DE flag, then proceeds with the instruction. The denormal operand in single- or double precision floating-point format is automatically normalized when converted to the double extended precision floating-point format. Subsequent operations will benefit from the additional precision of the internal double extended precision floating-point format.

When a denormal-operand exception occurs and the exception is not masked, the DE flag is set and a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software"). The top-of-stack pointer (TOP) and source operands remain unchanged.

For additional information about the denormal-operation exception, see Section 4.9.1.2, "Denormal Operand Exception (#D)."

8.5.3 Divide-By-Zero Exception (#Z)

The x87 FPU reports a floating-point divide-by-zero exception whenever an instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) for this exception is bit 2 of the x87 FPU status word, and the mask bit (ZM) is bit 2 of the x87 FPU control word. The FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, and FIDIVR instructions and the other instructions that perform division internally (FYL2X and FXTRACT) can report the divide-by-zero exception.

When a divide-by-zero exception occurs and the exception is masked, the x87 FPU sets the ZE flag and returns the values shown in Table 8-10. If the divide-by-zero exception is not masked, the ZE flag is set, a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software"); the top-of-stack pointer (TOP) and source operands remain unchanged.

For additional information about the divide-by-zero exception, see Section 4.9.1.2, "Divide-By-Zero Exception (#Z)."

8.5.4 Numeric Overflow Exception (#O)

The x87 FPU reports a floating-point numeric overflow exception (#O) whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that will fit into the floating-point format of the destination operand. (See Section 4.9.1.4, "Numeric Overflow Exception (#O)," for additional information about the numeric overflow exception.)

When using the x87 FPU, numeric overflow can occur on arithmetic operations where the result is stored in an x87 FPU data register. It can also occur on store floating-point operations (using the FST and FSTP instructions), where a within-range value in a data register is stored in memory in a single precision or double precision floating-point format.
format. The numeric overflow exception cannot occur when storing values in an integer or BCD integer format. Instead, the invalid-arithmetic-operand exception is signaled.

The flag (OE) for the numeric-overflow exception is bit 3 of the x87 FPU status word, and the mask bit (OM) is bit 3 of the x87 FPU control word.

When a numeric-overflow exception occurs and the exception is masked, the x87 FPU sets the OE flag and returns one of the values shown in Table 4-10. The value returned depends on the current rounding mode of the x87 FPU (see Section 8.1.5.3, "Rounding Control Field").

The action that the x87 FPU takes when numeric overflow occurs and the numeric-overflow exception is not masked, depends on whether the instruction is supposed to store the result in memory or on the register stack.

- **Destination is a memory location** — The OE flag is set and a software exception handler is invoked (see Section 8.7, "Handling x87 FPU Exceptions in Software"). The top-of-stack pointer (TOP) and source and destination operands remain unchanged. Because the data in the stack is in double extended precision format, the exception handler has the option either of re-executing the store instruction after proper adjustment of the operand or of rounding the significand on the stack to the destination's precision as the standard requires. The exception handler should ultimately store a value into the destination location in memory if the program is to continue.

- **Destination is the register stack** — The significand of the result is rounded according to current settings of the precision and rounding control bits in the x87 FPU control word and the exponent of the result is adjusted by dividing it by $2^{24576}$. (For instructions not affected by the precision field, the significand is rounded to double-extended precision.) The resulting value is stored in the destination operand. Condition code bit C1 in the x87 FPU status word (called in this situation the "round-up bit") is set if the significand was rounded upward and cleared if the result was rounded toward 0. After the result is stored, the OE flag is set and a software exception handler is invoked. The scaling bias value 24,576 is equal to $3 \times 2^{13}$. Biasing the exponent by 24,576 normally translates the number as nearly as possible to the middle of the double extended precision floating-point exponent range so that, if desired, it can be used in subsequent scaled operations with less risk of causing further exceptions.

When using the FSCALE instruction, massive overflow can occur, where the result is too large to be represented, even with a bias-adjusted exponent. Here, if overflow occurs again, after the result has been biased, a properly signed $\infty$ is stored in the destination operand.

### 8.5.5 Numeric Underflow Exception (#U)

The x87 FPU detects a potential floating-point numeric underflow condition whenever the result of an arithmetic instruction is non-zero and tiny; that is, the magnitude of the rounded result with unbounded exponent is non-zero and less than the smallest possible normalized, finite value that will fit into the floating-point format of the destination operand. See Section 4.9.1.5, "Numeric Underflow Exception (#U)," for additional information about the numeric underflow exception.

Like numeric overflow, numeric underflow can occur on arithmetic operations where the result is stored in an x87 FPU data register. It can also occur on store floating-point operations (with the FST and FSTP instructions), where a within-range value in a data register is stored in memory in the smaller single precision or double precision floating-point formats. A numeric underflow exception cannot occur when storing values in an integer or BCD integer format, because a value with magnitude less than 1 is always rounded to an integral value of 0 or 1, depending on the rounding mode in effect.

The flag (UE) for the numeric-underflow exception is bit 4 of the x87 FPU status word, and the mask bit (UM) is bit 4 of the x87 FPU control word.

When a numeric-underflow condition occurs and the exception is masked, the x87 FPU performs the operation described in Section 4.9.1.5, "Numeric Underflow Exception (#U)."

When the exception is not masked, the action of the x87 FPU depends on whether the instruction is supposed to store the result in a memory location or on the x87 FPU register stack.

- **Destination is a memory location** — (Can occur only with a store instruction.) The UE flag is set and a software exception handler is invoked; see Section 8.2, "x87 FPU Data Types." The top-of-stack pointer (TOP) and source and destination operands remain unchanged, and no result is stored in memory. Because the data in the stack is in double extended precision format, the exception handler has the option...
either of re-exchanges the store instruction after proper adjustment of the operand or of rounding the
significand on the stack to the destination’s precision as the standard requires. The exception handler should
ultimately store a value into the destination location in memory if the program is to continue.

• **Destination is the register stack** — The significand of the result is rounded according to current settings of
the precision and rounding control bits in the x87 FPU control word and the exponent of the result is adjusted
by multiplying it by $2^{24576}$. (For instructions not affected by the precision field, the significand is rounded to
double extended precision.) The resulting value is stored in the destination operand. Condition code bit C1 in
the x87 FPU status register (acting here as a “round-up bit”) is set if the significand was rounded upward and
cleared if the result was rounded toward 0. After the result is stored, the UE flag is set and a software exception
handler is invoked. The scaling bias value 24,576 is the same as is used for the overflow exception and has the
same effect, which is to translate the result as nearly as possible to the middle of the double extended precision
floating-point exponent range.

When using the FSCALE instruction, massive underflow can occur, where the magnitude of the result is too
small to be represented, even with a bias-adjusted exponent. Here, if underflow occurs again after the result
has been biased, a properly signed 0 is stored in the destination operand.

### 8.5.6 Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly
representable in the destination format. (See Section 4.9.1.6, “Inexact-Result (Precision) Exception (#P),” for
additional information about the numeric overflow exception.) Note that the transcendental instructions (FSIN,
FCOS, FSINCOS, FPTAN, FPATAN, F2XM1, FYL2X, and FYL2XP1) by nature produce inexact results.

The inexact-result exception flag (PE) is bit 5 of the x87 FPU status word, and the mask bit (PM) is bit 5 of the x87
FPU control word.

If the inexact-result exception is masked when an inexact-result condition occurs and a numeric overflow or under-
flow condition has not occurred, the x87 FPU handles the exception as describe in Section 4.9.1.6, “Inexact-Result
(Precision) Exception (#P),” with one additional action. The C1 (round-up) bit in the x87 FPU status word is set to
indicate whether the inexact result was rounded up (C1 is set) or “not rounded up” (C1 is cleared). In the “not
rounded up” case, the least-significant bits of the inexact result are truncated so that the result fits in the destina-
tion format.

If the inexact-result exception is not masked when an inexact result occurs and numeric overflow or underflow has
not occurred, the x87 FPU handles the exception as described in the previous paragraph and, in addition, invokes
a software exception handler.

If an inexact result occurs in conjunction with numeric overflow or underflow, the x87 FPU carries out one of the
following operations:

• If an inexact result occurs in conjunction with masked overflow or underflow, the OE or UE flag and the PE flag
are set and the result is stored as described for the overflow or underflow exceptions (see Section 8.5.4,
“Numeric Overflow Exception (#O),” or Section 8.5.5, “Numeric Underflow Exception (#U)”). If the inexact
result exception is unmasked, the x87 FPU also invokes a software exception handler.

• If an inexact result occurs in conjunction with unmasked overflow or underflow and the destination operand
is a register, the OE or UE flag and the PE flag are set, the result is stored as described for the overflow or
underflow exceptions (see Section 8.5.4, “Numeric Overflow Exception (#O),” or Section 8.5.5, “Numeric
Underflow Exception (#U)”) and a software exception handler is invoked.

If an unmasked numeric overflow or underflow exception occurs and the destination operand is a memory location
(which can happen only for a floating-point store), the inexact-result condition is not reported and the C1 flag is
cleared.

### 8.6 X87 FPU EXCEPTION SYNCHRONIZATION

Because the integer unit and x87 FPU are separate execution units, it is possible for the processor to execute
floating-point, integer, and system instructions concurrently. No special programming techniques are required to
gain the advantages of concurrent execution. (Floating-point instructions are placed in the instruction stream along
with the integer and system instructions.) However, concurrent execution can cause problems for floating-point exception handlers.

This problem is related to the way the x87 FPU signals the existence of unmasked floating-point exceptions. (Special exception synchronization is not required for masked floating-point exceptions, because the x87 FPU always returns a masked result to the destination operand.)

When a floating-point exception is unmasked and the exception condition occurs, the x87 FPU stops further execution of the floating-point instruction and signals the exception event. On the next occurrence of a floating-point instruction or a WAIT/FWAIT instruction in the instruction stream, the processor checks the ES flag in the x87 FPU status word for pending floating-point exceptions. If floating-point exceptions are pending, the x87 FPU makes an implicit call (traps) to the floating-point software exception handler. The exception handler can then execute recovery procedures for selected or all floating-point exceptions.

Synchronization problems occur in the time between the moment when the exception is signaled and when it is actually handled. Because of concurrent execution, integer or system instructions can be executed during this time. It is thus possible for the source or destination operands for a floating-point instruction that faulted to be overwritten in memory, making it impossible for the exception handler to analyze or recover from the exception.

To solve this problem, an exception synchronizing instruction (either a floating-point instruction or a WAIT/FWAIT instruction) can be placed immediately after any floating-point instruction that might present a situation where state information pertaining to a floating-point exception might be lost or corrupted. Floating-point instructions that store data in memory are prime candidates for synchronization. For example, the following three lines of code have the potential for exception synchronization problems:

```
FILD COUNT ;Floating-point instruction
INC COUNT  ;Integer instruction
FSQRT      ;Subsequent floating-point instruction
```

In this example, the INC instruction modifies the source operand of the floating-point instruction, FILD. If an exception is signaled during the execution of the FILD instruction, the INC instruction would be allowed to overwrite the value stored in the COUNT memory location before the floating-point exception handler is called. With the COUNT variable modified, the floating-point exception handler would not be able to recover from the error.

Rearranging the instructions, as follows, so that the FSQRT instruction follows the FILD instruction, synchronizes floating-point exception handling and eliminates the possibility of the COUNT variable being overwritten before the floating-point exception handler is invoked.

```
FILD COUNT ;Floating-point instruction
FSQRT      ;Subsequent floating-point instruction synchronizes
           ;any exceptions generated by the FILD instruction.
INC COUNT  ;Integer instruction
```

The FSQRT instruction does not require any synchronization, because the results of this instruction are stored in the x87 FPU data registers and will remain there, undisturbed, until the next floating-point or WAIT/FWAIT instruction is executed. To absolutely ensure that any exceptions emanating from the FSQRT instruction are handled (for example, prior to a procedure call), a WAIT instruction can be placed directly after the FSQRT instruction.

Note that some floating-point instructions (non-waiting instructions) do not check for pending unmasked exceptions (see Section 8.3.11, “x87 FPU Control Instructions”). They include the FNINIT, FNSTENV, FNSAVE, FNSTSW, FNSTCW, and FNCLEX instructions. When an FNINIT, FNSTENV, FNSAVE, or FNCLEX instruction is executed, all pending exceptions are essentially lost (either the x87 FPU status register is cleared or all exceptions are masked). The FNSTSW and FNSTCW instructions do not check for pending interrupts, but they do not modify the x87 FPU status and control registers. A subsequent “waiting” floating-point instruction can then handle any pending exceptions.

### 8.7 Handling X87 FPU Exceptions in Software

The x87 FPU in Pentium and later IA-32 processors provides two different modes of operation for invoking a software exception handler for floating-point exceptions: native mode and MS-DOS compatibility mode. The mode of operation is selected by CR0.NE[bit 5]. See Chapter 2, “System Architecture Overview,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information about the NE flag.
8.7.1 Native Mode

The native mode for handling floating-point exceptions is selected by setting CR0.NE[bit 5] to 1. In this mode, if the x87 FPU detects an exception condition while executing a floating-point instruction and the exception is unmasked (the mask bit for the exception is cleared), the x87 FPU sets the flag for the exception and the ES flag in the x87 FPU status word. It then invokes the software exception handler through the floating-point-error exception (#MF, exception vector 16), immediately before execution of any of the following instructions in the processor’s instruction stream:

- The next floating-point instruction, unless it is one of the non-waiting instructions (FNINIT, FNCLEX, FNSTSW, FNSTCW, FNSTENV, and FNSAVE).
- The next WAIT/FWAIT instruction.
- The next MMX instruction.

If the next floating-point instruction in the instruction stream is a non-waiting instruction, the x87 FPU executes the instruction without invoking the software exception handler.

8.7.2 MS-DOS* Compatibility Sub-mode

If CR0.NE[bit 5] is 0, the MS-DOS compatibility mode for handling floating-point exceptions is selected. In this mode, the software exception handler for floating-point exceptions is invoked externally using the processor’s FERR#, INTR, and IGNNE# pins. This method of reporting floating-point errors and invoking an exception handler is provided to support the floating-point exception handling mechanism used in PC systems that are running the MS-DOS or Windows* 95 operating system.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems, this approach also limits newer processors to operate with one logical processor active.

The MS-DOS compatibility mode is typically used as follows to invoke the floating-point exception handler:

1. If the x87 FPU detects an unmasked floating-point exception, it sets the flag for the exception and the ES flag in the x87 FPU status word.
2. If the IGNNE# pin is deasserted, the x87 FPU then asserts the FERR# pin either immediately, or else delayed (deferred) until just before the execution of the next waiting floating-point instruction or MMX instruction. Whether the FERR# pin is asserted immediately or delayed depends on the type of processor, the instruction, and the type of exception.
3. If a preceding floating-point instruction has set the exception flag for an unmasked x87 FPU exception, the processor freezes just before executing the next WAIT instruction, waiting floating-point instruction, or MMX instruction. Whether the FERR# pin was asserted at the preceding floating-point instruction or is just now being asserted, the freezing of the processor assures that the x87 FPU exception handler will be invoked before the new floating-point (or MMX) instruction gets executed.
4. The FERR# pin is connected through external hardware to IRQ13 of a cascaded, programmable interrupt controller (PIC). When the FERR# pin is asserted, the PIC is programmed to generate an interrupt 75H.
5. The PIC asserts the INTR pin on the processor to signal the interrupt 75H.
6. The BIOS for the PC system handles the interrupt 75H by branching to the interrupt 02H (NMI) interrupt handler.
7. The interrupt 02H handler determines if the interrupt is the result of an NMI interrupt or a floating-point exception.
8. If a floating-point exception is detected, the interrupt 02H handler branches to the floating-point exception handler.

If the IGNNE# pin is asserted, the processor ignores floating-point error conditions. This pin is provided to inhibit floating-point exceptions from being generated while the floating-point exception handler is servicing a previously signaled floating-point exception.

Appendix D, “Guidelines for Writing SIMD Floating-Point Exception Handlers,” describes the MS-DOS compatibility mode in much greater detail. This mode is somewhat more complicated in the Intel486 and Pentium processor implementations, as described in Appendix D.
8.7.3  Handling x87 FPU Exceptions in Software

Section 4.9.3, “Typical Actions of a Floating-Point Exception Handler,” shows actions that may be carried out by a floating-point exception handler. The state of the x87 FPU can be saved with the FSTENV/FNSTENV or FSAVE/FNSAVE instructions; see Section 8.1.10, "Saving the x87 FPU State with FSTENV/FNSTENV and FSAVE/FNSAVE."

If the faulting floating-point instruction is followed by one or more non-floating-point instructions, it may not be useful to re-execute the faulting instruction. See Section 8.6, “x87 FPU Exception Synchronization,” for more information on synchronizing floating-point exceptions.

In cases where the handler needs to restart program execution with the faulting instruction, the IRET instruction cannot be used directly. The reason for this is that because the exception is not generated until the next floating-point or WAIT/FWAIT instruction following the faulting floating-point instruction, the return instruction pointer on the stack may not point to the faulting instruction. To restart program execution at the faulting instruction, the exception handler must obtain a pointer to the instruction from the saved x87 FPU state information, load it into the return instruction pointer location on the stack, and then execute the IRET instruction.
The Intel MMX technology was introduced into the IA-32 architecture in the Pentium II processor family and Pentium processor with MMX technology. The extensions introduced in MMX technology support a single-instruction, multiple-data (SIMD) execution model that is designed to accelerate the performance of advanced media and communications applications.

This chapter describes MMX technology.

9.1 OVERVIEW OF MMX TECHNOLOGY

MMX technology defines a simple and flexible SIMD execution model to handle 64-bit packed integer data. This model adds the following features to the IA-32 architecture, while maintaining backwards compatibility with all IA-32 applications and operating-system code:

- Eight new 64-bit data registers, called MMX registers.
- Three new packed data types:
  - 64-bit packed byte integers (signed and unsigned).
  - 64-bit packed word integers (signed and unsigned).
  - 64-bit packed doubleword integers (signed and unsigned).
- Instructions that support the new data types and to handle MMX state management.
- Extensions to the CPUID instruction.

MMX technology is accessible from all the IA32-architecture execution modes (protected mode, real address mode, and virtual 8086 mode). It does not add any new modes to the architecture.

The following sections of this chapter describe MMX technology’s programming environment, including MMX register set, data types, and instruction set. Additional instructions that operate on MMX registers have been added to the IA-32 architecture by the SSE/SSE2 extensions.

For more information, see:

- Section 10.4.4, “Intel® SSE 64-Bit SIMD Integer Instructions,” describes MMX instructions added to the IA-32 architecture with the SSE extensions.
- Section 11.4.2, “Intel® SSE2 64-Bit and 128-Bit SIMD Integer Instructions,” describes MMX instructions added to the IA-32 architecture with SSE2 extensions.
- Chapter 13, “Intel® MMX™ Technology System Programming,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, describes the manner in which MMX technology is integrated into the IA-32 system programming model.

9.2 THE MMX TECHNOLOGY PROGRAMMING ENVIRONMENT

Figure 9-1 shows the execution environment for MMX technology. All MMX instructions operate on MMX registers, the general-purpose registers, and/or memory as follows:

- **MMX registers** — These eight registers (see Figure 9-1) are used to perform operations on 64-bit packed integer data. They are named MM0 through MM7.
• **General-purpose registers** — The eight general-purpose registers (see Figure 3-5) are used with existing IA-32 addressing modes to address operands in memory. (MMX registers cannot be used to address memory). General-purpose registers are also used to hold operands for some MMX technology operations. They are EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.

### 9.2.1 MMX Technology in 64-Bit Mode and Compatibility Mode

In compatibility mode and 64-bit mode, MMX instructions function like they do in protected mode. Memory operands are specified using the ModR/M, SIB encoding described in Section 3.7.5.

### 9.2.2 MMX Registers

The MMX register set consists of eight 64-bit registers (see Figure 9-2), that are used to perform calculations on the MMX packed integer data types. Values in MMX registers have the same format as a 64-bit quantity in memory. The MMX registers have two data access modes: 64-bit access mode and 32-bit access mode. The 64-bit access mode is used for:

- 64-bit memory accesses.
- 64-bit transfers between MMX registers.
- All pack, logical, and arithmetic instructions.
- Some unpack instructions.

The 32-bit access mode is used for:

- 32-bit memory accesses.
- 32-bit transfer between general-purpose registers and MMX registers.
- Some unpack instructions.
Although MMX registers are defined in the IA-32 architecture as separate registers, they are aliased to the registers in the FPU data register stack (R0 through R7).

See also Section 9.5, “Compatibility with x87 FPU Architecture.”

9.2.3 MMX Data Types

MMX technology introduced the following 64-bit data types to the IA-32 architecture (see Figure 9-3):

- 64-bit packed byte integers — eight packed bytes.
- 64-bit packed word integers — four packed words.
- 64-bit packed doubleword integers — two packed doublewords.

MMX instructions move 64-bit packed data types (packed bytes, packed words, or packed doublewords) and the quadword data type between MMX registers and memory or between MMX registers in 64-bit blocks. However, when performing arithmetic or logical operations on the packed data types, MMX instructions operate in parallel on the individual bytes, words, or doublewords contained in MMX registers; see Section 9.2.5, “Single Instruction, Multiple Data (SIMD) Execution Model.”

9.2.4 Memory Data Formats

When stored in memory: bytes, words, and doublewords in the packed data types are stored in consecutive addresses. The least significant byte, word, or doubleword is stored at the lowest address and the most significant byte, word, or doubleword is stored at the high address. The ordering of bytes, words, or doublewords in memory is always little endian. That is, the bytes with the low addresses are less significant than the bytes with high addresses.
9.2.5 Single Instruction, Multiple Data (SIMD) Execution Model

MMX technology uses the single instruction, multiple data (SIMD) technique for performing arithmetic and logical operations on bytes, words, or doublewords packed into MMX registers (see Figure 9-4). For example, the PADDSW instruction adds 4 signed word integers from one source operand to 4 signed word integers in a second source operand and stores 4 word integer results in a destination operand. This SIMD technique speeds up software performance by allowing the same operation to be carried out on multiple data elements in parallel. MMX technology supports parallel operations on byte, word, and doubleword data elements when contained in MMX registers.

The SIMD execution model supported in the MMX technology directly addresses the needs of modern media, communications, and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words, and doublewords). For example, most audio data is represented in 16-bit (word) quantities. The MMX instructions can operate on 4 words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities. In Figure 9-4, one MMX instruction operates on 8 bytes simultaneously.

![Figure 9-4. SIMD Execution Model](image)

9.3 SATURATION AND WRAPAROUND MODES

When performing integer arithmetic, an operation may result in an out-of-range condition, where the true result cannot be represented in the destination format. For example, when performing arithmetic on signed word integers, positive overflow can occur when the true signed result is larger than 16 bits.

The MMX technology provides three ways of handling out-of-range conditions:

- **Wraparound arithmetic** — With wraparound arithmetic, a true out-of-range result is truncated (that is, the carry or overflow bit is ignored and only the least significant bits of the result are returned to the destination). Wraparound arithmetic is suitable for applications that control the range of operands to prevent out-of-range results. If the range of operands is not controlled, however, wraparound arithmetic can lead to large errors. For example, adding two large signed numbers can cause positive overflow and produce a negative result.

- **Signed saturation arithmetic** — With signed saturation arithmetic, out-of-range results are limited to the representable range of signed integers for the integer size being operated on (see Table 9-1). For example, if positive overflow occurs when operating on signed word integers, the result is “saturated” to 7FFFH, which is the largest positive integer that can be represented in 16 bits; if negative overflow occurs, the result is saturated to 8000H.

- **Unsigned saturation arithmetic** — With unsigned saturation arithmetic, out-of-range results are limited to the representable range of unsigned integers for the integer size. So, positive overflow when operating on unsigned byte integers results in FFH being returned and negative overflow results in 00H being returned.
Saturation arithmetic provides an answer for many overflow situations. For example, in color calculations, saturation causes a color to remain pure black or pure white without allowing inversion. It also prevents wraparound artifacts from entering into computations when range checking of source operands is not used.

MMX instructions do not indicate overflow or underflow occurrence by generating exceptions or setting flags in the EFLAGS register.

### 9.4 MMX INSTRUCTIONS

The MMX instruction set consists of 47 instructions, grouped into the following categories:

- Data transfer
- Arithmetic
- Comparison
- Conversion
- Unpacking
- Logical
- Shift
- Empty MMX state instruction (EMMS)

Table 9-2 gives a summary of the instructions in the MMX instruction set. The following sections give a brief overview of the instructions within each group.

#### NOTES

The MMX instructions described in this chapter are those instructions that are available in an IA-32 processor when CPUID.01H:EDX.MMX[bit 23] = 1.

Section 10.4.4, "Intel® SSE 64-Bit SIMD Integer Instructions," and Section 11.4.2, "Intel® SSE2 64-Bit and 128-Bit SIMD Integer Instructions," list additional instructions included with the Intel SSE/SSE2 extensions that operate on MMX registers but are not considered part of the MMX instruction set.
9.4.1 Data Transfer Instructions

The MOVD (Move 32 Bits) instruction transfers 32 bits of packed data from memory to an MMX register and vice versa; or from a general-purpose register to an MMX register and vice versa.

The MOVQ (Move 64 Bits) instruction transfers 64 bits of packed data from memory to an MMX register and vice versa; or transfers data between MMX registers.

9.4.2 Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiplication, and multiply/add operations on packed data types.

The PADDB/PADDW/PADDD (add packed integers) instructions and the PSUBB/PSUBW/PSUBD (subtract packed integers) instructions add or subtract the corresponding signed or unsigned data elements of the source and desti-
nation operands in wraparound mode. These instructions operate on packed byte, word, and doubleword data types.

The PADDSB/PADDSW (add packed signed integers with signed saturation) instructions and the PSUBSB/PSUBSW (subtract packed signed integers with signed saturation) instructions add or subtract the corresponding signed data elements of the source and destination operands and saturate the result to the limits of the signed data-type range. These instructions operate on packed byte and word data types.

The PADDUSB/PADDUSW (add packed unsigned integers with unsigned saturation) instructions and the PSUBUSB/PSUBUSW (subtract packed unsigned integers with unsigned saturation) instructions add or subtract the corresponding unsigned data elements of the source and destination operands and saturate the result to the limits of the unsigned data-type range. These instructions operate on packed byte and word data types.

The PMULHW (multiply packed signed integers and store high result) and PMULLW (multiply packed signed integers and store low result) instructions perform a signed multiply of the corresponding words of the source and destination operands and write the high-order or low-order 16 bits of each of the results, respectively, to the destination operand.

The PMADDWD (multiply and add packed integers) instruction computes the products of the corresponding signed words of the source and destination operands. The four intermediate 32-bit doubleword products are summed in pairs (high-order pair and low-order pair) to produce two 32-bit doubleword results.

9.4.3 Comparison Instructions

The PCMPEQB/PCMPEQW/PCMPEQD (compare packed data for equal) instructions and the PCMPGTB/PCMPGTW/PCMPGTD (compare packed signed integers for greater than) instructions compare the corresponding signed data elements (bytes, words, or doublewords) in the source and destination operands for equal to or greater than, respectively.

These instructions generate a mask of ones or zeros which are written to the destination operand. Logical operations can use the mask to select packed elements. This can be used to implement a packed conditional move operation without a branch or a set of branch instructions. No flags in the EFLAGS register are affected.

9.4.4 Conversion Instructions

The PACKSSWB (pack words into bytes with signed saturation) and PACKSSDW (pack doublewords into words with signed saturation) instructions convert signed words into signed bytes and signed doublewords into signed words, respectively, using signed saturation.

PACKUSWB (pack words into bytes with unsigned saturation) converts signed words into unsigned bytes, using unsigned saturation.

9.4.5 Unpack Instructions

The PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ (unpack high-order data elements) instructions and the PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ (unpack low-order data elements) instructions unpack bytes, words, or doublewords from the high- or low-order data elements of the source and destination operands and interleave them in the destination operand. By placing all 0s in the source operand, these instructions can be used to convert byte integers to word integers, word integers to doubleword integers, or doubleword integers to quadword integers.

9.4.6 Logical Instructions

PAND (bitwise logical AND), PANDN (bitwise logical AND NOT), POR (bitwise logical OR), and PXOR (bitwise logical exclusive OR) perform bitwise logical operations on the quadword source and destination operands.
9.4.7 Shift Instructions

The logical shift left, logical shift right and arithmetic shift right instructions shift each element by a specified number of bit positions.

The PSLLW/PSLLD/PSLLQ (shift packed data left logical) instructions and the PSRLW/PSRLD/PSRLQ (shift packed data right logical) instructions perform a logical left or right shift of the data elements and fill the empty high or low order bit positions with zeros. These instructions operate on packed words, doublewords, and quadwords.

The PSRAW/PSRAD (shift packed data right arithmetic) instructions perform an arithmetic right shift, copying the sign bit for each data element into empty bit positions on the upper end of each data element. This instruction operates on packed words and doublewords.

9.4.8 EMMS Instruction

The EMMS instruction empties the MMX state by setting the tags in x87 FPU tag word to 11B, indicating empty registers. This instruction must be executed at the end of an MMX routine before calling other routines that can execute floating-point instructions. See Section 9.6.3, “Using the EMMS Instruction,” for more information on the use of this instruction.

9.5 COMPATIBILITY WITH X87 FPU ARCHITECTURE

The MMX state is aliased to the x87 FPU state. No new states or modes have been added to IA-32 architecture to support the MMX technology. The same floating-point instructions that save and restore the x87 FPU state also handle the MMX state (for example, during context switching).

MMX technology uses the same interface techniques between the x87 FPU and the operating system (primarily for task switching purposes). For more details, see Chapter 13, “Intel® MMX™ Technology System Programming,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

9.5.1 MMX Instructions and the x87 FPU Tag Word

After each MMX instruction, the entire x87 FPU tag word is set to valid (00B). The EMMS instruction (empty MMX state) sets the entire x87 FPU tag word to empty (11B).

Chapter 13, “Intel® MMX™ Technology System Programming,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, provides additional information about the effects of x87 FPU and MMX instructions on the x87 FPU tag word. For a description of the tag word, see Section 8.1.7, “x87 FPU Tag Word.”

9.6 WRITING APPLICATIONS WITH MMX CODE

The following sections give guidelines for writing application code that uses MMX technology.

9.6.1 Checking for MMX Technology Support

Before an application attempts to use the MMX technology, it should check that it is present on the processor. Check by following these steps:

1. Check that the processor supports the CPUID instruction by attempting to execute the CPUID instruction. If the processor does not support the CPUID instruction, this will generate an invalid-opcode exception (#UD).

2. Check that the processor supports the MMX technology
   (if CPUID.01H:EDX.MMX[bit 23] = 1).

3. Check that emulation of the x87 FPU is disabled (if CR0.EM[bit 2] = 0).

If the processor attempts to execute an unsupported MMX instruction or attempts to execute an MMX instruction with CR0.EM[bit 2] set, this generates an invalid-opcode exception (#UD).
Example 9-1 illustrates how to use the CPUID instruction to detect the MMX technology. This example does not represent the entire CPUID sequence, but shows the portion used for detection of MMX technology.

Example 9-1. Partial Routine for Detecting MMX Technology with the CPUID Instruction

```assembly
... ; identify existence of CPUID instruction
... ; identify Intel processor
mov EAX, 1 ; request for feature flags
CPUID ; 0FH, 0A2H CPUID instruction
test EDX, 00800000H ; Is IA MMX technology bit (Bit 23 of EDX) set?
jnz ; MMX_Technology_Found
```

9.6.2 Transitions Between x87 FPU and MMX Code

Applications can contain both x87 FPU floating-point and MMX instructions. However, because the MMX registers are aliased to the x87 FPU register stack, care must be taken when making transitions between x87 FPU instructions and MMX instructions to prevent incoherent or unexpected results.

When an MMX instruction (other than the EMMS instruction) is executed, the processor changes the x87 FPU state as follows:

- The TOS (top of stack) value of the x87 FPU status word is set to 0.
- The entire x87 FPU tag word is set to the valid state (00B in all tag fields).
- When an MMX instruction writes to an MMX register, it writes ones (11B) to the exponent part of the corresponding floating-point register (bits 64 through 79).

The net result of these actions is that any x87 FPU state prior to the execution of the MMX instruction is essentially lost.

When an x87 FPU instruction is executed, the processor assumes that the current state of the x87 FPU register stack and control registers is valid and executes the instruction without any preparatory modifications to the x87 FPU state.

If the application contains both x87 FPU floating-point and MMX instructions, the following guidelines are recommended:

- When transitioning between x87 FPU and MMX code, save the state of any x87 FPU data or control registers that need to be preserved for future use. The FSAVE and FXSAVE instructions save the entire x87 FPU state.
- When transitioning between MMX and x87 FPU code, do the following:
  - Save any data in the MMX registers that needs to be preserved for future use. FSAVE and FXSAVE also save the state of MMX registers.
  - Execute the EMMS instruction to clear the MMX state from the x87 data and control registers.

The following sections describe the use of the EMMS instruction and give additional guidelines for mixing x87 FPU and MMX code.

9.6.3 Using the EMMS Instruction

As described in Section 9.6.2, “Transitions Between x87 FPU and MMX Code,” when an MMX instruction executes, the x87 FPU tag word is marked valid (00B). In this state, the execution of subsequent x87 FPU instructions may produce unexpected x87 FPU floating-point exceptions and/or incorrect results because the x87 FPU register stack appears to contain valid data. The EMMS instruction is provided to prevent this problem by marking the x87 FPU tag word as empty.

The EMMS instruction should be used in each of the following cases:

- When an application using the x87 FPU instructions calls an MMX technology library/DLL (use the EMMS instruction at the end of the MMX code).
• When an application using MMX instructions calls a x87 FPU floating-point library/DLL (use the EMMS instruction before calling the x87 FPU code).
• When a switch is made between MMX code in a task or thread and other tasks or threads in cooperative operating systems, unless it is certain that more MMX instructions will be executed before any x87 FPU code.

EMMS is not required when mixing MMX technology instructions with Intel SSE/SSE2/SSE3 instructions; see Section 11.6.7, "Interaction of Intel® SSE and SSE2 Instructions with x87 FPU and MMX Instructions."

### 9.6.4 Mixing MMX and x87 FPU Instructions

An application can contain both x87 FPU floating-point and MMX instructions. However, frequent transitions between MMX and x87 FPU instructions are not recommended, because they can degrade performance in some processor implementations. When mixing MMX code with x87 FPU code, follow these guidelines:

• Keep the code in separate modules, procedures, or routines.
• Do not rely on register contents across transitions between x87 FPU and MMX code modules.
• When transitioning between MMX code and x87 FPU code, save the MMX register state (if it will be needed in the future) and execute an EMMS instruction to empty the MMX state.
• When transitioning between x87 FPU code and MMX code, save the x87 FPU state if it will be needed in the future.

### 9.6.5 Interfacing with MMX Code

MMX technology enables direct access to all the MMX registers. This means that all existing interface conventions that apply to the use of the processor’s general-purpose registers (EAX, EBX, etc.) also apply to the use of MMX registers.

An efficient interface to MMX routines might pass parameters and return values through the MMX registers or through a combination of memory locations (via the stack) and MMX registers. Do not use the EMMS instruction or mix MMX and x87 FPU code when using to the MMX registers to pass parameters.

If a high-level language that does not support the MMX data types directly is used, the MMX data types can be defined as a 64-bit structure containing packed data types.

When implementing MMX instructions in high-level languages, other approaches can be taken, such as:

• Passing parameters to an MMX routine by passing a pointer to a structure via the stack.
• Returning a value from a function by returning a pointer to a structure.

### 9.6.6 Using MMX Code in a Multitasking Operating System Environment

An application needs to identify the nature of the multitasking operating system on which it runs. Each task retains its own state which must be saved when a task switch occurs. The processor state (context) consists of the general-purpose registers and the floating-point and MMX registers.

Operating systems can be classified into two types:

• Cooperative multitasking operating system.
• Preemptive multitasking operating system.

Cooperative multitasking operating systems do not save the FPU or MMX state when performing a context switch. Therefore, the application needs to save the relevant state before relinquishing direct or indirect control to the operating system.

Preemptive multitasking operating systems are responsible for saving and restoring the FPU and MMX state when performing a context switch. Therefore, the application does not have to save or restore the FPU and MMX state.
9.6.7 Exception Handling in MMX Code

MMX instructions generate the same type of memory-access exceptions as other IA-32 instructions (page fault, segment not present, and limit violations). Existing exception handlers do not have to be modified to handle these types of exceptions for MMX code.

Unless there is a pending floating-point exception, MMX instructions do not generate numeric exceptions. Therefore, there is no need to modify existing exception handlers or add new ones to handle numeric exceptions.

If a floating-point exception is pending, the subsequent MMX instruction generates a numeric error exception (interrupt 16 and/or assertion of the FERR# pin). The MMX instruction resumes execution upon return from the exception handler.

9.6.8 Register Mapping

MMX registers and their tags are mapped to physical locations of the floating-point registers and their tags. Register aliasing and mapping is described in more detail in Chapter 13, “Intel® MMX™ Technology System Programming,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

9.6.9 Effect of Instruction Prefixes on MMX Instructions

Table 9-3 describes the effect of instruction prefixes on MMX instructions. Unpredictable behavior can range from being treated as a reserved operation on one generation of IA-32 processors to generating an invalid opcode exception on another generation of processors.

<table>
<thead>
<tr>
<th>Prefix Type</th>
<th>Effect on MMX Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Size Prefix (67H)</td>
<td>Affects instructions with a memory operand.</td>
</tr>
<tr>
<td></td>
<td>Reserved for instructions without a memory operand and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Operand Size (66H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Segment Override (2EH, 36H, 3EH, 26H, 64H, 65H)</td>
<td>Affects instructions with a memory operand.</td>
</tr>
<tr>
<td></td>
<td>Reserved for instructions without a memory operand and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Repeat Prefix (F3H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Repeat NE Prefix (F2H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Lock Prefix (F0H)</td>
<td>Reserved; generates invalid opcode exception (#UD).</td>
</tr>
<tr>
<td>Branch Hint Prefixes (2EH and 3EH)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
</tbody>
</table>

The Intel® Streaming SIMD Extensions (Intel® SSE) were introduced into the IA-32 architecture in the Pentium III processor family. These extensions enhance the performance of IA-32 processors for advanced 2-D and 3-D graphics, motion video, image processing, speech recognition, audio synthesis, telephony, and video conferencing.

This chapter describes SSE. Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2),” provides information to assist in writing application programs that use Intel SSE2. Chapter 12, “Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI,” provides this information for Intel SSE3.

10.1 OVERVIEW OF INTEL® SSE

Intel MMX technology introduced single-instruction multiple-data (SIMD) capability into the IA-32 architecture, with the 64-bit MMX registers, 64-bit packed integer data types, and instructions that allowed SIMD operations to be performed on packed integers. Intel SSE expanded the SIMD execution model by adding facilities for handling packed and scalar single precision floating-point values contained in 128-bit registers.

If CPUID.01H:EDX.SSE[bit 25] = 1, Intel SSE is available.

Intel SSE adds the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 processors, applications, and operating systems:

- Eight 128-bit data registers (called XMM registers) in non-64-bit modes; 16 XMM registers are available in 64-bit mode.
- The 32-bit MXCSR register, which provides control and status bits for operations performed on XMM registers.
- The 128-bit packed single precision floating-point data type (four IEEE single precision floating-point values packed into a double quadword).
- Instructions that perform SIMD operations on single precision floating-point values and that extend SIMD operations that can be performed on integers:
  - 128-bit Packed and scalar single precision floating-point instructions that operate on data located in MMX registers.
  - 64-bit SIMD integer instructions that support additional operations on packed integer operands located in MMX registers.
- Instructions that save and restore the state of the MXCSR register.
- Instructions that support explicit prefetching of data, control of the cacheability of data, and control the ordering of store operations.
- Extensions to the CPUID instruction.

These features extend the IA-32 architecture’s SIMD programming model in four important ways:

- The ability to perform SIMD operations on four packed single precision floating-point values enhances the performance of IA-32 processors for advanced media and communications applications that use computation-intensive algorithms to perform repetitive operations on large arrays of simple, native data elements.
- The ability to perform SIMD single precision floating-point operations in XMM registers and SIMD integer operations in MMX registers provides greater flexibility and throughput for executing applications that operate on large arrays of floating-point and integer data.
- Cache control instructions provide the ability to stream data in and out of XMM registers without polluting the caches and the ability to prefetch data to selected cache levels before it is actually used. Applications that require regular access to large amounts of data benefit from these prefetching and streaming store capabilities.
- The SFENCE (store fence) instruction provides greater control over the ordering of store operations when using weakly-ordered memory types.
Intel SSE is fully compatible with all software written for IA-32 processors. All existing software continues to run correctly, without modification, on processors that incorporate Intel SSE. Enhancements to CPUID permit detection of Intel SSE. Intel SSE is accessible from all IA-32 execution modes: protected mode, real address mode, and virtual-8086 mode.

The following sections of this chapter describe the programming environment for Intel SSE, including: XMM registers, the packed single precision floating-point data type, and Intel SSE instructions. For additional information, see:

- Section 11.6, “Writing Applications with Intel® SSE and SSE2.”
- Section 11.5, “Intel® SSE, SSE2, and SSE3 Exceptions,” describes the exceptions that can be generated with Intel SSE/SSE2/SSE3 instructions.
- The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, provides a detailed description of these instructions.

10.2 INTEL® SSE PROGRAMMING ENVIRONMENT

Figure 10-1 shows the execution environment for Intel SSE. All Intel SSE instructions operate on the XMM registers, MMX registers, and/or memory as follows:

- **XMM registers** — These eight registers (see Figure 10-2 and Section 10.2.2, “XMM Registers”) are used to operate on packed or scalar single precision floating-point data. Scalar operations are operations performed on individual (unpacked) single precision floating-point values stored in the low doubleword of an XMM register. XMM registers are referenced by the names XMM0 through XMM7.

- **MXCSR register** — This 32-bit register (see Figure 10-3 and Section 10.2.3, “MXCSR Control and Status Register”) provides status and control bits used in SIMD floating-point operations.

- **MMX registers** — These eight registers (see Figure 9-2) are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between the MMX and XMM registers. MMX registers are referenced by the names MM0 through MM7.

- **General-purpose registers** — The eight general-purpose registers (see Figure 3-5) are used along with the existing IA-32 addressing modes to address operands in memory. (MMX and XMM registers cannot be used to access memory.)
address memory). The general-purpose registers are also used to hold operands for some SSE instructions and
are referenced as EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.

• **EFLAGS register** — This 32-bit register (see Figure 3-8) is used to record result of some compare operations.

### 10.2.1 Intel® SSE in 64-Bit Mode and Compatibility Mode

In compatibility mode, Intel SSE functions like it does in protected mode. In 64-bit mode, eight additional XMM
registers are accessible. Registers XMM8-XMM15 are accessed by using REX prefixes. Memory operands are spec-
ified using the ModR/M, SIB encoding described in Section 3.7.5.

Some Intel SSE instructions may be used to operate on general-purpose registers. Use the REX.W prefix to access
64-bit general-purpose registers. Note that if a REX prefix is used when it has no meaning, the prefix is ignored.

### 10.2.2 XMM Registers

Eight 128-bit XMM data registers were introduced into the IA-32 architecture with Intel SSE (see Figure 10-2).
These registers can be accessed directly using the names XMM0 to XMM7; and they can be accessed independently
from the x87 FPU and MMX registers and the general-purpose registers (that is, they are not aliased to any other
of the processor’s registers).

![Figure 10-2. XMM Registers](image)

Intel SSE instructions use the XMM registers only to operate on packed single precision floating-point operands.
SSE2 extensions expand the functions of the XMM registers to operand on packed or scalar double precision
floating-point operands and packed integer operands; see Section 11.2, “Intel® SSE2 Programming Environment,”
and Section 12.1, “Programming Environment and Data types.”

XMM registers can only be used to perform calculations on data; they cannot be used to address memory.
Addressing memory is accomplished by using the general-purpose registers.

Data can be loaded into XMM registers or written from the registers to memory in 32-bit, 64-bit, and 128-bit incre-
ments. When storing the entire contents of an XMM register in memory (128-bit store), the data is stored in 16
consecutive bytes, with the low-order byte of the register being stored in the first byte in memory.

### 10.2.3 MXCSR Control and Status Register

The 32-bit MXCSR register (see Figure 10-3) contains control and status information for Intel SSE, SSE2, and SSE3
SIMD floating-point operations. This register contains:

• Flag and mask bits for SIMD floating-point exceptions.
• Rounding control field for SIMD floating-point operations.
• Flush-to-zero flag that provides a means of controlling underflow conditions on SIMD floating-point operations.
• Denormals-are-zeros flag that controls how SIMD floating-point instructions handle denormal source operands.

The contents of this register can be loaded from memory with the LDMXCSR and FXRSTOR instructions and stored in memory with STMXCSR and FXSAVE.

Bits 16 through 31 of the MXCSR register are reserved and are cleared on a power-up or reset of the processor; attempting to write a non-zero value to these bits, using either the FXRSTOR or LDMXCSR instructions, will result in a general-protection exception (#GP) being generated.

![MXCSR Control/Status Register](image)

**Figure 10-3. MXCSR Control/Status Register**

### 10.2.3.1 SIMD Floating-Point Mask and Flag Bits

Bits 0 through 5 of the MXCSR register indicate whether a SIMD floating-point exception has been detected. They are "sticky" flags. That is, after a flag is set, it remains set until explicitly cleared. To clear these flags, use the LDMXCSR or the FXRSTOR instruction to write zeroes to them.

Bits 7 through 12 provide individual mask bits for the SIMD floating-point exceptions. An exception type is masked if the corresponding mask bit is set, and it is unmasked if the bit is clear. These mask bits are set upon a power-up or reset. This causes all SIMD floating-point exceptions to be initially masked.

If LDMXCSR or FXRSTOR clears a mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be generated as a result of this change. The unmasked exception will be generated only upon the execution of the next SSE/SSE2/SSE3 instruction that detects the unmasked exception condition.

For more information about the use of the SIMD floating-point exception mask and flag bits, see Section 11.5, "Intel® SSE, SSE2, and SSE3 Exceptions,” and Section 12.8, "Intel® SSE3, SSSE3, And Intel® SSE4 Exceptions.”

### 10.2.3.2 SIMD Floating-Point Rounding Control Field

Bits 13 and 14 of the MXCSR register (the rounding control [RC] field) control how the results of SIMD floating-point instructions are rounded. See Section 4.8.4, “Rounding,” for a description of the function and encoding of the rounding control bits.

### 10.2.3.3 Flush-To-Zero

Bit 15 (FTZ) of the MXCSR register enables the flush-to-zero mode, which controls the masked response to a SIMD floating-point underflow condition. When the underflow exception is masked and the flush-to-zero mode is enabled, the processor performs the following operations when it detects a floating-point underflow condition.

---

10-4 Vol. 1
• Returns a zero result with the sign of the true result.
• Sets the precision and underflow exception flags.

If the underflow exception is not masked, the flush-to-zero bit is ignored.

The flush-to-zero mode is not compatible with IEEE Standard 754. The IEEE-mandated masked response to underflow is to deliver the denormalized result (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers"). The flush-to-zero mode is provided primarily for performance reasons. At the cost of a slight precision loss, faster execution can be achieved for applications where underflows are common and rounding the underflow result to zero can be tolerated.

The flush-to-zero bit is cleared upon a power-up or reset of the processor, disabling the flush-to-zero mode.

10.2.3.4 Denormals-Are-Zeros

Bit 6 (DAZ) of the MXCSR register enables the denormals-are-zeros mode, which controls the processor’s response to a SIMD floating-point denormal operand condition. When the denormals-are-zeros flag is set, the processor converts all denormal source operands to a zero with the sign of the original operand before performing any computations on them. The processor does not set the denormal-operand exception flag (DE), regardless of the setting of the denormal-operand exception mask bit (DM); and it does not generate a denormal-operand exception if the exception is unmasked.

The denormals-are-zeros mode is not compatible with IEEE Standard 754 (see Section 4.8.3.2, "Normalized and Denormalized Finite Numbers"). The denormals-are-zeros mode is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not appreciably affect the quality of the processed data.

The denormals-are-zeros flag is cleared upon a power-up or reset of the processor, disabling the denormals-are-zeros mode.

The denormals-are-zeros mode was introduced in the Pentium 4 and Intel Xeon processor with the SSE2 extensions; however, it is fully compatible with the SSE SIMD floating-point instructions (that is, the denormals-are-zeros flag affects the operation of the SSE SIMD floating-point instructions). In earlier IA-32 processors and in some models of the Pentium 4 processor, this flag (bit 6) is reserved. See Section 11.6.3, “Checking for the DAZ Flag in the MXCSR Register,” for instructions for detecting the availability of this feature.

Attempting to set bit 6 of the MXCSR register on processors that do not support the DAZ flag will cause a general-protection exception (#GP). See Section 11.6.6, “Guidelines for Writing to the MXCSR Register,” for instructions for preventing such general-protection exceptions by using the MXCSR_MASK value returned by the FXSAVE instruction.

10.2.4 Compatibility of Intel® SSE with Intel® SSE2 and SSE3, MMX, and the x87 FPU

The state (XMM registers and MXCSR register) introduced into the IA-32 execution environment with Intel SSE is shared with Intel SSE2 and SSE3. Intel SSE, SSE2, and SSE3 instructions are fully compatible; they can be executed together in the same instruction stream with no need to save state when switching between instruction sets.

XMM registers are independent of the x87 FPU and MMX registers, so Intel SSE, SSE2, and SSE3 operations performed on the XMM registers can be performed in parallel with operations on the x87 FPU and MMX registers; see Section 11.6.7, “Interaction of Intel® SSE and SSE2 Instructions with x87 FPU and MMX Instructions.”

The FXSAVE and FXRSTOR instructions save and restore the SSE/SSE2/SSE3 states along with the x87 FPU and MMX state.

10.3 INTEL® SSE DATA TYPES

Intel SSE introduced one data type, the 128-bit packed single precision floating-point data type, to the IA-32 architecture (see Figure 10-4). This data type consists of four IEEE 32-bit single precision floating-point values packed
into a double quadword. See Figure 4-3 for the layout of a single precision floating-point value; refer to Section 4.2.2, “Floating-Point Data Types,” for a detailed description of the single precision floating-point format.

![Figure 10-4. 128-Bit Packed Single Precision Floating-Point Data Type](chart)

This 128-bit packed single precision floating-point data type is operated on in the XMM registers or in memory. Conversion instructions are provided to convert two packed single precision floating-point values into two packed doubleword integers or a scalar single precision floating-point value into a doubleword integer (see Figure 11-8).

Intel SSE provides conversion instructions between XMM registers and MMX registers, and between XMM registers and general-purpose bit registers. See Figure 11-8.

The address of a 128-bit packed memory operand must be aligned on a 16-byte boundary, except in the following cases:

- The MOVUPS instruction supports unaligned accesses.
- Scalar instructions that use a 4-byte memory operand that is not subject to alignment requirements.

Figure 4-2 shows the byte order of 128-bit (double quadword) data types in memory.

## 10.4 INTEL® SSE INSTRUCTION SET

Intel SSE instructions are divided into four functional groups:

- Packed and scalar single precision floating-point instructions.
- 64-bit SIMD integer instructions.
- State management instructions.
- Cacheability control, prefetch, and memory ordering instructions.

The following sections give an overview of each of the instructions in these groups.

### 10.4.1 Intel® SSE Packed and Scalar Floating-Point Instructions

The packed and scalar single precision floating-point instructions are divided into the following subgroups:

- Data movement instructions.
- Arithmetic instructions.
- Logical instructions.
- Comparison instructions.
- Shuffle instructions.
- Conversion instructions.

The packed single precision floating-point instructions perform SIMD operations on packed single precision floating-point operands (see Figure 10-5). Each source operand contains four single precision floating-point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, X1 and Y1, X2 and Y2, and X3 and Y3) in each operand.
The scalar single precision floating-point instructions operate on the low (least significant) doublewords of the two source operands (X0 and Y0); see Figure 10-6. The three most significant doublewords (X1, X2, and X3) of the first source operand are passed through to the destination. The scalar operations are similar to the floating-point operations performed in the x87 FPU data registers with the precision control field in the x87 FPU control word set for single precision (24-bit significand), except that x87 stack operations use a 15-bit exponent range for the result, while SSE operations use an 8-bit exponent range.

10.4.1.1 Intel® SSE Data Movement Instructions

Intel SSE data movement instructions move single precision floating-point data between XMM registers and between an XMM register and memory.

The MOVAPS (move aligned packed single precision floating-point values) instruction transfers a double quadword operand containing four packed single precision floating-point values from memory to an XMM register and vice versa, or between XMM registers. The memory address must be aligned to a 16-byte boundary; otherwise, a general-protection exception (#GP) is generated.

The MOVUPS (move unaligned packed single precision, floating-point) instruction performs the same operations as the MOVAPS instruction, except that 16-byte alignment of a memory address is not required.

The MOVSS (move scalar single precision floating-point) instruction transfers a 32-bit single precision floating-point operand from memory to the low doubleword of an XMM register and vice versa, or between XMM registers.

The MOVLPS (move low packed single precision floating-point) instruction moves two packed single precision floating-point values from memory to the low quadword of an XMM register and vice versa. The high quadword of the register is left unchanged.
The MOVHPS (move high packed single precision floating-point) instruction moves two packed single precision floating-point values from memory to the high quadword of an XMM register and vice versa. The low quadword of the register is left unchanged.

The MOVLHPS (move packed single precision floating-point low to high) instruction moves two packed single precision floating-point values from the low quadword of the source XMM register into the high quadword of the destination XMM register. The low quadword of the destination register is left unchanged.

The MOVHLPS (move packed single precision floating-point high to low) instruction moves two packed single precision floating-point values from the high quadword of the source XMM register into the low quadword of the destination XMM register. The high quadword of the destination register is left unchanged.

The MOVMSKPS (move packed single precision floating-point mask) instruction transfers the most significant bit of each of the four packed single precision floating-point numbers in an XMM register to a general-purpose register. This 4-bit value can then be used as a condition to perform branching.

10.4.1.2 Intel® SSE Arithmetic Instructions

Intel SSE arithmetic instructions perform addition, subtraction, multiply, divide, reciprocal, square root, reciprocal of square root, and maximum/minimum operations on packed and scalar single precision floating-point values.

The ADDPS (add packed single precision floating-point values) and SUBPS (subtract packed single precision floating-point values) instructions add and subtract, respectively, two packed single precision floating-point operands.

The ADDSS (add scalar single precision floating-point values) and SUBSS (subtract scalar single precision floating-point values) instructions add and subtract, respectively, the low single precision floating-point values of two operands and store the result in the low doubleword of the destination operand.

The MULPS (multiply packed single precision floating-point values) instruction multiplies two packed single precision floating-point operands.

The MULSS (multiply scalar single precision floating-point values) instruction multiplies the low single precision floating-point values of two operands and stores the result in the low doubleword of the destination operand.

The DIVPS (divide packed, single precision floating-point values) instruction divides two packed single precision floating-point operands.

The DIVSS (divide scalar single precision floating-point values) instruction divides the low single precision floating-point values of two operands and stores the result in the low doubleword of the destination operand.

The RCPPS (compute reciprocals of packed single precision floating-point values) instruction computes the approximate reciprocals of values in a packed single precision floating-point operand.

The RCPSS (compute reciprocal of scalar single precision floating-point values) instruction computes the approximate reciprocal of the low single precision floating-point value in the source operand and stores the result in the low doubleword of the destination operand.

The SQRTPS (compute square roots of packed single precision floating-point values) instruction computes the square roots of the values in a packed single precision floating-point operand.

The SQRTSS (compute square root of scalar single precision floating-point values) instruction computes the square root of the low single precision floating-point value in the source operand and stores the result in the low doubleword of the destination operand.

The RSQRTPS (compute reciprocals of square roots of packed single precision floating-point values) instruction computes the approximate reciprocals of the square roots of the values in a packed single precision floating-point operand.

The RSQRTSS (reciprocal of square root of scalar single precision floating-point value) instruction computes the approximate reciprocal of the square root of the low single precision floating-point value in the source operand and stores the result in the low doubleword of the destination operand.

The MAXPS (return maximum of packed single precision floating-point values) instruction compares the corresponding values from two packed single precision floating-point operands and returns the numerically greater value from each comparison to the destination operand.
The MAXSS (return maximum of scalar single precision floating-point values) instruction compares the low values from two packed single precision floating-point operands and returns the numerically greater value from the comparison to the low doubleword of the destination operand.

The MINPS (return minimum of packed single precision floating-point values) instruction compares the corresponding values from two packed single precision floating-point operands and returns the numerically lesser value from each comparison to the destination operand.

The MINSS (return minimum of scalar single precision floating-point values) instruction compares the low values from two packed single precision floating-point operands and returns the numerically lesser value from the comparison to the low doubleword of the destination operand.

10.4.2 Intel® SSE Logical Instructions

Intel SSE logical instructions perform AND, AND NOT, OR, and XOR operations on packed single precision floating-point values.

The ANDPS (bitwise logical AND of packed single precision floating-point values) instruction returns the logical AND of two packed single precision floating-point operands.

The ANDNPS (bitwise logical AND NOT of packed single precision, floating-point values) instruction returns the logical AND NOT of two packed single precision floating-point operands.

The ORPS (bitwise logical OR of packed single precision, floating-point values) instruction returns the logical OR of two packed single precision floating-point operands.

The XORPS (bitwise logical XOR of packed single precision, floating-point values) instruction returns the logical XOR of two packed single precision floating-point operands.

10.4.2.1 Intel® SSE Comparison Instructions

The compare instructions compare packed and scalar single precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

The CMPPS (compare packed single precision floating-point values) instruction compares the corresponding values from two packed single precision floating-point operands, using an immediate operand as a predicate, and returns a 32-bit mask result of all 1s or all 0s for each comparison to the destination operand. The value of the immediate operand allows the selection of any of 8 compare conditions: equal, less than, less than equal, unordered, not equal, not less than, not less than or equal, or ordered.

The CMPSS (compare scalar single precision, floating-point values) instruction compares the low values from two packed single precision floating-point operands, using an immediate operand as a predicate, and returns a 32-bit mask result of all 1s or all 0s for the comparison to the low doubleword of the destination operand. The immediate operand selects the compare conditions as with the CMPPS instruction.

The COMISS (compare scalar single precision floating-point values and set EFLAGS) and UCOMISS (unordered compare scalar single precision floating-point values and set EFLAGS) instructions compare the low values of two packed single precision floating-point operands and set the ZF, PF, and CF flags in the EFLAGS register to show the result (greater than, less than, equal, or unordered). These two instructions differ as follows: the COMISS instruction signals a floating-point invalid-operation (#I) exception when a source operand is either a QNaN or an SNaN; the UCOMISS instruction only signals an invalid-operation exception when a source operand is an SNaN.

10.4.2.2 Intel® SSE Shuffle and Unpack Instructions

Intel SSE shuffle and unpack instructions shuffle or interleave the contents of two packed single precision floating-point values and store the results in the destination operand.

The SHUFPS (shuffle packed single precision floating-point values) instruction places any two of the four packed single precision floating-point values from the destination operand into the two low-order doublewords of the destination operand, and places any two of the four packed single precision floating-point values from the source operand in the two high-order doublewords of the destination operand (see Figure 10-7). By using the same register for the source and destination operands, the SHUFPS instruction can shuffle four single precision floating-point values into any order.
The UNPCKHPS (unpack and interleave high packed single precision floating-point values) instruction performs an interleaved unpack of the high-order single precision floating-point values from the source and destination operands and stores the result in the destination operand (see Figure 10-8).

The UNPCKLPS (unpack and interleave low packed single precision floating-point values) instruction performs an interleaved unpack of the low-order single precision floating-point values from the source and destination operands and stores the result in the destination operand (see Figure 10-9).

---

**Figure 10-7. SHUFPS Instruction, Packed Shuffle Operation**

**Figure 10-8. UNPCKHPS Instruction, High Unpack and Interleave Operation**

**Figure 10-9. UNPCKLPS Instruction, Low Unpack and Interleave Operation**
10.4.3 Intel® SSE Conversion Instructions

Intel SSE conversion instructions (see Figure 11-8) support packed and scalar conversions between single precision floating-point and doubleword integer formats.

The CVTPI2PS (convert packed doubleword integers to packed single precision floating-point values) instruction converts two packed signed doubleword integers into two packed single precision floating-point values. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTSI2SS (convert doubleword integer to scalar single precision floating-point value) instruction converts a signed doubleword integer into a single precision floating-point value. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTPS2PI (convert packed single precision floating-point values to packed doubleword integers) instruction converts two packed single precision floating-point values into two packed signed doubleword integers. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register. The CVTTPS2PI (convert with truncation packed single precision floating-point values to packed doubleword integers) instruction is similar to the CVTPS2PI instruction, except that truncation is used to round a source value to an integer value; see Section 4.8.4.2, “Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions.”

The CVTSS2SI (convert scalar single precision floating-point value to doubleword integer) instruction converts a single precision floating-point value into a signed doubleword integer. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register. The CVTTSS2SI (convert with truncation scalar single precision floating-point value to doubleword integer) instruction is similar to the CVTSS2SI instruction, except that truncation is used to round the source value to an integer value; see Section 4.8.4.2, “Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions.”

10.4.4 Intel® SSE 64-Bit SIMD Integer Instructions

Intel SSE adds the following 64-bit packed integer instructions to the IA-32 architecture. These instructions operate on data in MMX registers and 64-bit memory locations.

NOTE

When Intel SSE2 is present in an IA-32 processor, these instructions are extended to operate on 128-bit operands in XMM registers and 128-bit memory locations.

The PAVGB (compute average of packed unsigned byte integers) and PAVGW (compute average of packed unsigned word integers) instructions compute a SIMD average of two packed unsigned byte or word integer operands, respectively. For each corresponding pair of data elements in the packed source operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The PEXTRW (extract word) instruction copies a selected word from an MMX register into a general-purpose register.

The PINSRW (insert word) instruction copies a word from a general-purpose register or from memory into a selected word location in an MMX register.

The PMAXUB (maximum of packed unsigned byte integers) instruction compares the corresponding unsigned byte integers in two packed operands and returns the greater of each comparison to the destination operand.

The PMINUB (minimum of packed unsigned byte integers) instruction compares the corresponding unsigned byte integers in two packed operands and returns the lesser of each comparison to the destination operand.

The PMAXSW (maximum of packed signed word integers) instruction compares the corresponding signed word integers in two packed operands and returns the greater of each comparison to the destination operand.

The PMINSW (minimum of packed signed word integers) instruction compares the corresponding signed word integers in two packed operands and returns the lesser of each comparison to the destination operand.

The PMOVMSKB (move byte mask) instruction creates an 8-bit mask from the packed byte integers in an MMX register and stores the result in the low byte of a general-purpose register. The mask contains the most significant bit of each byte in the MMX register. (When operating on 128-bit operands, a 16-bit mask is created.)
The PMULHUW (multiply packed unsigned word integers and store high result) instruction performs a SIMD unsigned multiply of the words in the two source operands and returns the high word of each result to an MMX register.

The PSADBW (compute sum of absolute differences) instruction computes the SIMD absolute differences of the corresponding unsigned byte integers in two source operands, sums the differences, and stores the sum in the low word of the destination operand.

The PSHUFW (shuffle packed word integers) instruction shuffles the words in the source operand according to the order specified by an 8-bit immediate operand and returns the result to the destination operand.

10.4.5 MXCSR State Management Instructions

The MXCSR state management instructions (LDMXCSR and STMXCSR) load and save the state of the MXCSR register, respectively. The LDMXCSR instruction loads the MXCSR register from memory, while the STMXCSR instruction stores the contents of the register to memory.

10.4.6 Cacheability Control, Prefetch, and Memory Ordering Instructions

Intel SSE introduced several new instructions to give programs more control over the caching of data. They also introduces the PREFETCHh instructions, which provide the ability to prefetch data to a specified cache level, and the SFENCE instruction, which enforces program ordering on stores. These instructions are described in the following sections.

10.4.6.1 Cacheability Control Instructions

The following three instructions enable data from the MMX and XMM registers to be stored to memory using a non-temporal hint. The non-temporal hint directs the processor to store the data to memory without writing the data into the cache hierarchy. See Section 10.4.6.2, “Caching of Temporal vs. Non-Temporal Data,” for information about non-temporal stores and hints.

The MOVNTQ (store quadword using non-temporal hint) instruction stores packed integer data from an MMX register to memory, using a non-temporal hint.

The MOVNTPS (store packed single precision floating-point values using non-temporal hint) instruction stores packed floating-point data from an XMM register to memory, using a non-temporal hint.

The MASKMOVQ (store selected bytes of quadword) instruction stores selected byte integers from an MMX register to memory, using a byte mask to selectively write the individual bytes. This instruction also uses a non-temporal hint.

10.4.6.2 Caching of Temporal vs. Non-Temporal Data

Data referenced by a program can be temporal (data will be used again) or non-temporal (data will be referenced once and not reused in the immediate future). For example, program code is generally temporal, whereas, multimedia data, such as the display list in a 3-D graphics application, is often non-temporal. To make efficient use of the processor’s caches, it is generally desirable to cache temporal data and not cache non-temporal data. Overloading the processor’s caches with non-temporal data is sometimes referred to as “polluting the caches.” The Intel SSE and SSE2 cacheability control instructions enable a program to write non-temporal data to memory in a manner that minimizes pollution of caches.

These Intel SSE and SSE2 non-temporal store instructions minimize cache pollutions by treating the memory being accessed as the write combining (WC) type. If a program specifies a non-temporal store with one of these instructions and the memory type of the destination region is write back (WB), write through (WT), or write combining (WC), the processor will do the following:

- If the memory location being written to is present in the cache hierarchy, the data in the caches is evicted.¹

¹ Some older CPU implementations (e.g., Pentium M) allowed addresses being written with a non-temporal store instruction to be updated in-place if the memory type was not WC and line was already in the cache.
• The non-temporal data is written to memory with WC semantics.

See also: Chapter 12, "Memory Cache Control," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

Using the WC semantics, the store transaction will be weakly ordered, meaning that the data may not be written to memory in program order, and the store will not write allocate (that is, the processor will not fetch the corresponding cache line into the cache hierarchy, prior to performing the store). Also, different processor implementations may choose to collapse and combine these stores.

The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in uncacheable memory. Uncacheable as referred to here means that the region being written to has been mapped with either an uncacheable (UC) or write protected (WP) memory type.

In general, WC semantics require software to ensure coherence, with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and fencing must be performed for producer-consumer usage models. Fencing ensures that all system agents have global visibility of the stored data; for instance, failure to fence may result in a written cache line staying within a processor and not being visible to other agents.

The memory type visible on the bus in the presence of memory type aliasing is implementation specific. As one possible example, the memory type written to the bus may reflect the memory type for the first store to this line, as seen in program order; other alternatives are possible. This behavior should be considered reserved, and dependence on the behavior of any particular implementation risks future incompatibility.

**NOTE**
Some older CPU implementations (e.g., Pentium M) may implement non-temporal stores by updating in place data that already reside in the cache hierarchy. For such processors, the destination region should also be mapped as WC. If mapped as WB or WT, there is the potential for speculative processor reads to bring the data into the caches; in this case, non-temporal stores would then update in place, and data would not be flushed from the processor by a subsequent fencing operation.

### 10.4.6.3 PREFETCH<sub>h</sub> Instructions

The PREFETCH<sub>h</sub> instructions permit programs to load data into the processor at a suggested cache level, so that the data is closer to the processor’s load and store unit when it is needed. These instructions fetch 32 aligned bytes (or more, depending on the implementation) containing the addressed byte to a location in the cache hierarchy specified by the temporal locality hint (see Table 10-1). In this table, the first-level cache is closest to the processor and second-level cache is farther away from the processor than the first-level cache. The hints specify a prefetch of either temporal or non-temporal data (see Section 10.4.6.2, "Caching of Temporal vs. Non-Temporal Data").

Subsequent accesses to temporal data are treated like normal accesses, while those to non-temporal data will continue to minimize cache pollution. If the data is already present at a level of the cache hierarchy that is closer to the processor, the PREFETCH<sub>h</sub> instruction will not result in any data movement. The PREFETCH<sub>h</sub> instructions do not affect functional behavior of the program.

See Section 11.6.13, “Cacheability Hint Instructions,” for additional information about the PREFETCH<sub>h</sub> instructions.

<table>
<thead>
<tr>
<th><strong>Table 10-1. PREFETCH&lt;sub&gt;h&lt;/sub&gt; Instructions Caching Hints</strong></th>
<th>Actions</th>
</tr>
</thead>
</table>
| PREFETCHT0 Temporal data—fetch data into all levels of cache hierarchy: | • Pentium III processor—1st-level cache or 2nd-level cache  
• Pentium 4 and Intel Xeon processor—2nd-level cache |
| PREFETCHT1 Temporal data—fetch data into level 2 cache and higher | • Pentium III processor—2nd-level cache  
• Pentium 4 and Intel Xeon processor—2nd-level cache |
10.4.6.4 SFENCE Instruction

The SFENCE (Store Fence) instruction controls write ordering by creating a fence for memory store operations. This instruction guarantees that the result of every store instruction that precedes the store fence in program order is globally visible before any store instruction that follows the fence. The SFENCE instruction provides an efficient way of ensuring ordering between procedures that produce weakly-ordered data and procedures that consume that data.

10.5 FXSAVE AND FXRSTOR INSTRUCTIONS

The FXSAVE and FXRSTOR instructions were introduced into the IA-32 architecture in the Pentium II processor family (prior to the introduction of the SSE extensions). The original versions of these instructions performed a fast save and restore, respectively, of the x87 execution environment (x87 state). (By saving the state of the x87 FPU data registers, the FXSAVE and FXRSTOR instructions implicitly save and restore the state of the MMX registers.) The SSE extensions expanded the scope of these instructions to save and restore the states of the XMM registers and the MXCSR register (SSE state), along with x87 state.

The FXSAVE and FXRSTOR instructions can be used in place of the FSAVE/FNSAVE and FRSTOR instructions; however, the operation of the FXSAVE and FXRSTOR instructions are not identical to the operation of FSAVE/FNSAVE and FRSTOR.

NOTE
The FXSAVE and FXRSTOR instructions are not considered part of the SSE instruction group. They have a separate CPUID feature bit to indicate whether they are present (if CPUID.01H:EDX.FXSR[bit 24] = 1).

The CPUID feature bit for SSE extensions does not indicate the presence of FXSAVE and FXRSTOR.

The FXSAVE and FXRSTOR instructions organize x87 state and SSE state in a region of memory called the FXSAVE area. Section 10.5.1 provides details of the FXSAVE area and its format. Section 10.5.2 describes operation of FXSAVE, and Section 10.5.3 describes the operation of FXRSTOR.

10.5.1 FXSAVE Area

The FXSAVE and FXRSTOR instructions organize x87 state and SSE state in a region of memory called the FXSAVE area. Each of the instructions takes a memory operand that specifies the 16-byte aligned base address of the FXSAVE area on which it operates.

<table>
<thead>
<tr>
<th>PREFETCHInstruction Mnemonic</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFETCHT2</td>
<td>Temporal data—fetch data into level 2 cache and higher</td>
</tr>
<tr>
<td></td>
<td>• Pentium III processor—2nd-level cache</td>
</tr>
<tr>
<td></td>
<td>• Pentium 4 and Intel Xeon processor—2nd-level cache</td>
</tr>
<tr>
<td>PREFETCHNTA</td>
<td>Non-temporal data—fetch data into location close to the processor, minimizing cache pollution</td>
</tr>
<tr>
<td></td>
<td>• Pentium III processor—1st-level cache</td>
</tr>
<tr>
<td></td>
<td>• Pentium 4 and Intel Xeon processor—2nd-level cache</td>
</tr>
</tbody>
</table>
Every FXSAVE area comprises the 512 bytes starting at the area’s base address. Table 10-2 illustrates the format of the first 416 bytes of the legacy region of an FXSAVE area.

### Table 10-2. Format of an FXSAVE Area

<table>
<thead>
<tr>
<th>Bit</th>
<th>MXCSR_MASK</th>
<th>Reserved</th>
<th>MXCSR</th>
<th>Reserved</th>
<th>FPU IP bits 63:32</th>
<th>FPU IP bits 31:0</th>
<th>DS or FPU DP bits 63:32</th>
<th>FPU DP bits 31:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reserved</td>
<td>CS or FPU IP bits 63:32</td>
<td>Reserved</td>
<td>MXCSR</td>
<td>FOP</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>FPU IP bits 63:32</td>
<td></td>
<td>FTW</td>
<td>FSW</td>
<td></td>
<td>DS or FPU DP bits 63:32</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Reserved</td>
<td></td>
<td>FOP</td>
<td>FCW</td>
<td>0</td>
<td>FPU DP bits 31:0</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>Reserved</td>
<td>32</td>
<td>Reserved</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>MXCSR MASK</td>
<td>48</td>
<td>Reserved</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>MXCSR DF63:32</td>
<td>64</td>
<td>Reserved</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>Reserved</td>
<td>80</td>
<td>Reserved</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>MXCSR DF31:0</td>
<td>96</td>
<td>Reserved</td>
<td>96</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>Reserved</td>
<td>112</td>
<td>Reserved</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>Reserved</td>
<td>128</td>
<td>Reserved</td>
<td>128</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
<td>XMM0</td>
<td></td>
<td>MXCSR</td>
<td>144</td>
<td>Reserved</td>
<td>Reserved</td>
<td>144</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>XMM1</td>
<td></td>
<td>MXCSR</td>
<td>160</td>
<td>XMM0</td>
<td>Reserved</td>
<td>160</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>XMM2</td>
<td></td>
<td>MXCSR</td>
<td>176</td>
<td>XMM1</td>
<td>Reserved</td>
<td>176</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>XMM3</td>
<td></td>
<td>MXCSR</td>
<td>192</td>
<td>XMM2</td>
<td>Reserved</td>
<td>192</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>XMM4</td>
<td></td>
<td>MXCSR</td>
<td>208</td>
<td>XMM3</td>
<td>Reserved</td>
<td>208</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>XMM5</td>
<td></td>
<td>MXCSR</td>
<td>224</td>
<td>XMM4</td>
<td>Reserved</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM6</td>
<td></td>
<td>MXCSR</td>
<td>240</td>
<td>XMM5</td>
<td>Reserved</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM7</td>
<td></td>
<td>MXCSR</td>
<td>256</td>
<td>XMM6</td>
<td>Reserved</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM8</td>
<td></td>
<td>MXCSR</td>
<td>272</td>
<td>XMM7</td>
<td>Reserved</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM9</td>
<td></td>
<td>MXCSR</td>
<td>288</td>
<td>XMM8</td>
<td>Reserved</td>
<td>288</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM10</td>
<td></td>
<td>MXCSR</td>
<td>304</td>
<td>XMM9</td>
<td>Reserved</td>
<td>304</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM11</td>
<td></td>
<td>MXCSR</td>
<td>320</td>
<td>XMM10</td>
<td>Reserved</td>
<td>320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM12</td>
<td></td>
<td>MXCSR</td>
<td>336</td>
<td>XMM11</td>
<td>Reserved</td>
<td>336</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM13</td>
<td></td>
<td>MXCSR</td>
<td>352</td>
<td>XMM12</td>
<td>Reserved</td>
<td>352</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM14</td>
<td></td>
<td>MXCSR</td>
<td>368</td>
<td>XMM13</td>
<td>Reserved</td>
<td>368</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM15</td>
<td></td>
<td>MXCSR</td>
<td>384</td>
<td>XMM14</td>
<td>Reserved</td>
<td>384</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td>Reserved</td>
<td>400</td>
<td>XMM15</td>
<td>400</td>
</tr>
</tbody>
</table>

The x87 state component comprises bytes 23:0 and bytes 159:32. The SSE state component comprises bytes 31:24 and bytes 415:160. FXSAVE and FXRSTOR do not use bytes 511:416; bytes 463:416 are reserved. Section 10.5.2 and Section 10.5.3 provide details of how FXSAVE and FXRSTOR use an FXSAVE area.

### 10.5.1.1 x87 State

Table 10-2 illustrates how FXSAVE and FXRSTOR organize x87 state and SSE state; the x87 state is listed below, along with details of its interactions with FXSAVE and FXRSTOR:

- Bytes 1:0, 3:2, and 7:6 are used for x87 FPU Control Word (FCW), x87 FPU Status Word (FSW), and x87 FPU Opcode (FOP), respectively.
• Byte 4 is used for an abridged version of the x87 FPU Tag Word (FTW). The following items describe its usage:
  — For each \( j \), \( 0 \leq j \leq 7 \), FXSAVE saves a 0 into bit \( j \) of byte 4 if x87 FPU data register \( ST_0 \) has an empty tag; otherwise, FXSAVE saves a 1 into bit \( j \) of byte 4.
  — For each \( j \), \( 0 \leq j \leq 7 \), FXRSTOR establishes the tag value for x87 FPU data register \( ST_j \) as follows. If bit \( j \) of byte 4 is 0, the tag for \( ST_j \) in the tag register for that data register is marked empty (11B); otherwise, the x87 FPU sets the tag for \( ST_j \) based on the value being loaded into that register (see below).
• Bytes 15:8 are used as follows:
  — If the instruction has no REX prefix, or if REX.W = 0:
    • Bytes 11:8 are used for bits 31:0 of the x87 FPU Instruction Pointer Offset (FIP).
    • If CPUID.(EAX=07H, ECX=0H):EBX[bit 13] = 0, bytes 13:12 are used for x87 FPU Instruction Pointer Selector (FPU CS). Otherwise, the processor deprecates the FPU CS value: FXSAVE saves it as 0000H.
    • Bytes 15:14 are not used.
  — If the instruction has a REX prefix with REX.W = 1, bytes 15:8 are used for the full 64 bits of FIP.
• Bytes 23:16 are used as follows:
  — If the instruction has no REX prefix, or if REX.W = 0:
    • Bytes 19:16 are used for bits 31:0 of the x87 FPU Data Pointer Offset (FDP).
    • If CPUID.(EAX=07H, ECX=0H):EBX[bit 13] = 0, bytes 21:20 are used for x87 FPU Data Pointer Selector (FPU DS). Otherwise, the processor deprecates the FPU DS value: FXSAVE saves it as 0000H.
    • Bytes 23:22 are not used.
  — If the instruction has a REX prefix with REX.W = 1, bytes 23:16 are used for the full 64 bits of FDP.
• Bytes 31:24 are used for SSE state (see Section 10.5.1.2).
• Bytes 287:160 are used for the registers XMM0–XMM7.
• Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of FXSAVE outside 64-bit mode do not write to these bytes; executions of FXRSTOR outside 64-bit mode do not read these bytes and do not update XMM8–XMM15.
If CR4.OSFXSR = 0, FXSAVE and FXRSTOR may or may not operate on SSE state; this behavior is implementation dependent. Moreover, SSE instructions cannot be used unless CR4.OSFXSR = 1.

10.5.2 Operation of FXSAVE
The FXSAVE instruction takes a single memory operand, which is an FXSAVE area. The instruction stores x87 state and SSE state to the FXSAVE area. See Section 10.5.1.1 and Section 10.5.1.2 for details regarding mode-specific operation and operation determined by instruction prefixes.
10.5.3  Operation of FXRSTOR

The FXRSTOR instruction takes a single memory operand, which is an FXSAVE area. If the value at bytes 27:24 of the FXSAVE area is not a legal value for the MXCSR register (e.g., the value sets reserved bits), execution of FXRSTOR results in a general-protection fault (#GP). Otherwise, the instruction loads x87 state and SSE state from the FXSAVE area. See Section 10.5.1.1 and Section 10.5.1.2 for details regarding mode-specific operation and operation determined by instruction prefixes.

10.6  HANDLING INTEL® SSE INSTRUCTION EXCEPTIONS

See Section 11.5, "Intel® SSE, SSE2, and SSE3 Exceptions," for a detailed discussion of the general and SIMD floating-point exceptions that can be generated with the Intel SSE instructions and for guidelines for handling these exceptions when they occur.

10.7  WRITING APPLICATIONS WITH INTEL® SSE

See Section 11.6, "Writing Applications with Intel® SSE and SSE2," for additional information about writing applications and operating-system code using Intel SSE.
The streaming SIMD extensions 2 (SSE2) were introduced into the IA-32 architecture in the Pentium 4 and Intel Xeon processors. These extensions enhance the performance of IA-32 processors for advanced 3-D graphics, video decoding/encoding, speech recognition, E-commerce, Internet, scientific, and engineering applications.

This chapter describes the SSE2 extensions and provides information to assist in writing application programs that use these and the SSE extensions.

11.1 OVERVIEW OF INTEL® SSE2

Intel SSE2 uses the single instruction multiple data (SIMD) execution model that is used with MMX technology and Intel SSE. They extend this model with support for packed double precision floating-point values and for 128-bit packed integers.

If CPUID.01H:EDX.SSE2[bit 26] = 1, Intel SSE2 is present.

Intel SSE2 adds the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 processors, applications, and operating systems.

• Six data types:
  — 128-bit packed double precision floating-point (two IEEE Standard 754 double precision floating-point values packed into a double quadword).
  — 128-bit packed byte integers.
  — 128-bit packed word integers.
  — 128-bit packed doubleword integers.
  — 128-bit packed quadword integers.

• Instructions to support the additional data types and extend existing SIMD integer operations:
  — Packed and scalar double precision floating-point instructions.
  — Additional 64-bit and 128-bit SIMD integer instructions.
  — 128-bit versions of SIMD integer instructions introduced with the MMX technology and Intel SSE.
  — Additional cacheability-control and instruction-ordering instructions.

• Modifications to existing IA-32 instructions to support Intel SSE2 features:
  — Extensions and modifications to the CPUID instruction.
  — Modifications to the RDPMC instruction.

These new features extend the IA-32 architecture’s SIMD programming model in three important ways:

• They provide the ability to perform SIMD operations on pairs of packed double precision floating-point values. This permits higher precision computations to be carried out in XMM registers, which enhances processor performance in scientific and engineering applications and in applications that use advanced 3-D geometry techniques (such as ray tracing). Additional flexibility is provided with instructions that operate on single (scalar) double precision floating-point values located in the low quadword of an XMM register.

• They provide the ability to operate on 128-bit packed integers (bytes, words, doublewords, and quadwords) in XMM registers. This provides greater flexibility and greater throughput when performing SIMD operations on packed integers. The capability is particularly useful for applications such as RSA authentication and RC5 encryption. Using the full set of SIMD registers, data types, and instructions provided with the MMX technology and Intel SSE/SSE2, programmers can develop algorithms that finely mix packed single- and double precision floating-point data and 64- and 128-bit packed integer data.

• Intel SSE2 enhances the support introduced with Intel SSE for controlling the cacheability of SIMD data. Intel SSE2 cache control instructions provide the ability to stream data in and out of the XMM registers without polluting the caches and the ability to prefetch data before it is actually used.
Intel SSE2 is fully compatible with all software written for IA-32 processors. All existing software continues to run correctly, without modification, on processors that incorporate Intel SSE2, as well as in the presence of applications that incorporate these extensions. Enhancements to the CPUID instruction permit detection of Intel SSE2. Also, because Intel SSE2 uses the same registers as Intel SSE, no new operating-system support is required for saving and restoring program state during a context switch beyond that provided for Intel SSE.

Intel SSE2 is accessible from all IA-32 execution modes: protected mode, real address mode, and virtual 8086 mode.

The following sections in this chapter describe the programming environment for Intel SSE2, including: the 128-bit XMM floating-point register set, data types, and Intel SSE2 instructions. The chapter also describes exceptions that can be generated with the Intel SSE and SSE2 instructions and gives guidelines for writing applications with Intel SSE and SSE2.

For additional information about Intel SSE2, see:
• The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, provides a detailed description of individual Intel SSE2 instructions.

11.2 INTEL® SSE2 PROGRAMMING ENVIRONMENT

Figure 11-1 shows the programming environment for Intel SSE2. No new registers or other instruction execution state are defined with Intel SSE2. Intel SSE2 instructions use XMM registers, MMX registers, and/or IA-32 general-purpose registers, as follows:

• **XMM registers** — These eight registers (see Figure 10-2) are used to operate on packed or scalar double precision floating-point data. Scalar operations are operations performed on individual (unpacked) double precision floating-point values stored in the low quadword of an XMM register. XMM registers are also used to perform operations on 128-bit packed integer data. They are referenced by the names XMM0 through XMM7.

  - **MXCSR register** — This 32-bit register (see Figure 10-3) provides status and control bits used in floating-point operations. The denormals-are-zeros and flush-to-zero flags in this register provide a higher performance alternative for the handling of denormal source operands and denormal (underflow) results. For more
information on the functions of these flags see Section 10.2.3.4, “Denormals-Are-Zeros,” and Section 10.2.3.3, “Flush-To-Zero.”

- **MMX registers** — These eight registers (see Figure 9-2) are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between MMX and XMM registers. MMX registers are referenced by the names MM0 through MM7.

- **General-purpose registers** — The eight general-purpose registers (see Figure 3-5) are used along with the existing IA-32 addressing modes to address operands in memory. MMX and XMM registers cannot be used to address memory. The general-purpose registers are also used to hold operands for some SSE2 instructions. These registers are referenced by the names EAX, EBX, ECX, EDX, EBP, ESI, EDI, and ESP.

- **EFLAGS register** — This 32-bit register (see Figure 3-8) is used to record the results of some compare operations.

### 11.2.1 Intel® SSE2 in 64-Bit Mode and Compatibility Mode

In compatibility mode, Intel SSE2 functions like it does in protected mode. In 64-bit mode, eight additional XMM registers are accessible. Registers XMM8-XMM15 are accessed by using REX prefixes.

Memory operands are specified using the ModR/M, SIB encoding described in Section 3.7.5.

Some Intel SSE2 instructions may be used to operate on general-purpose registers. Use the REX.W prefix to access 64-bit general-purpose registers. Note that if a REX prefix is used when it has no meaning, the prefix is ignored.

### 11.2.2 Compatibility of Intel® SSE2 with Intel® SSE, MMX Technology, and x87 FPU Programming Environment

Intel SSE2 does not introduce any new state to the IA-32 execution environment beyond that of Intel SSE. Intel SSE2 represents an enhancement of Intel SSE; they are fully compatible and share the same state information. Intel SSE and SSE2 instructions can be executed together in the same instruction stream without the need to save state when switching between instruction sets.

XMM registers are independent of the x87 FPU and MMX registers; so Intel SSE and SSE2 operations performed on XMM registers can be performed in parallel with x87 FPU or MMX technology operations; see Section 11.6.7, “Interaction of Intel® SSE and SSE2 Instructions with x87 FPU and MMX Instructions.”

The FXSAVE and FXRSTOR instructions save and restore the SSE and SSE2 states along with the x87 FPU and MMX states.

### 11.2.3 Denormals-Are-Zeros Flag

The denormals-are-zeros flag (bit 6 in the MXCSR register) was introduced into the IA-32 architecture with Intel SSE2. See Section 10.2.3.4, “Denormals-Are-Zeros,” for a description of this flag.

### 11.3 INTEL® SSE2 DATA TYPES

Intel SSE2 introduced one 128-bit packed floating-point data type and four 128-bit SIMD integer data types to the IA-32 architecture (see Figure 11-2).

- **Packed double precision floating-point** — This 128-bit data type consists of two IEEE 64-bit double precision floating-point values packed into a double quadword. See Figure 4-3 for the layout of a 64-bit double precision floating-point value; refer to Section 4.2.2, “Floating-Point Data Types,” for a detailed description of double precision floating-point values.

- **128-bit packed integers** — The four 128-bit packed integer data types can contain 16 byte integers, 8 word integers, 4 doubleword integers, or 2 quadword integers. Refer to Section 4.6.2, “128-Bit Packed SIMD Data Types,” for a detailed description of the 128-bit packed integers.
All of these data types are operated on in XMM registers or memory. Instructions are provided to convert between these 128-bit data types and the 64-bit and 32-bit data types.

The address of a 128-bit packed memory operand must be aligned on a 16-byte boundary, except in the following cases:

- A MOVUPD instruction that supports unaligned accesses.
- Scalar instructions that use an 8-byte memory operand that is not subject to alignment requirements.

Figure 4-2 shows the byte order of 128-bit (double quadword) and 64-bit (quadword) data types in memory.

### 11.4 INTEL® SSE2 INSTRUCTIONS

The Intel SSE2 instructions are divided into four functional groups:

- Packed and scalar double precision floating-point instructions.
- 64-bit and 128-bit SIMD integer instructions.
- 128-bit extensions of SIMD integer instructions introduced with the MMX technology and Intel SSE.
- Cacheability-control and instruction-ordering instructions.

The following sections provide more information about each group.

#### 11.4.1 Packed and Scalar Double Precision Floating-Point Instructions

The packed and scalar double precision floating-point instructions are divided into the following sub-groups:

- Data movement instructions.
- Arithmetic instructions.
- Comparison instructions.
- Conversion instructions.
- Logical instructions.
- Shuffle instructions.

The packed double precision floating-point instructions perform SIMD operations similarly to the packed single precision floating-point instructions (see Figure 11-3). Each source operand contains two double precision floating-
point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, and X1 and Y1) in each operand.

![Figure 11-3. Packed Double Precision Floating-Point Operations](image)

The scalar double precision floating-point instructions operate on the low (least significant) quadwords of two source operands (X0 and Y0), as shown in Figure 11-4. The high quadword (X1) of the first source operand is passed through to the destination. The scalar operations are similar to the floating-point operations performed in x87 FPU data registers with the precision control field in the x87 FPU control word set for double precision (53-bit significand), except that x87 stack operations use a 15-bit exponent range for the result while Intel SSE2 operations use an 11-bit exponent range.

See Section 11.6.8, “Compatibility of SIMD and x87 FPU Floating-Point Data Types,” for more information about obtaining compatible results when performing both scalar double precision floating-point operations in XMM registers and in x87 FPU data registers.

![Figure 11-4. Scalar Double Precision Floating-Point Operations](image)

### 11.4.1.1 Data Movement Instructions

Data movement instructions move double precision floating-point data between XMM registers and between XMM registers and memory.

The MOVAPD (move aligned packed double precision floating-point) instruction transfers a 128-bit packed double precision floating-point operand from memory to an XMM register or vice versa, or between XMM registers. The memory address must be aligned to a 16-byte boundary; if not, a general-protection exception (GP#) is generated.
The MOVUPD (move unaligned packed double precision floating-point) instruction transfers a 128-bit packed double precision floating-point operand from memory to an XMM register or vice versa, or between XMM registers. Alignment of the memory address is not required.

The MOVSD (move scalar double precision floating-point) instruction transfers a 64-bit double precision floating-point operand from memory to the low quadword of an XMM register or vice versa, or between XMM registers. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVPD (move packed double precision floating-point) instruction transfers a 64-bit double precision floating-point operand from memory to the high quadword of an XMM register or vice versa. The low quadword of the register is left unchanged. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVLPD (move low packed double precision floating-point) instruction transfers a 64-bit double precision floating-point operand from memory to the low quadword of an XMM register or vice versa. The high quadword of the register is left unchanged. Alignment of the memory address is not required, unless alignment checking is enabled.

The MOVMSKPD (move packed double precision floating-point mask) instruction extracts the sign bit of each of the two packed double precision floating-point numbers in an XMM register and saves them in a general-purpose register. This 2-bit value can then be used as a condition to perform branching.

11.4.1.2 Intel® SSE2 Arithmetic Instructions

Intel SSE2 arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double precision floating-point values.

The ADDPD (add packed double precision floating-point values) and SUBPD (subtract packed double precision floating-point values) instructions add and subtract, respectively, two packed double precision floating-point operands.

The ADDSD (add scalar double precision floating-point values) and SUBSD (subtract scalar double precision floating-point values) instructions add and subtract, respectively, the low double precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The MULPD (multiply packed double precision floating-point values) instruction multiplies two packed double precision floating-point operands.

The MULPSD (multiply scalar double precision floating-point values) instruction multiplies the low double precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The DIVPD (divide packed double precision floating-point values) instruction divides two packed double precision floating-point operands.

The DIVPSD (divide scalar double precision floating-point values) instruction divides the low double precision floating-point values of two operands and stores the result in the low quadword of the destination operand.

The SQRTPD (compute square roots of packed double precision floating-point values) instruction computes the square roots of the values in a packed double precision floating-point operand.

The SQRTPSD (compute square root of scalar double precision floating-point values) instruction computes the square root of the low double precision floating-point value in the source operand and stores the result in the low quadword of the destination operand.

The MAXPD (return maximum of packed double precision floating-point values) instruction compares the corresponding values in two packed double precision floating-point operands and returns the numerically greater value from each comparison to the destination operand.

The MAXPSD (return maximum of scalar double precision floating-point values) instruction compares the low double precision floating-point values from two packed double precision floating-point operands and returns the numerically higher value from the comparison to the low quadword of the destination operand.

The MINPD (return minimum of packed double precision floating-point values) instruction compares the corresponding values from two packed double precision floating-point operands and returns the numerically lesser value from each comparison to the destination operand.
The MINSD (return minimum of scalar double precision floating-point values) instruction compares the low values from two packed double precision floating-point operands and returns the numerically lesser value from the comparison to the low quadword of the destination operand.

11.4.1.3 Intel® SSE2 Logical Instructions

Intel® SSE2 logical instructions perform AND, AND NOT, OR, and XOR operations on packed double precision floating-point values.

The ANDPD (bitwise logical AND of packed double precision floating-point values) instruction returns the logical AND of two packed double precision floating-point operands.

The ANDNPD (bitwise logical AND NOT of packed double precision floating-point values) instruction returns the logical AND NOT of two packed double precision floating-point operands.

The ORPD (bitwise logical OR of packed double precision floating-point values) instruction returns the logical OR of two packed double precision floating-point operands.

The XORPD (bitwise logical XOR of packed double precision floating-point values) instruction returns the logical XOR of two packed double precision floating-point operands.

11.4.1.4 Intel® SSE2 Comparison Instructions

Intel® SSE2 compare instructions compare packed and scalar double precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

The CMPPD (compare packed double precision floating-point values) instruction compares the corresponding values from two packed double precision floating-point operands, using an immediate operand as a predicate, and returns a 64-bit mask result of all 1s or all 0s for each comparison to the destination operand. The value of the immediate operand allows the selection of any of eight compare conditions: equal, less than, less than equal, unordered, not equal, not less than, not less than or equal, or ordered.

The CMPSD (compare scalar double precision floating-point values) instruction compares the low values from two packed double precision floating-point operands, using an immediate operand as a predicate, and returns a 64-bit mask result of all 1s or all 0s for the comparison to the low quadword of the destination operand. The immediate operand selects the compare condition as with the CMPPD instruction.

The COMISD (compare scalar double precision floating-point values and set EFLAGS) and UCOMISD (unordered compare scalar double precision floating-point values and set EFLAGS) instructions compare the low values of two packed double precision floating-point operands and set the ZF, PF, and CF flags in the EFLAGS register to show the result (greater than, less than, equal, or unordered). These two instructions differ as follows: the COMISD instruction signals a floating-point invalid-operation (#I) exception when a source operand is either a QNaN or an SNaN; the UCOMISD instruction only signals an invalid-operation exception when a source operand is an SNaN.

11.4.1.5 Intel® SSE2 Shuffle and Unpack Instructions

Intel® SSE2 shuffle instructions shuffle the contents of two packed double precision floating-point values and store the results in the destination operand.

The SHUFPD (shuffle packed double precision floating-point values) instruction places either of the two packed double precision floating-point values from the destination operand in the low quadword of the destination operand, and places either of the two packed double precision floating-point values from source operand in the high quadword of the destination operand (see Figure 11-5). By using the same register for the source and destination operands, the SHUFPD instruction can swap two packed double precision floating-point values.
The UNPCKHPD (unpack and interleave high packed double precision floating-point values) instruction performs an interleaved unpack of the high values from the source and destination operands and stores the result in the destination operand (see Figure 11-6).

The UNPCKLPD (unpack and interleave low packed double precision floating-point values) instruction performs an interleaved unpack of the low values from the source and destination operands and stores the result in the destination operand (see Figure 11-7).
11.4.1.6 Intel® SSE2 Conversion Instructions

Intel SSE2 conversion instructions (see Figure 11-8) support packed and scalar conversions between:

- Double precision and single precision floating-point formats.
- Double precision floating-point and doubleword integer formats.
- Single precision floating-point and doubleword integer formats.

**Conversion between double precision and single precision floating-points values** — The following instructions convert operands between double precision and single precision floating-point formats. The operands being operated on are contained in XMM registers or memory (at most, one operand can reside in memory; the destination is always an MMX register).

The CVTPS2PD (convert packed single precision floating-point values to packed double precision floating-point values) instruction converts two packed single precision floating-point values to two double precision floating-point values.

The CVTPD2PS (convert packed double precision floating-point values to packed single precision floating-point values) instruction converts two packed double-precision floating-point values to two single precision floating-point values. When a conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

The CVTSS2SD (convert scalar single precision floating-point value to scalar double precision floating-point value) instruction converts a single precision floating-point value to a double precision floating-point value.

The CVTSD2SS (convert scalar double precision floating-point value to scalar single precision floating-point value) instruction converts a double precision floating-point value to a single precision floating-point value. When the conversion is inexact, the result is rounded according to the rounding mode selected in the MXCSR register.

Conversion between double precision floating-point values and doubleword integers — The following instructions convert operands between double precision floating-point and doubleword integer formats. Operands...
are housed in XMM registers, MMX registers, general registers or memory (at most one operand can reside in memory; the destination is always an XMM, MMX, or general register).

The CVTDP2PI (convert packed double precision floating-point values to packed doubleword integers) instruction converts two packed double precision floating-point numbers to two packed signed doubleword integers, with the result stored in an MMX register. When rounding to an integer value, the source value is rounded according to the rounding mode in the MXCSR register. The CVTDP2PI (convert with truncation packed double precision floating-point values to packed doubleword integers) instruction is similar to the CVTDP2PI instruction except that truncation is used to round a source value to an integer value; see Section 4.8.4.2, "Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions."

The CVTPI2PD (convert packed doubleword integers to packed double precision floating-point values) instruction converts two packed signed doubleword integers to two double precision floating-point values.

The CVTDP2DQ (convert packed double precision floating-point values to packed doubleword integers) instruction converts two packed double precision floating-point numbers to two packed signed doubleword integers, with the result stored in the low quadword of an XMM register. When rounding an integer value, the source value is rounded according to the rounding mode selected in the MXCSR register. The CVTDP2DQ (convert with truncation packed double precision floating-point values to packed doubleword integers) instruction is similar to the CVTDP2DQ instruction except that truncation is used to round a source value to an integer value; see Section 4.8.4.2, "Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions."

The CVTDQ2PD (convert packed doubleword integers to packed double precision floating-point values) instruction converts two packed signed doubleword integers located in the low-order doublewords of an XMM register to two double precision floating-point values.

The CVTSD2SI (convert scalar double precision floating-point value to doubleword integer) instruction converts a double precision floating-point value to a doubleword integer, and stores the result in a general-purpose register. When rounding an integer value, the source value is rounded according to the rounding mode selected in the MXCSR register. The CVTSD2SI (convert with truncation scalar double precision floating-point value to doubleword integer) instruction is similar to the CVTSD2SI instruction except that truncation is used to round the source value to an integer value; see Section 4.8.4.2, "Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions."

The CVTSI2SD (convert doubleword integer to scalar double precision floating-point value) instruction converts a signed doubleword integer in a general-purpose register to a double precision floating-point number, and stores the result in an XMM register.

**Conversion between single precision floating-point and doubleword integer formats —** These instructions convert between packed single precision floating-point and packed doubleword integer formats. Operands are housed in XMM registers, MMX registers, general registers, or memory (the latter for at most one source operand). The destination is always an XMM, MMX, or general register. These SSE2 instructions supplement conversion instructions (CVTPI2PS, CVTPS2PI, CVTTPS2PI, CVTSI2SS, CVTSS2SI, and CVTTSS2SI) introduced with Intel SSE extensions.

The CVTTPS2DQ (convert packed single precision floating-point values to packed doubleword integers) instruction converts four packed single precision floating-point values to four packed signed doubleword integers, with the source and destination operands in XMM registers or memory (the latter for at most one source operand). When the conversion is inexact, the rounded value according to the rounding mode selected in the MXCSR register is returned. The CVTTPS2DQ (convert with truncation packed single precision floating-point values to packed doubleword integers) instruction is similar to the CVTTPS2DQ instruction except that truncation is used to round a source value to an integer value; see Section 4.8.4.2, "Truncation with Intel® SSE, SSE2, and AVX Conversion Instructions."

The CVTDQ2PS (convert packed doubleword integers to packed single precision floating-point values) instruction converts four packed signed doubleword integers to four packed single precision floating-point numbers, with the source and destination operands in XMM registers or memory (the latter for at most one source operand). When the conversion is inexact, the rounded value according to the rounding mode selected in the MXCSR register is returned.
11.4.2 Intel® SSE2 64-Bit and 128-Bit SIMD Integer Instructions

Intel SSE2 adds several 128-bit packed integer instructions to the IA-32 architecture. Where appropriate, a 64-bit version of each of these instructions is also provided. The 128-bit versions of instructions operate on data in XMM registers; 64-bit versions operate on data in MMX registers. The instructions follow.

The MOVDQA (move aligned double quadword) instruction transfers a double quadword operand from memory to an XMM register or vice versa; or between XMM registers. The memory address must be aligned to a 16-byte boundary; otherwise, a general-protection exception (#GP) is generated.

The MOVDQU (move unaligned double quadword) instruction performs the same operations as the MOVDQA instruction, except that 16-byte alignment of a memory address is not required.

The PADDQ (packed quadword add) instruction adds two packed quadword integer operands or two single quadword integer operands, and stores the results in an XMM or MMX register, respectively. This instruction can operate on either unsigned or signed (two's complement notation) integer operands.

The PSUBQ (packed quadword subtract) instruction subtracts two packed quadword integer operands or two single quadword integer operands, and stores the results in an XMM or MMX register, respectively. Like the PADDQ instruction, PSUBQ can operate on either unsigned or signed (two’s complement notation) integer operands.

The PMULUDQ (multiply packed unsigned doubleword integers) instruction performs an unsigned multiply of unsigned doubleword integers and returns a quadword result. Both 64-bit and 128-bit versions of this instruction are available. The 64-bit version operates on two doubleword integers stored in the low doubleword of each source operand, and the quadword result is returned to an MMX register. The 128-bit version performs a packed multiply of two pairs of doubleword integers. Here, the doublewords are packed in the first and third doublewords of the source operands, and the quadword results are stored in the low and high quadwords of an XMM register.

The PSHUFLW (shuffle packed low words) instruction shuffles the word integers packed into the low quadword of the source operand and stores the shuffled result in the low quadword of the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSHUFW (shuffle packed high words) instruction shuffles the word integers packed into the high quadword of the source operand and stores the shuffled result in the high quadword of the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSHUFD (shuffle packed doubleword integers) instruction shuffles the doubleword integers packed into the source operand and stores the shuffled result in the destination operand. An 8-bit immediate operand specifies the shuffle order.

The PSLLDQ (shift double quadword left logical) instruction shifts the contents of the source operand to the left by the amount of bytes specified by an immediate operand. The empty low-order bytes are cleared (set to 0).

The PSRLDQ (shift double quadword right logical) instruction shifts the contents of the source operand to the right by the amount of bytes specified by an immediate operand. The empty high-order bytes are cleared (set to 0).

The PUNPCKHQDQ (Unpack high quadwords) instruction interleaves the high quadword of the source operand and the high quadword of the destination operand and writes them to the destination register.

The PUNPCKLQDQ (Unpack low quadwords) instruction interleaves the low quadwords of the source operand and the low quadwords of the destination operand and writes them to the destination register.

Two additional SSE instructions enable data movement from the MMX registers to the XMM registers.

The MOVQ2DQ (move quadword integer from MMX to XMM registers) instruction moves the quadword integer from an MMX source register to an XMM destination register.

The MOVQ2Q (move quadword integer from XMM to MMX registers) instruction moves the low quadword integer from an XMM source register to an MMX destination register.

11.4.3 128-Bit SIMD Integer Instruction Extensions

All of 64-bit SIMD integer instructions introduced with MMX technology and Intel SSE (with the exception of the PSHUFW instruction) have been extended by Intel SSE2 to operate on 128-bit packed integer operands located in XMM registers. The 128-bit versions of these instructions follow the same SIMD conventions regarding packed...
operands as the 64-bit versions. For example, where the 64-bit version of the PADDB instruction operates on 8 packed bytes, the 128-bit version operates on 16 packed bytes.

### 11.4.4 Cacheability Control and Memory Ordering Instructions

Intel SSE2 instructions that give programs more control over the caching, loading, and storing of data. are described below.

#### 11.4.4.1 FLUSH Cache Line

The CLFLUSH (flush cache line) instruction writes and invalidates the cache line associated with a specified linear address. The invalidation is for all levels of the processor’s cache hierarchy, and it is broadcast throughout the cache coherency domain.

**NOTE**

CLFLUSH was introduced with Intel SSE2. However, the instruction can be implemented in IA-32 processors that do not implement Intel SSE2. Detect CLFLUSH using the feature bit (if CPUID.01H:EDX.CLFSH[bit 19] = 1).

#### 11.4.4.2 Cacheability Control Instructions

The following four instructions enable data from XMM and general-purpose registers to be stored to memory using a non-temporal hint. The non-temporal hint directs the processor to store data to memory without writing the data into the cache hierarchy. See Section 10.4.6.2, “Caching of Temporal vs. Non-Temporal Data,” for more information about non-temporal stores and hints.

- The MOVNTDQ (store double quadword using non-temporal hint) instruction stores packed integer data from an XMM register to memory, using a non-temporal hint.
- The MOVNTPD (store packed double precision floating-point values using non-temporal hint) instruction stores packed double precision floating-point data from an XMM register to memory, using a non-temporal hint.
- The MOVNTI (store doubleword using non-temporal hint) instruction stores integer data from a general-purpose register to memory, using a non-temporal hint.
- The MASKMOVDQU (store selected bytes of double quadword) instruction stores selected byte integers from an XMM register to memory, using a byte mask to selectively write the individual bytes. The memory location does not need to be aligned on a natural boundary. This instruction also uses a non-temporal hint.

#### 11.4.4.3 Memory Ordering Instructions

Intel SSE2 introduced two fence instructions (LFENCE and MFENCE) as companions to the SFENCE instruction introduced with Intel SSE.

The LFENCE instruction establishes a memory fence for loads. It guarantees ordering between two loads and prevents speculative loads from passing the load fence (that is, no speculative loads are allowed until all loads specified before the load fence have been carried out).

The MFENCE instruction establishes a memory fence for both loads and stores. The processor ensures that no load or store after MFENCE will become globally visible until all loads and stores before MFENCE are globally visible.\(^1\)

Note that the sequences LFENCE;SFENCE and SFENCE;LFENCE are not equivalent to MFENCE because neither ensures that older stores are globally observed prior to younger loads.

#### 11.4.4.4 Pause

The PAUSE instruction is provided to improve the performance of “spin-wait loops” executed on a Pentium 4 or Intel Xeon processor. On a Pentium 4 processor, it also provides the added benefit of reducing processor power.

\(^1\) A load is considered to become globally visible when the value to be loaded is determined.
consumption while executing a spin-wait loop. It is recommended that a PAUSE instruction always be included in
the code sequence for a spin-wait loop.

11.4.5 Branch Hints

Intel SSE2 designates two instruction prefixes (2EH and 3EH) to provide branch hints to the processor (see
"Instruction Prefixes" in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume
2A). These prefixes can only be used with the Jcc instruction and only at the machine code level (that is, there are
no mnemonics for the branch hints).

11.5 INTEL® SSE, SSE2, AND SSE3 EXCEPTIONS

Intel SSE, SSE2, and SSE3 instructions generate two general types of exceptions:
• Non-numeric exceptions.
• SIMD floating-point exceptions.¹

Intel SSE, SSE2, and SSE3 instructions can generate the same type of memory-access and non-numeric excep-
tions as other IA-32 architecture instructions. Existing exception handlers can generally handle these exceptions
without any code modification. See "Providing Non-Numeric Exception Handlers for Exceptions Generated by the
SSE, SSE2, and SSE3 Instructions" in Chapter 14 of the Intel® 64 and IA-32 Architectures Software Developer’s
Manual, Volume 3A, for a list of the non-numeric exceptions that can be generated by the Intel SSE, SSE2, SSE3
instructions and for guidelines for handling these exceptions.

Intel SSE, SSE2, and SSE3 instructions do not generate numeric exceptions on packed integer operations;
however, they can generate numeric (SIMD floating-point) exceptions on packed single precision and double preci-
sion floating-point operations. These SIMD floating-point exceptions are defined in the IEEE Standard 754 for
Floating-Point Arithmetic and are the same exceptions that are generated for x87 FPU instructions. See Section
11.5.1, “SIMD Floating-Point Exceptions,” for a description of these exceptions.

11.5.1 SIMD Floating-Point Exceptions

SIMD floating-point exceptions are those exceptions that can be generated by Intel SSE, SSE2, and SSE3 instruc-
tions that operate on packed or scalar floating-point operands.

Six classes of SIMD floating-point exceptions can be generated:
• Invalid operation (#I).
• Divide-by-zero (#Z).
• Denormal operand (#D).
• Numeric overflow (#O).
• Numeric underflow (#U).
• Inexact result (Precision) (#P).

All of these exceptions (except the denormal operand exception) are defined in IEEE Standard 754, and they are
the same exceptions that are generated with the x87 floating-point instructions. Section 4.9, "Overview of
Floating-Point Exceptions," gives a detailed description of these exceptions and of how and when they are gener-
ated. The following sections discuss the implementation of these exceptions in the Intel SSE/SSE2/SSE3 exten-
sions.

All SIMD floating-point exceptions are precise and occur as soon as the instruction completes execution.

¹ The FISTTP instruction in Intel SSE3 does not generate SIMD floating-point exceptions, but it can generate x87 FPU floating-point
exceptions.
11.5.2 SIMD Floating-Point Exception Conditions

Each of the six exception conditions has a corresponding flag (IE, DE, ZE, OE, UE, and PE) and mask bit (IM, DM, ZM, OM, UM, and PM) in the MXCSR register (see Figure 10-3). The mask bits can be set with the LDMXCSR or FXRSTOR instruction; the mask and flag bits can be read with the STMXCSR or FXSAVE instruction.

The OSXMMEXCEPT flag (bit 10) of control register CR4 provides additional control over generation of SIMD floating-point exceptions by allowing the operating system to indicate whether or not it supports software exception handlers for SIMD floating-point exceptions. If an unmasked SIMD floating-point exception is generated and the OSXMMEXCEPT flag is set, the processor invokes a software exception handler by generating a SIMD floating-point exception (#XM). If the OSXMMEXCEPT bit is clear, the processor generates an invalid-opcode exception (#UD) on the first Intel SSE or SSE2 instruction that detects a SIMD floating-point exception condition. See Section 11.6.2, “Checking for Intel® SSE and SSE2 Support.”

11.5.2.1 Invalid Operation Exception (#I)

The floating-point invalid-operation exception (#I) occurs in response to an invalid arithmetic operand. The flag (IE) and mask (IM) bits for the invalid operation exception are bits 0 and 7, respectively, in the MXCSR register.

If the invalid-operation exception is masked, the processor returns a QNaN, QNaN floating-point indefinite, integer indefinite, one of the source operands to the destination operand, or it sets the EFLAGS, depending on the operation being performed. When a value is returned to the destination operand, it overwrites the destination register specified by the instruction. Table 11-1 lists the invalid-arithmetic operations that the processor detects for instructions and the masked responses to these operations.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Masked Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, ADDSUBPD, ADDSUBSP, HADDPD, HADDPD, HADDPS, HSUBPD or HSUBPS instruction with an SNaN operand</td>
<td>Return the SNaN converted to a QNaN; Refer to Table 4-7 for more details.</td>
</tr>
<tr>
<td>SQRTPS, SQRTPS, SQRTPS, or SQRTPS with SNaN operands</td>
<td>Return the SNaN converted to a QNaN.</td>
</tr>
<tr>
<td>SQRTPS, SQRTPS, SQRTPS, or SQRTPS with negative operands (except zero)</td>
<td>Return the QNaN floating-point Indefinite.</td>
</tr>
<tr>
<td>MAXPS, MAXSS, MAXPD, MAXSD, MINPS, MINSS, MINPD, or MINSD instruction with QNaN or SNaN operands</td>
<td>Return the source 2 operand value.</td>
</tr>
<tr>
<td>CMPPS, CMPSS, CMPPD or CMPPD instruction with QNaN or SNaN operands</td>
<td>Return a mask of all 0s (except for the predicates “not-equal,” “unordered,” “not-less-than,” or “not-less-than-or-equal,” which returns a mask of all 1s).</td>
</tr>
<tr>
<td>CVTPD2PS, CVTSS2SS, CVTSPS2PD, CVTSS2SD with SNaN operands</td>
<td>Return the SNaN converted to a QNaN.</td>
</tr>
<tr>
<td>COMISS or COMISS with QNaN or SNaN operand(s)</td>
<td>Set EFLAGS values to “not comparable.”</td>
</tr>
<tr>
<td>Addition of opposite signed infinities or subtraction of like-signed infinities</td>
<td>Return the QNaN floating-point Indefinite.</td>
</tr>
<tr>
<td>Multiplication of infinity by zero</td>
<td>Return the QNaN floating-point Indefinite.</td>
</tr>
<tr>
<td>Divide of (0/0) or (∞ / ∞)</td>
<td>Return the QNaN floating-point Indefinite.</td>
</tr>
</tbody>
</table>
If the invalid operation exception is not masked, a software exception handler is invoked and the operands remain unchanged. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software."

Normally, when one or more of the source operands are QNaNs (and neither is an SNaN or in an unsupported format), an invalid-operation exception is not generated. The following instructions are exceptions to this rule: the COMISS and COMISD instructions; and the CMPPS, CMPSS, CMPPD, and CMPSD instructions (when the predicate is less than, less-than or equal, not less-than, or not less-than or equal). With these instructions, a QNaN source operand will generate an invalid-operation exception.

The invalid-operation exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

### 11.5.2.2 Denormal-Operand Exception (#D)

The processor signals the denormal-operand exception if an arithmetic instruction attempts to operate on a denormal operand. The flag (DE) and mask (DM) bits for the denormal-operand exception are bits 1 and 8, respectively, in the MXCSR register.

The CVTPI2PD, CVTTPD2PI, CVTDQ2PD, CVTDP2DQ, CVTTPD2DQ, CVTSI2SD, CVTSD2SI, CVTTSD2SI, CVTI2PS, CVTSS2SI, CVTTSS2SI, CVTSI2SS, CVTTSD2SS, CVTDQ2PS, CVTSD2Q, and CVTTPD2Q conversion instructions do not signal denormal exceptions. The RCPSS, RCPPS, RSQRTSS, and RSQRTPS instructions do not signal any kind of floating-point exception.

The denormals-are-zero flag (bit 6) of the MXCSR register provides an additional option for handling denormal-operand exceptions. When this flag is set, denormal source operands are automatically converted to zeros with the sign of the source operand (see Section 10.2.3.4, "Denormals-Are-Zeros"). The denormal operand exception is not affected by the flush-to-zero mode.

See Section 4.9.1.2, "Denormal Operand Exception (#D)," for more information about the denormal exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

### 11.5.2.3 Divide-By-Zero Exception (#Z)

The processor reports a divide-by-zero exception when a DIVPS, DIVSS, DIVPD or DIVSD instruction attempts to divide a finite non-zero operand by 0. The flag (ZE) and mask (ZM) bits for the divide-by-zero exception are bits 2 and 9, respectively, in the MXCSR register.

The divide-by-zero exception is not affected by the flush-to-zero mode at a single-instruction boundary. While DAZ does not affect the rules for signaling IEEE exceptions, operations on denormal inputs might have different results when DAZ=1. As a consequence, DAZ can have an effect on the floating-point exceptions - including the divide-by-zero exception - when observed for a given operation involving denormal inputs.

### 11.5.2.4 Numeric Overflow Exception (#O)

The processor reports a numeric overflow exception whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that fits in the destination operand. This exception can be generated with the ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, CVTDPD2PS, CVTSD2SS, ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD, and

**Table 11-1. Masked Responses of Intel® SSE, SSE2, and SSE3 Instructions to Invalid Arithmetic Operations (Contd.)**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Masked Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion to integer when the value in the source register is a NaN, –∞, or exceeds the representable range for CVTPI2PD, CVTTPD2PI, CVTSS2SI, CVTTSS2SI, CVTDP2PI, CVTSD2SI, CVTDPD2DQ, CVTTPD2DQ, CVTSD2DQ, or CVTTPD2DQ</td>
<td>Return the integer Indefinite.</td>
</tr>
</tbody>
</table>
HSUBPS instructions. The flag (OE) and mask (OM) bits for the numeric overflow exception are bits 3 and 10, respectively, in the MXCSR register.

See Section 4.9.1.4, "Numeric Overflow Exception (#O)," for more information about the numeric-overflow exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

The numeric overflow exception is not affected by the flush-to-zero mode or by the denormals-are-zeros mode.

11.5.2.5  Numeric Underflow Exception (#U)

The processor reports a numeric underflow exception whenever the magnitude of the rounded result of an arithmetic instruction, with unbounded exponent, is less than the smallest possible normalized, finite value that will fit in the destination operand and the numeric-underflow exception is not masked. If the numeric underflow exception is masked, both underflow and the inexact-result condition must be detected before numeric underflow is reported. This exception can be generated with the ADDPS, ADDSS, ADDPD, ADDSD, SUBPS, SUBSS, SUBPD, SUBSD, MULPS, MULSS, MULPD, MULSD, DIVPS, DIVSS, DIVPD, DIVSD, CVTTPD2PS, CVTSS2PS, CVTSD2SS, ADDSUBPD, ADDSUBPS, HADDPD, HADDSS, HSUBPD, and HSUBPS instructions. The flag (UE) and mask (UM) bits for the numeric underflow exception are bits 4 and 11, respectively, in the MXCSR register.

The flush-to-zero flag (bit 15) of the MXCSR register provides an additional option for handling numeric underflow exceptions. When this flag is set and the numeric underflow exception is masked, tiny results are returned as a zero with the sign of the true result; see Section 10.2.3.3, "Flush-To-Zero."

Underflow will occur when a tiny non-zero result is detected (the result has to be also inexact if underflow exceptions are masked), as described in the IEEE Standard 754-2008. While DAZ does not affect the rules for signaling IEEE exceptions, operations on denormal inputs might have different results when DAZ=1. As a consequence, DAZ can have an effect on the floating-point exceptions - including the underflow exception - when observed for a given operation involving denormal inputs.

See Section 4.9.1.5, "Numeric Underflow Exception (#U)," for more information about the numeric underflow exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

11.5.2.6  Inexact-Result (Precision) Exception (#P)

The inexact-result exception (also called the precision exception) occurs if the result of an operation is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (normally acceptable) accuracy has been lost. The exception is supported for applications that need to perform exact arithmetic only. Because the rounded result is generally satisfactory for most applications, this exception is commonly masked.

The flag (PE) and mask (PM) bits for the inexact-result exception are bits 5 and 12, respectively, in the MXCSR register.

See Section 4.9.1.6, "Inexact-Result (Precision) Exception (#P)," for more information about the inexact-result exception. See Section 11.5.4, "Handling SIMD Floating-Point Exceptions in Software," for information on handling unmasked exceptions.

In flush-to-zero mode, the inexact result exception is reported.

11.5.3  Generating SIMD Floating-Point Exceptions

When the processor executes a packed or scalar floating-point instruction, it looks for and reports on SIMD floating-point exception conditions using two sequential steps:

1. Looks for, reports on, and handles pre-computation exception conditions (invalid-operand, divide-by-zero, and denormal operand)
2. Looks for, reports on, and handles post-computation exception conditions (numeric overflow, numeric underflow, and inexact result)
If both pre- and post-computational exceptions are unmasked, it is possible for the processor to generate a SIMD floating-point exception (#XM) twice during the execution of an SSE, SSE2 or SSE3 instruction: once when it detects and handles a pre-computational exception and when it detects a post-computational exception.

### 11.5.3.1 Handling Masked Exceptions

If all exceptions are masked, the processor handles the exceptions it detects by placing the masked result (or results for packed operands) in a destination operand and continuing program execution. The masked result may be a rounded normalized value, signed infinity, a denormal finite number, zero, a QNaN floating-point indefinite, or a QNaN depending on the exception condition detected. In most cases, the corresponding exception flag bit in MXCSR is also set. The one situation where an exception flag is not set is when an underflow condition is detected and it is not accompanied by an inexact result.

When operating on packed floating-point operands, the processor returns a masked result for each of the sub-operand computations and sets a separate set of internal exception flags for each computation. It then performs a logical-OR on the internal exception flag settings and sets the exception flags in the MXCSR register according to the results of OR operations.

For example, Figure 11-9 shows the results of an MULPS instruction. In the example, all SIMD floating-point exceptions are masked. Assume that a denormal exception condition is detected prior to the multiplication of sub-operands X0 and Y0, no exception condition is detected for the multiplication of X1 and Y1, a numeric overflow exception condition is detected for the multiplication of X2 and Y2, and another denormal exception is detected prior to the multiplication of sub-operands X3 and Y3. Because denormal exceptions are masked, the processor uses the denormal source values in the multiplications of (X0 and Y0) and of (X3 and Y3) passing the results of the multiplications through to the destination operand. With the denormal operand, the result of the X0 and Y0 computation is a normalized finite value, with no exceptions detected. However, the X3 and Y3 computation produces a tiny and inexact result. This causes the corresponding internal numeric underflow and inexact-result exception flags to be set.

![Figure 11-9. Example Masked Response for Packed Operations](image)

For the multiplication of X2 and Y2, the processor stores the floating-point $\infty$ in the destination operand, and sets the corresponding internal sub-operand numeric overflow flag. The result of the X1 and Y1 multiplication is passed through to the destination operand, with no internal sub-operand exception flags being set. Following the computations, the individual sub-operand exceptions flags for denormal operand, numeric underflow, inexact result, and numeric overflow are OR’d and the corresponding flags are set in the MXCSR register.

The net result of this computation is that:

- Multiplication of X0 and Y0 produces a normalized finite result
- Multiplication of X1 and Y1 produces a normalized finite result
- Multiplication of X2 and Y2 produces a floating-point $\infty$ result
- Multiplication of X3 and Y3 produces a tiny, inexact, finite result
• Denormal operand, numeric underflow, numeric underflow, and inexact result flags are set in the MXCSR register

11.5.3.2 Handling Unmasked Exceptions

If all exceptions are unmasked, the processor:

1. First detects any pre-computation exceptions: it ORs those exceptions, sets the appropriate exception flags, leaves the source and destination operands unaltered, and goes to step 2. If it does not detect any pre-computation exceptions, it goes to step 5.

2. Checks CR4.OSXMMEXCPT[bit 10]. If this flag is set, the processor goes to step 3; if the flag is clear, it generates an invalid-opcode exception (#UD) and makes an implicit call to the invalid-opcode exception handler.

3. Generates a SIMD floating-point exception (#XM) and makes an implicit call to the SIMD floating-point exception handler.

4. If the exception handler is able to fix the source operands that generated the pre-computation exceptions or mask the condition in such a way as to allow the processor to continue executing the instruction, the processor resumes instruction execution as described in step 5.

5. Upon returning from the exception handler (or if no pre-computation exceptions were detected), the processor checks for post-computation exceptions. If the processor detects any post-computation exceptions: it ORs those exceptions, sets the appropriate exception flags, leaves the source and destination operands unaltered, and repeats steps 2, 3, and 4.

6. Upon returning from the exceptions handler in step 4 (or if no post-computation exceptions were detected), the processor completes the execution of the instruction.

The implication of this procedure is that for unmasked exceptions, the processor can generate a SIMD floating-point exception (#XM) twice: once if it detects pre-computation exception conditions and a second time if it detects post-computation exception conditions. For example, if SIMD floating-point exceptions are unmasked for the computation shown in Figure 11-9, the processor would generate one SIMD floating-point exception for denormal operand conditions and a second SIMD floating-point exception for overflow and underflow (no inexact result exception would be generated because the multiplications of X0 and Y0 and of X1 and Y1 are exact).

11.5.3.3 Handling Combinations of Masked and Unmasked Exceptions

In situations where both masked and unmasked exceptions are detected, the processor will set exception flags for the masked and the unmasked exceptions. However, it will not return masked results until after the processor has detected and handled unmasked post-computation exceptions and returned from the exception handler (as in step 6 above) to finish executing the instruction.

11.5.4 Handling SIMD Floating-Point Exceptions in Software

Section 4.9.3, “Typical Actions of a Floating-Point Exception Handler,” shows actions that may be carried out by a SIMD floating-point exception handler. The SSE/SSE2/SSE3 state is saved with the FXSAVE instruction; see Section 11.6.5, “Saving and Restoring the SSE/SSE2 State.”

11.5.5 Interaction of SIMD and x87 FPU Floating-Point Exceptions

SIMD floating-point exceptions are generated independently from x87 FPU floating-point exceptions. SIMD floating-point exceptions do not cause assertion of the FERR# pin (independent of the value of CR0.NE[bit 5]). They ignore the assertion and deassertion of the IGNNE# pin.

If applications use Intel SSE/SSE2/SSE3 instructions along with x87 FPU instructions (in the same task or program), consider the following:

• SIMD floating-point exceptions are reported independently from the x87 FPU floating-point exceptions. SIMD and x87 FPU floating-point exceptions can be unmasked independently. Separate x87 FPU and SIMD floating-
point exception handlers must be provided if the same exception is unmasked for x87 FPU and for Intel SSE/SSE2/SSE3 operations.

• The rounding mode specified in the MXCSR register does not affect x87 FPU instructions. Likewise, the rounding mode specified in the x87 FPU control word does not affect the Intel SSE/SSE2/SSE3 instructions. To use the same rounding mode, the rounding control bits in the MXCSR register and in the x87 FPU control word must be set explicitly to the same value.

• The flush-to-zero mode set in the MXCSR register for Intel SSE/SSE2/SSE3 instructions has no counterpart in the x87 FPU. For compatibility with the x87 FPU, set the flush-to-zero bit to 0.

• The denormals-are-zeros mode set in the MXCSR register for Intel SSE/SSE2/SSE3 instructions has no counterpart in the x87 FPU. For compatibility with the x87 FPU, set the denormals-are-zeros bit to 0.

• An application that expects to detect x87 FPU exceptions that occur during the execution of x87 FPU instructions will not be notified if exceptions occurs during the execution of corresponding Intel SSE/SSE2/SSE3 instructions, unless the exception masks that are enabled in the x87 FPU control word have also been enabled in the MXCSR register and the application is capable of handling SIMD floating-point exceptions (#XM).
  — Masked exceptions that occur during an SSE/SSE2/SSE3 library call cannot be detected by unmasking the exceptions after the call (in an attempt to generate the fault based on the fact that an exception flag is set). A SIMD floating-point exception flag that is set when the corresponding exception is unmasked will not generate a fault; only the next occurrence of that unmasked exception will generate a fault.
  — An application which checks the x87 FPU status word to determine if any masked exception flags were set during an x87 FPU library call will also need to check the MXCSR register to detect a similar occurrence of a masked exception flag being set during an SSE/SSE2/SSE3 library call.

11.6 WRITING APPLICATIONS WITH INTEL® SSE AND SSE2

The following sections give some guidelines for writing application programs and operating-system code that uses Intel SSE and SSE2. Because Intel SSE and SSE2 share the same state and perform companion operations, these guidelines apply to both sets of extensions.

Chapter 14 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, discusses the interface to the processor for context switching as well as other operating system considerations when writing code that uses Intel SSE, SSE2, and SSE3.

11.6.1 General Guidelines for Using Intel® SSE and SSE2

The following guidelines describe how to take full advantage of the performance gains available with Intel SSE and SSE2:

• Ensure that the processor supports Intel SSE and SSE2.
• Ensure that your operating system supports Intel SSE and SSE2. (Operating system support for Intel SSE implies support for Intel SSE2, and vice versa.)
• Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
• Use the non-temporal store instructions offered with Intel SSE and SSE2.
• Employ the optimization and scheduling techniques described in the Intel® 64 and IA-32 Architectures Optimization Reference Manual; see Section 1.4, "Related Literature," for the order number for this manual.

11.6.2 Checking for Intel® SSE and SSE2 Support

Before an application attempts to use Intel SSE and/or Intel SSE2, it should check that they are present on the processor:

1. Intel SSE3 refers to ADDSUBPD, ADDSUBPS, HADDPD, HADDP, HSUBPD, HSUBP, and HSUBPS. The only other Intel SSE3 instruction that can raise floating-point exceptions is FISTTP; it can generate x87 FPU invalid operation and inexact result exceptions.
1. Check that the processor supports the CPUID instruction. Bit 21 of the EFLAGS register can be used to check processor’s support the CPUID instruction.

2. Check that the processor supports Intel SSE and/or SSE2 (true if CPUID.01H:EDX.SSE[bit 25] = 1 and/or CPUID.01H:EDX.SSE2[bit 26] = 1).

The operating system must provide system level support for handling SSE state, exceptions before an application can use Intel SSE and/or Intel SSE2; see Chapter 14 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

If the processor attempts to execute an unsupported Intel SSE or SSE2 instruction, the processor will generate an invalid-opcode exception (#UD). If an operating system did not provide adequate system level support for Intel SSE, executing an Intel SSE or SSE2 instructions can also generate #UD.

11.6.3 Checking for the DAZ Flag in the MXCSR Register

The denormals-are-zero flag in the MXCSR register is available in most of the Pentium 4 processors and in the Intel Xeon processor, with the exception of some early steppings. To check for the presence of the DAZ flag in the MXCSR register, do the following:

1. Establish a 512-byte FXSAVE area in memory.
2. Clear the FXSAVE area to all 0s.
3. Execute the FXSAVE instruction, using the address of the first byte of the cleared FXSAVE area as a source operand. See “FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for a description of the FXSAVE instruction and the layout of the FXSAVE image.
4. Check the value in the MXCSR_MASK field in the FXSAVE image (bytes 28 through 31).
   — If the value of the MXCSR_MASK field is 00000000H, the DAZ flag and denormals-are-zero mode are not supported.
   — If the value of the MXCSR_MASK field is non-zero and bit 6 is set, the DAZ flag and denormals-are-zero mode are supported.

If the DAZ flag is not supported, then it is a reserved bit and attempting to write a 1 to it will cause a general-protection exception (#GP). See Section 11.6.6, “Guidelines for Writing to the MXCSR Register,” for general guidelines for preventing general-protection exceptions when writing to the MXCSR register.

11.6.4 Initialization of Intel® SSE and SSE2

The SSE and SSE2 state is contained in the XMM and MXCSR registers. Upon a hardware reset of the processor, this state is initialized as follows (see Table 11-2):

- All SIMD floating-point exceptions are masked (bits 7 through 12 of the MXCSR register is set to 1).
- All SIMD floating-point exception flags are cleared (bits 0 through 5 of the MXCSR register is set to 0).
- The rounding control is set to round-nearest (bits 13 and 14 of the MXCSR register are set to 00B).
- The flush-to-zero mode is disabled (bit 15 of the MXCSR register is set to 0).
- The denormals-are-zeros mode is disabled (bit 6 of the MXCSR register is set to 0). If the denormals-are-zeros mode is not supported, this bit is reserved and will be set to 0 on initialization.
- Each of the XMM registers is cleared (set to all zeros).

<table>
<thead>
<tr>
<th>Registers</th>
<th>Power-Up or Reset</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM0 through XMM7</td>
<td>+0.0</td>
<td>Unchanged</td>
</tr>
<tr>
<td>MXCSR</td>
<td>1F80H</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

If the processor is reset by asserting the INIT# pin, the SSE and SSE2 state is not changed.
11.6.5 Saving and Restoring the SSE/SSE2 State

The FXSAVE instruction saves the x87 FPU, MMX, SSE, and SSE2 states (which includes the contents of eight XMM registers and the MXCSR registers) in a 512-byte block of memory. The FXRSTOR instruction restores the saved SSE and SSE2 state from memory. See the FXSAVE instruction in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for the layout of the 512-byte state block.

In addition to saving and restoring the SSE and SSE2 state, FXSAVE and FXRSTOR also save and restore the x87 FPU state (because MMX registers are aliased to the x87 FPU data registers this includes saving and restoring the MMX state). For greater code efficiency, it is suggested that FXSAVE and FXRSTOR be substituted for the FSAVE, FNSAVE, and FRSTOR instructions in the following situations:

- When a context switch is being made in a multitasking environment
- During calls and returns from interrupt and exception handlers

In situations where the code is switching between x87 FPU and MMX technology computations (without a context switch or a call to an interrupt or exception), the FSAVE/FNSAVE and FRSTOR instructions are more efficient than the FXSAVE and FXRSTOR instructions.

11.6.6 Guidelines for Writing to the MXCSR Register

The MXCSR has several reserved bits, and attempting to write a 1 to any of these bits will cause a general-protection exception (#GP) to be generated. To allow software to identify these reserved bits, the MXCSR_MASK value is provided. Software can determine this mask value as follows:

1. Establish a 512-byte FXSAVE area in memory.
2. Clear the FXSAVE area to all 0s.
3. Execute the FXSAVE instruction, using the address of the first byte of the cleared FXSAVE area as a source operand. See “FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for a description of FXSAVE and the layout of the FXSAVE image.
4. Check the value in the MXCSR_MASK field in the FXSAVE image (bytes 28 through 31).
   - If the value of the MXCSR_MASK field is 00000000H, then the MXCSR_MASK value is the default value of 0000FFBFH. Note that this value indicates that bit 6 of the MXCSR register is reserved; this setting indicates that the denormals-are-zero mode is not supported on the processor.
   - If the value of the MXCSR_MASK field is non-zero, the MXCSR_MASK value should be used as the MXCSR_MASK.

All bits set to 0 in the MXCSR_MASK value indicate reserved bits in the MXCSR register. Thus, if the MXCSR_MASK value is AND’d with a value to be written into the MXCSR register, the resulting value will be assured of having all its reserved bits set to 0, preventing the possibility of a general-protection exception being generated when the value is written to the MXCSR register.

For example, the default MXCSR_MASK value when 00000000H is returned in the FXSAVE image is 0000FFBFH. If software AND’s a value to be written to MXCSR register with 0000FFBFH, bit 6 of the result (the DAZ flag) will be ensured of being set to 0, which is the required setting to prevent general-protection exceptions on processors that do not support the denormals-are-zero mode.

To prevent general-protection exceptions, the MXCSR_MASK value should be AND’d with the value to be written into the MXCSR register in the following situations:

- Operating system routines that receive a parameter from an application program and then write that value to the MXCSR register (either with an FXRSTOR or LDMXCSR instruction)
- Any application program that writes to the MXCSR register and that needs to run robustly on several different IA-32 processors

Note that all bits in the MXCSR_MASK value that are set to 1 indicate features that are supported by the MXCSR register; they can be treated as feature flags for identifying processor capabilities.
11.6.7 Interaction of Intel® SSE and SSE2 Instructions with x87 FPU and MMX Instructions

The XMM registers and the x87 FPU and MMX registers represent separate execution environments, which has certain ramifications when executing Intel SSE, SSE2, MMX, and x87 FPU instructions in the same code module or when mixing code modules that contain these instructions:

- Those Intel SSE and SSE2 instructions that operate only on XMM registers (such as the packed and scalar floating-point instructions and the 128-bit SIMD integer instructions) in the same instruction stream with 64-bit SIMD integer or x87 FPU instructions without any restrictions. For example, an application can perform the majority of its floating-point computations in the XMM registers, using the packed and scalar floating-point instructions, and at the same time use the x87 FPU to perform trigonometric and other transcendental computations. Likewise, an application can perform packed 64-bit and 128-bit SIMD integer operations together without restrictions.

- Those Intel SSE and SSE2 instructions that operate on MMX registers (such as the CVTPS2PI, CVTTPS2PI, CVTP1PI, CVTTPD2PI, CVTPID2PI, CVTPI2PQ, MOVDQ2Q, MOVQ2DQ, PADDQ, and PSUBQ instructions) can also be executed in the same instruction stream as 64-bit SIMD integer or x87 FPU instructions, however, here they are subject to the restrictions on the simultaneous use of MMX technology and x87 FPU instructions, which include:
  - Transition from x87 FPU to MMX technology instructions or to Intel SSE or SSE2 instructions that operate on MMX registers should be preceded by saving the state of the x87 FPU.
  - Transition from MMX technology instructions or from Intel SSE or SSE2 instructions that operate on MMX registers to x87 FPU instructions should be preceded by execution of the EMMS instruction.

11.6.8 Compatibility of SIMD and x87 FPU Floating-Point Data Types

Intel SSE and SSE2 instructions operate on the same single precision and double precision floating-point data types that the x87 FPU operates on. However, when operating on these data types, Intel SSE and SSE2 operate on them in their native format (single precision or double precision), in contrast to the x87 FPU which extends them to double extended precision floating-point format to perform computations and then rounds the result back to a single precision or double precision format before writing results to memory. Because the x87 FPU operates on a higher precision format and then rounds the result to a lower precision format, it may return a slightly different result when performing the same operation on the same single precision or double precision floating-point values than is returned by Intel SSE and SSE2. The difference occurs only in the least-significant bits of the significand.

11.6.9 Mixing Packed and Scalar Floating-Point and 128-Bit SIMD Integer Instructions and Data

Intel SSE and SSE2 define typed operations on packed and scalar floating-point data types and on 128-bit SIMD integer data types, but IA-32 processors do not enforce this typing at the architectural level. They only enforce it at the microarchitectural level. Therefore, when a Pentium 4 or Intel Xeon processor loads a packed or scalar floating-point operand or a 128-bit packed integer operand from memory into an XMM register, it does not check that the actual data being loaded matches the data type specified in the instruction. Likewise, when the processor performs an arithmetic operation on the data in an XMM register, it does not check that the data being operated on matches the data type specified in the instruction.

As a general rule, because data typing of SIMD floating-point and integer data types is not enforced at the architectural level, it is the responsibility of the programmer, assembler, or compiler to ensure that code enforces data typing. Failure to enforce correct data typing can lead to computations that return unexpected results.

For example, in the following code sample, two packed single precision floating-point operands are moved from memory into XMM registers (using MOVAPS instructions); then a double precision packed add operation (using the ADDPD instruction) is performed on the operands:

```
movaps xmm0, [eax] ; EAX register contains pointer to packed single precision floating-point operand
movaps xmm1, [ebx]
addpd xmm0, xmm1
```
Pentium 4 and Intel Xeon processors execute these instructions without generating an invalid-operand exception (#UD) and will produce the expected results in register XMM0 (that is, the high and low 64-bits of each register will be treated as a double precision floating-point value and the processor will operate on them accordingly). Because the data types operated on and the data type expected by the ADDPD instruction were inconsistent, the instruction may result in a SIMD floating-point exception (such as numeric overflow [#O] or invalid operation [#I]) being generated, but the actual source of the problem (inconsistent data types) is not detected.

The ability to operate on an operand that contains a data type that is inconsistent with the typing of the instruction being executed, permits some valid operations to be performed. For example, the following instructions load a packed double precision floating-point operand from memory to register XMM0, and a mask to register XMM1; then they use XORPD to toggle the sign bits of the two packed values in register XMM0.

```
movapd xmm0, [eax] ; EAX register contains pointer to packed double precision floating-point operand
movaps xmm1, [ebx] ; EBX register contains pointer to packed double precision floating-point mask
xorpd xmm0, xmm1 ; XOR operation toggles sign bits using the mask in xmm1
```

In this example: XORPS or PXOR can be used in place of XORPD and yield the same correct result. However, because of the type mismatch between the operand data type and the instruction data type, a latency penalty will be incurred due to implementations of the instructions at the microarchitecture level.

Latency penalties can also be incurred by using move instructions of the wrong type. For example, MOVAPS and MOVAPD can both be used to move a packed single precision operand from memory to an XMM register. However, if MOVAPD is used, a latency penalty will be incurred when a correctly typed instruction attempts to use the data in the register.

Note that these latency penalties are not incurred when moving data from XMM registers to memory.

### 11.6.10 Interfacing with Intel® SSE and SSE2 Procedures and Functions

Intel SSE and SSE2 allow direct access to XMM registers. This means that all existing interface conventions between procedures and functions that apply to the use of the general-purpose registers (EAX, EBX, etc.) also apply to XMM register usage.

#### 11.6.10.1 Passing Parameters in XMM Registers

The state of XMM registers is preserved across procedure (or function) boundaries. Parameters can be passed from one procedure to another using XMM registers.

#### 11.6.10.2 Saving XMM Register State on a Procedure or Function Call

The state of XMM registers can be saved in two ways: using an FXSAVE instruction or a move instruction. FXSAVE saves the state of all XMM registers (along with the state of MXCSR and the x87 FPU registers). This instruction is typically used for major changes in the context of the execution environment, such as a task switch. FXRSTOR restores the XMM, MXCSR, and x87 FPU registers stored with FXSAVE.

In cases where only XMM registers must be saved, or where selected XMM registers need to be saved, move instructions (MOVAPS, MOVUPS, MOVSS, MOVAPD, MOVUPD, MOVSD, MOVDQA, and MOVDQU) can be used. These instructions can also be used to restore the contents of XMM registers. To avoid performance degradation when saving XMM registers to memory or when loading XMM registers from memory, be sure to use the appropriately typed move instructions.

The move instructions can also be used to save the contents of XMM registers on the stack. Here, the stack pointer (in the ESP register) can be used as the memory address to the next available byte in the stack. Note that the stack pointer is not automatically incremented when using a move instruction (as it is with PUSH).
A move-instruction procedure that saves the contents of an XMM register to the stack is responsible for decrementing the value in the ESP register by 16. Likewise, a move-instruction procedure that loads an XMM register from the stack needs also to increment the ESP register by 16. To avoid performance degradation when moving the contents of XMM registers, use the appropriately typed move instructions.

Use the LDMXCSR and STMXCSR instructions to save and restore, respectively, the contents of the MXCSR register on a procedure call and return.

**11.6.10.3 Caller-Save Recommendation for Procedure and Function Calls**

When making procedure (or function) calls from SSE or SSE2 code, a caller-save convention is recommended for saving the state of the calling procedure. Using this convention, any register whose content must survive intact across a procedure call must be stored in memory by the calling procedure prior to executing the call.

The primary reason for using the caller-save convention is to prevent performance degradation. XMM registers can contain packed or scalar double precision floating-point, packed single precision floating-point, and 128-bit packed integer data types. The called procedure has no way of knowing the data types in XMM registers following a call; so it is unlikely to use the correctly typed move instruction to store the contents of XMM registers in memory or to restore the contents of XMM registers from memory.

As described in Section 11.6.9, "Mixing Packed and Scalar Floating-Point and 128-Bit SIMD Integer Instructions and Data," executing a move instruction that does not match the type for the data being moved to/from XMM registers will be carried out correctly, but can lead to a greater instruction latency.

**11.6.11 Updating Existing MMX Technology Routines Using 128-Bit SIMD Integer Instructions**

Intel SSE2 extends all 64-bit MMX SIMD integer instructions to operate on 128-bit SIMD integers using XMM registers. The extended 128-bit SIMD integer instructions operate like the 64-bit SIMD integer instructions; this simplifies the porting of MMX technology applications. However, there are considerations:

- To take advantage of wider 128-bit SIMD integer instructions, MMX technology code must be recompiled to reference the XMM registers instead of MMX registers.
- Computation instructions that reference memory operands that are not aligned on 16-byte boundaries should be replaced with an unaligned 128-bit load (MOVUDQ instruction) followed by a version of the same computation operation that uses register instead of memory operands. Use of 128-bit packed integer computation instructions with memory operands that are not 16-byte aligned results in a general protection exception (#GP).
- Extension of the PSHUFW instruction (shuffle word across 64-bit integer operand) across a full 128-bit operand is emulated by a combination of the following instructions: PSHUFHW, PSHUFLW, and PSHUFD.
- Use of the 64-bit shift by bit instructions (PSRLQ, PSLLQ) can be extended to 128 bits in either of two ways:
  - Use of PSRLQ and PSLLQ, along with masking logic operations.
  - Rewriting the code sequence to use PSRDLQ and PSSLQ (shift double quadword operand by bytes)
- Loop counters need to be updated, since each 128-bit SIMD integer instruction operates on twice the amount of data as its 64-bit SIMD integer counterpart.

**11.6.12 Branching on Arithmetic Operations**

There are no condition codes in SSE or SSE2 states. A packed-data comparison instruction generates a mask which can then be transferred to an integer register. The following code sequence provides an example of how to perform a conditional branch, based on the result of an Intel SSE2 arithmetic operation.

```asm
 cmppd    XMM0, XMM1 ; generates a mask in XMM0
 movmskpd EAX, XMM0 ; moves a 2 bit mask to eax
 test     EAX, 0 ; compare with desired result
 jne     BRANCH TARGET
```
11.6.13 Cacheability Hint Instructions

Intel SSE and SSE2 cacheability control instructions enable the programmer to control prefetching, caching, loading, and storing of data. When correctly used, these instructions improve application performance.

To make efficient use of the processor's super-scalar microarchitecture, a program needs to provide a steady stream of data to the executing program to avoid stalling the processor. PREFETCH\*h instructions minimize the latency of data accesses in performance-critical sections of application code by allowing data to be fetched into the processor cache hierarchy in advance of actual usage.

PREFETCH\*h instructions do not change the user-visible semantics of a program, although they may affect performance. The operation of these instructions is implementation-dependent. Programmers may need to tune code for each IA-32 processor implementation. Excessive usage of PREFETCH\*h instructions may waste memory bandwidth and reduce performance. For more detailed information on the use of prefetch hints, refer to Chapter 7, "Optimizing Cache Usage," in the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

The non-temporal store instructions (MOVNTI, MOVNTPD, MOVNTPS, MOVNTDQ, MOVNTQ, MASKMOVQ, and MASKMOVQDQU) minimize cache pollution when writing non-temporal data to memory (see Section 10.4.6.1, "Cacheability Control Instructions," and Section 10.4.6.2, "Caching of Temporal vs. Non-Temporal Data"). They prevent non-temporal data from being written into processor caches on a store operation.

Besides reducing cache pollution, the use of weakly-ordered memory types can be important under certain data sharing relationships, such as a producer-consumer relationship. The use of weakly ordered memory can make the assembling of data more efficient; but care must be taken to ensure that the consumer obtains the data that the producer intended. Some common usage models that may be affected in this way by weakly-ordered stores are:

* Library functions that use weakly ordered memory to write results.
* Compiler-generated code that writes weakly-ordered results.
* Hand-crafted code.

The degree to which a consumer of data knows that the data is weakly ordered can vary for these cases. As a result, the SFENCE or MFENCE instruction should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume the data. SFENCE and MFENCE provide a performance-efficient way to ensure ordering by guaranteeing that every store instruction that precedes SFENCE/MFENCE in program order is globally visible before a store instruction that follows the fence.

11.6.14 Effect of Instruction Prefixes on Intel® SSE and SSE2 Instructions

Table 11-3 describes the effects of instruction prefixes on Intel SSE and SSE2 instructions. (Table 11-3 also applies to SIMD integer and SIMD floating-point instructions in Intel SSE3.) Unpredictable behavior can range from prefixes being treated as a reserved operation on one generation of IA-32 processors to generating an invalid opcode exception on another generation of processors.

See also "Instruction Prefixes" in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for complete description of instruction prefixes.

NOTE

Some Intel SSE, SSE2, and SSE3 instructions have two-byte opcodes that are either 2 bytes or 3 bytes in length. Two-byte opcodes that are 3 bytes in length consist of: a mandatory prefix (F2H, F3H, or 66H), 0FH, and an opcode byte. See Table 11-3.
<table>
<thead>
<tr>
<th>Prefix Type</th>
<th>Effect on the Intel® SSE, SSE2, and SSE3 Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Size Prefix (67H)</td>
<td>Affects instructions with a memory operand.</td>
</tr>
<tr>
<td></td>
<td>Reserved for instructions without a memory operand and may result in unpredictable</td>
</tr>
<tr>
<td></td>
<td>behavior.</td>
</tr>
<tr>
<td>Operand Size (66H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Segment Override</td>
<td>Affects instructions with a memory operand.</td>
</tr>
<tr>
<td>(2EH,36H,3EH,26H,64H,65H)</td>
<td>Reserved for instructions without a memory operand and may result in unpredictable</td>
</tr>
<tr>
<td></td>
<td>behavior.</td>
</tr>
<tr>
<td>Repeat Prefixes (F2H and F3H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
<tr>
<td>Lock Prefix (F0H)</td>
<td>Reserved; generates invalid opcode exception (#UD).</td>
</tr>
<tr>
<td>Branch Hint Prefixes (E2H and E3H)</td>
<td>Reserved and may result in unpredictable behavior.</td>
</tr>
</tbody>
</table>
This chapter describes the Intel SSE3, SSSE3, and Intel SSE4 instructions, and provides information to assist in writing application programs that use these extensions.

Intel AES-NI and PCLMLQDQ are instruction extensions targeted to accelerate high-speed block encryption and cryptographic processing. Section 12.13 covers these instructions and their relationship to the Advanced Encryption Standard (AES).

12.1 PROGRAMMING ENVIRONMENT AND DATA TYPES

The programming environment for using Intel SSE3, SSSE3, and Intel SSE4 is unchanged from those shown in Figure 3-1 and Figure 3-2. These extensions do not introduce new data types. XMM registers are used to operate on packed integer data, single precision floating-point data, or double precision floating-point data.

One Intel SSE3 instruction uses the x87 FPU for x87-style programming. There are two Intel SSE3 instructions that use the general registers for thread synchronization. The MXCSR register governs SIMD floating-point operations. Note, however, that the x87 FPU control word does not affect the Intel SSE3 instruction that is executed by the x87 FPU (FISTTP), other than by unmasking an invalid operand or inexact result exception.

Intel SSE4 instructions do not use MMX registers. The majority of Intel SSE4.2 and SSE4.1 instructions operate on XMM registers.

12.1.1 Intel® SSE3, SSSE3, and Intel® SSE4 in 64-Bit Mode and Compatibility Mode

In compatibility mode, Intel SSE3, SSSE3, and Intel SSE4 function like they do in protected mode. In 64-bit mode, eight additional XMM registers are accessible. Registers XMM8-XMM15 are accessed by using REX prefixes.

Memory operands are specified using the ModR/M, SIB encoding described in Section 3.7.5.

Some Intel SSE3, SSSE3, and Intel SSE4 instructions may be used to operate on general-purpose registers. Use the REX.W prefix to access 64-bit general-purpose registers. Note that if a REX prefix is used when it has no meaning, the prefix is ignored.

12.1.2 Compatibility of Intel® SSE3 and SSSE3 with MMX Technology, the x87 FPU Environment, and Intel® SSE and SSE2

Intel SSE3, SSSE3, and Intel SSE4 do not introduce any new state to the Intel 64 and IA-32 execution environments.

For SIMD and x87 programming, the FXSAVE and FXRSTOR instructions save and restore the architectural states of XMM, MXCSR, x87 FPU, and MMX registers. The MONITOR and MWAIT instructions use general purpose registers on input, they do not modify the content of those registers.

12.1.3 Horizontal and Asymmetric Processing

Many of the Intel SSE/SSE2/SSE3 and SSSE3 instructions accelerate SIMD data processing using a model referred to as vertical computation. Using this model, data flow is vertical between the data elements of the inputs and the output.

1. Although the presence of CRC32 support is enumerated by CPUID.O1h:ECX[SSE4.2] = 1, CRC32 operates on general purpose registers.
Figure 12-1 illustrates the asymmetric processing of the Intel SSE3 instruction ADDSUBPD. Figure 12-2 illustrates the horizontal data movement of the Intel SSE3 instruction HADDPD.

Figure 12-1. Asymmetric Processing in ADDSUBPD

Figure 12-2. Horizontal Data Movement in HADDPD

12.2 OVERVIEW OF INTEL® SSE3 INSTRUCTIONS

Intel SSE3 extensions include 13 instructions. See:
• Section 12.3, “Intel® SSE3 Instructions,” provides an introduction to individual Intel SSE3 instructions.
• The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, provides detailed information on individual instructions.

12.3 INTEL® SSE3 INSTRUCTIONS

Intel SSE3 instructions are grouped as follows:
• x87 FPU instruction:
  — One instruction that improves x87 FPU floating-point to integer conversion.
• SIMD integer instruction:
  — One instruction that provides a specialized 128-bit unaligned data load.
• SIMD floating-point instructions:
  — Three instructions that enhance LOAD/MOVE/DUPLICATE performance.
  — Two instructions that provide packed addition/subtraction.
  — Four instructions that provide horizontal addition/subtraction.
• Thread synchronization instructions:
  — Two instructions that improve synchronization between multi-threaded agents.

The instructions are discussed in more detail in the following paragraphs.

### 12.3.1 x87 FPU Instruction for Integer Conversion

The FISTTP instruction (x87 FPU Store Integer and Pop with Truncation) behaves like FISTP, but uses truncation regardless of what rounding mode is specified in the x87 FPU control word. The instruction converts the top of stack (ST0) to integer with rounding to and pops the stack.

The FISTTP instruction is available in three precisions: short integer (word or 16-bit), integer (double word or 32-bit), and long integer (64-bit). With FISTTP, applications no longer need to change the FCW when truncation is required.

### 12.3.2 SIMD Integer Instruction for Specialized 128-Bit Unaligned Data Load

The LDDQU instruction is a special 128-bit unaligned load designed to avoid cache line splits. If the address of a 16-byte load is on a 16-byte boundary, LDDQU loads the bytes requested. If the address of the load is not aligned on a 16-byte boundary, LDDQU loads a 32-byte block starting at the 16-byte aligned address immediately below the load request. It then extracts the requested 16 bytes.

The instruction provides significant performance improvement on 128-bit unaligned memory accesses at the cost of some usage model restrictions.

### 12.3.3 SIMD Floating-Point Instructions That Enhance LOAD/MOVE/DUPLICATE Performance

The MOVSHDUP instruction loads/moves 128-bits, duplicating the second and fourth 32-bit data elements.

- MOVSHDUPOperandA, OperandB
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (stored in OperandA): 3b, 3b, 1b, 1b

The MOVSLDUP instruction loads/moves 128-bits, duplicating the first and third 32-bit data elements.

- MOVSLDUPOperandA, OperandB
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (stored in OperandA): 2b, 2b, 0b, 0b

The MOVDDUP instruction loads/moves 64-bits; duplicating the 64 bits from the source.

- MOVDDUPOperandA, OperandB
  - OperandA (128 bits, two data elements): 1a, 0a
  - OperandB (64 bits, one data element): 0b
  - Result (stored in OperandA): 0b, 0b
12.3.4 **SIMD Floating-Point Instructions Provide Packed Addition/Subtraction**

The ADDSUBPS instruction has two 128-bit operands. The instruction performs single precision addition on the second and fourth pairs of 32-bit data elements within the operands; and single precision subtraction on the first and third pairs.

- **ADDSUBPS**OperandA, OperandB
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (stored in OperandA): 3a+3b, 2a-2b, 1a+1b, 0a-0b

The ADDSUBPD instruction has two 128-bit operands. The instruction performs double precision addition on the second pair of quadwords, and double precision subtraction on the first pair.

- **ADDSUBPD**OperandA, OperandB
  - OperandA (128 bits, two data elements): 1a, 0a
  - OperandB (128 bits, two data elements): 1b, 0b
  - Result (stored in OperandA): 1a+1b, 0a-0b

12.3.5 **SIMD Floating-Point Instructions Provide Horizontal Addition/Subtraction**

Most SIMD instructions operate vertically. This means that the result in position i is a function of the elements in position i of both operands. Horizontal addition/subtraction operates horizontally. This means that contiguous data elements in the same source operand are used to produce a result.

The HADDPS instruction performs a single precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.

- **HADDPS**OperandA, OperandB
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (Stored in OperandA): 3a+3b, 2a+2b, 1a+1b, 0a+0b

The HSUBPS instruction performs a single precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.

- **HSUBPS**OperandA, OperandB
  - OperandA (128 bits, four data elements): 3a, 2a, 1a, 0a
  - OperandB (128 bits, four data elements): 3b, 2b, 1b, 0b
  - Result (Stored in OperandA): 2a-3b, 1b-2b, 0a-1a

The HADDPD instruction performs a double precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.

- **HADDPD**OperandA, OperandB
  - OperandA (128 bits, two data elements): 1a, 0a
  - OperandB (128 bits, two data elements): 1b, 0b
  - Result (stored in OperandA): 1a+1b, 0a+0b
The HSUBPD instruction performs a double precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

- **HSUBPD** OperandA OperandB
  - OperandA (128 bits, two data elements): 1a, 0a
  - OperandB (128 bits, two data elements): 1b, 0b
  - Result (Stored in OperandA): 0b-1b, 0a-1a

### 12.3.6 Two Thread Synchronization Instructions

The MONITOR instruction sets up an address range that is used to monitor write-back-stores. MWAIT enables a logical processor to enter into an optimized state while waiting for a write-back-store to the address range set up by MONITOR. MONITOR and MWAIT require the use of general purpose registers for its input. The registers used by MONITOR and MWAIT must be initialized properly; register content is not modified by these instructions.

### 12.4 Writing Applications with Intel® SSE3

The following sections give guidelines for writing application programs and operating-system code that use Intel SSE3 instructions.

#### 12.4.1 Guidelines for Using Intel® SSE3

The following guidelines describe how to maximize the benefits of using Intel SSE3:

- Check that the processor supports Intel SSE3.
  - Applications may need to ensure that the target operating system supports Intel SSE3. (Operating system support for the Intel SSE implies sufficient support for Intel SSE2 and SSE3.)
- Ensure your operating system supports MONITOR and MWAIT.
- Employ the optimization and scheduling techniques described in the Intel® 64 and IA-32 Architectures Optimization Reference Manual (see Section 1.4, “Related Literature”).

#### 12.4.2 Checking for Intel® SSE3 Support

Before an application attempts to use the SIMD subset of Intel SSE3 instructions, the application should follow the steps illustrated in Section 11.6.2, “Checking for Intel® SSE and SSE2 Support.” Next, use the additional step provided below:

- Check that the processor supports the SIMD and x87 Intel SSE3 extensions (if CPUID.01H:ECX.SSE3[bit 0] = 1).

An operating system that provides application support for Intel SSE and SSE2 also provides sufficient application support for Intel SSE3. To use FISTTP, software only needs to check support for Intel SSE3.

In the initial implementation of MONITOR and MWAIT, these two instructions are available to ring 0 and conditionally available at ring level greater than 0. Before an application attempts to use the MONITOR and MWAIT instructions, the application should use the following steps:

1. Check that the processor supports MONITOR and MWAIT. If CPUID.01H:ECX.MONITOR[bit 3] = 1, MONITOR and MWAIT are available at ring 0.
2. Query the smallest and largest line size that MONITOR uses. Use CPUID.05H:EAX.smallest[bits 15:0];EBX.largest[bits15:0]. Values are returned in bytes in EAX and EBX.
3. Ensure the memory address range(s) that will be supplied to MONITOR meets memory type requirements. MONITOR and MWAIT are targeted for system software that supports efficient thread synchronization, see Chapter 14 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A for details.

12.4.3 Enable FTZ and DAZ for SIMD Floating-Point Computation

Enabling the FTZ and DAZ flags in the MXCSR register is likely to accelerate SIMD floating-point computation where strict compliance to the IEEE standard 754-1985 is not required. The FTZ flag is available to Intel 64 and IA-32 processors that support Intel SSE; DAZ is available to Intel 64 processors and to most IA-32 processors that support Intel SSE, SSE2, and SSE3.

Software can detect the presence of DAZ, modify the MXCSR register, and save and restore state information by following the techniques discussed in Section 11.6.3 through Section 11.6.6.

12.4.4 Programming Intel® SSE3 with Intel® SSE and SSE2

SIMD instructions in Intel SSE3 are intended to complement the use of Intel SSE and SSE2 in programming SIMD applications. Application software that intends to use Intel SSE3 instructions should also check for the availability of Intel SSE and SSE2 instructions.

The FISTTP instruction in Intel SSE3 is intended to accelerate x87 style programming where performance is limited by frequent floating-point conversion to integers; this happens when the x87 FPU control word is modified frequently. Use of the FISTTP instruction can eliminate the need to access the x87 FPU control word.

12.5 Overview of SSSE3 Instructions

SSSE3 provides 32 instructions to accelerate a variety of multimedia and signal processing applications employing SIMD integer data. See:

- Section 12.6, “SSSE3 Instructions,” provides an introduction to individual SSSE3 instructions.
- The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, provides detailed information on individual instructions.

12.6 SSSE3 Instructions

SSSE3 instructions include:

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate the absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

The operands of these instructions are packed integers of byte, word, or double word sizes. The operands are stored as 64 or 128 bit data in MMX registers, XMM registers, or memory.

The instructions are discussed in more detail in the following paragraphs.
12.6.1 Horizontal Addition/Subtraction

In analogy to the packed, floating-point horizontal add and subtract instructions in Intel SSE3, SSSE3 offers similar capabilities on packed integer data. Data elements of signed words, doublewords are supported. Saturated version for horizontal add and subtract on signed words are also supported. The horizontal data movement of PHADD is shown in Figure 12-3.

![Figure 12-3. Horizontal Data Movement in PHADDD](image)

There are six horizontal add instructions (represented by three mnemonics); three operate on 128-bit operands and three operate on 64-bit operands. The width of each data element is either 16 bits or 32 bits. The mnemonics are listed below.

- **PHADDW** adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.
- **PHADDSW** adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
- **PHADDD** adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.

There are six horizontal subtract instructions (represented by three mnemonics); three operate on 128-bit operands and three operate on 64-bit operands. The width of each data element is either 16 bits or 32 bits. These are listed below.

- **PHSUBW** performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed 16-bit results are packed and written to the destination operand.
- **PHSUBSW** performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.
- **PHSUBD** performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

12.6.2 Packed Absolute Values

There are six packed-absolute-value instructions (represented by three mnemonics). Three operate on 128-bit operands and three operate on 64-bit operands. The widths of data elements are 8 bits, 16 bits or 32 bits. The absolute value of each data element of the source operand is stored as an UNSIGNED result in the destination operand.

- **PABSB** computes the absolute value of each signed byte data element.
• PABSW computes the absolute value of each signed 16-bit data element.
• PABSD computes the absolute value of each signed 32-bit data element.

12.6.3 Multiply and Add Packed Signed and Unsigned Bytes

There are two multiply-and-add-packed-signed-unsigned-byte instructions (represented by one mnemonic). One operates on 128-bit operands and the other operates on 64-bit operands. Multiplications are performed on each vertical pair of data elements. The data elements in the source operand are signed byte values, the input data elements of the destination operand are unsigned byte values.
• PMADDUBSW multiplies each unsigned byte value with the corresponding signed byte value to produce an intermediate, 16-bit signed integer. Each adjacent pair of 16-bit signed values are added horizontally. The signed, saturated 16-bit results are packed to the destination operand.

12.6.4 Packed Multiply High with Round and Scale

There are two packed-multiply-high-with-round-and-scale instructions (represented by one mnemonic). One operates on 128-bit operands and the other operates on 64-bit operands.
• PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand with the corresponding signed 16-bit integer of the source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

12.6.5 Packed Shuffle Bytes

There are two packed-shuffle-bytes instructions (represented by one mnemonic). One operates on 128-bit operands and the other operates on 64-bit operands. The shuffle operations are performed bytewise on the destination operand using the source operand as a control mask.
• PSHUFB permutes each byte in place, according to a shuffle control mask. The least significant three or four bits of each shuffle control byte of the control mask form the shuffle index. The shuffle mask is unaffected. If the most significant bit (bit 7) of a shuffle control byte is set, the constant zero is written in the result byte.

12.6.6 Packed Sign

There are six packed-sign instructions (represented by three mnemonics). Three operate on 128-bit operands and three operate on 64-bit operands. The widths of each data element for these instructions are 8 bit, 16 bit or 32 bit signed integers.
• PSIGNB/W/D negates each signed integer element of the destination operand if the sign of the corresponding data element in the source operand is less than zero.

12.6.7 Packed Align Right

There are two packed-align-right instructions (represented by one mnemonic). One operates on 128-bit operands and the other operates on 64-bit operands. These instructions concatenate the destination and source operand into a composite, and extract the result from the composite according to an immediate constant.
• PALIGNR’s source operand is appended after the destination operand forming an intermediate value of twice the width of an operand. The result is extracted from the intermediate value into the destination operand by selecting the 128-bit or 64-bit value that are right-aligned to the byte offset specified by the immediate value.
12.7  WRITING APPLICATIONS WITH SSSE3 EXTENSIONS

The following sections give guidelines for writing application programs and operating-system code that use SSSE3 instructions.

12.7.1  Guidelines for Using SSSE3

The following guidelines describe how to maximize the benefits of using SSSE3:

•  Check that the processor supports SSSE3.
•  Ensure that your operating system supports SSSE3 and Intel SSE, SSE2, and SSE3. (Operating system support for Intel SSE implies sufficient support for SSSE3 and Intel SSE2 and SSE3.)
•  Employ the optimization and scheduling techniques described in the Intel® 64 and IA-32 Architectures Optimization Reference Manual (see Section 1.4, "Related Literature").

12.7.2  Checking for SSSE3 Support

Before an application attempts to use SSSE3, the application should follow the steps illustrated in Section 11.6.2, "Checking for Intel® SSE and SSE2 Support." Next, use the additional step provided below:

•  Check that the processor supports SSSE3 (if CPUID.01H:ECX.SSSE3[bit 9] = 1).

12.8  INTEL® SSE3, SSSE3, AND INTEL® SSE4 EXCEPTIONS

Intel SSE3, SSSE3, and Intel SSE4 instructions can generate the same type of memory-access and non-numeric exceptions as other Intel 64 or IA-32 instructions. Existing exception handlers generally handle these exceptions without code modification.

FISTTP can generate floating-point exceptions. Some Intel SSE3 instructions can also generate SIMD floating-point exceptions.

Intel SSE3 additions and changes are noted in the following sections. See also: Section 11.5, "Intel® SSE, SSE2, and SSE3 Exceptions".

12.8.1  Device Not Available (DNA) Exceptions

Intel SSE3, SSSE3, and Intel SSE4 will cause a DNA Exception (#NM) if the processor attempts to execute an Intel SSE3 instruction while CR0.TS[bit 3] = 1. If CPUID.01H:ECX.SSSE3[bit 0] = 0, execution of an Intel SSE3 instruction will cause an invalid opcode fault regardless of the state of CR0.TS[bit 3].

Similarly, an attempt to execute an SSSE3 instruction on a processor that reports CPUID.01H:ECX.SSSE3[bit 9] = 0 will cause an invalid opcode fault regardless of the state of CR0.TS[bit 3]. An attempt to execute an Intel SSE4.1 instruction on a processor that reports CPUID.01H:ECX.SSE4_1[bit 19] = 0 will cause an invalid opcode fault regardless of the state of CR0.TS[bit 3].

An attempt to execute PCMPGTQ or any one of the four string processing instructions in Intel SSE4.2 on a processor that reports CPUID.01H:ECX.SSE4_2[bit 20] = 0 will cause an invalid opcode fault regardless of the state of CR0.TS[bit 3]. CRC32 and POPCNT do not cause #NM.

12.8.2  Numeric Error Flag and IGNNE#

Most Intel SSE3 instructions ignore CR0.NE[bit 5] (treats it as if it were always set) and the IGNNE# pin. With one exception, all use the exception 19 (#XM) software exception for error reporting. The exception is FISTTP; it behaves like other x87-FP instructions.

SSSE3 instructions ignore CR0.NE[bit 5] (treats it as if it were always set) and the IGNNE# pin.
12.8.3 Emulation
CR0.EM is used by some software to emulate x87 floating-point instructions. CR0.EM[bit 2] cannot be used for emulation of SSSE3 and Intel SSE, SSE2, SSE3, and SSE4. If an Intel SSE3, SSSE3, or Intel SSE4 instruction execute with CR0.EM[bit 2] set, an invalid opcode exception (INT 6) is generated instead of a device not available exception (INT 7).

12.8.4 IEEE 754 Compliance of Intel® SSE4.1 Floating-Point Instructions
The six Intel SSE4.1 instructions that perform floating-point arithmetic are:
- DPPS
- DPPD
- ROUNDPS
- ROUNDPD
- ROUNDSS
- ROUNDSD

Dot Product operations are not specified in IEEE-754. When neither FTZ nor DAZ are enabled, the dot product instructions resemble sequences of IEEE-754 multiplies and adds (with rounding at each stage), except that the treatment of input NaN’s is implementation specific (there will be at least one NaN in the output). The input select fields (bits imm8[4:7]) force input elements to +0.0f prior to the first multiply and will suppress input exceptions that would otherwise have been generated.

As a convenience to the exception handler, any exceptions signaled from DPPS or DPPD leave the destination unmodified.

Round operations signal invalid and precision only.

<table>
<thead>
<tr>
<th></th>
<th>DPPS</th>
<th>DPPD</th>
<th>ROUNDPS</th>
<th>ROUNDPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Underflow</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>X</td>
<td>X</td>
<td>X (1)</td>
<td>X (1)</td>
</tr>
<tr>
<td>Inexact Precision</td>
<td>X</td>
<td>X</td>
<td>X (2)</td>
<td>X (2)</td>
</tr>
<tr>
<td>Denormal</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE:
1. Invalid is signaled only if Src = SNaN.
2. Precision is ignored (regardless of the MXCSR precision mask) if if imm8[3] = ‘1’.

The other Intel SSE4.1 instructions with floating-point arguments (BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, INSERTPS, EXTRACTPS) do not signal any SIMD numeric exceptions.

12.9 INTEL® SSE4 OVERVIEW
Intel SSE4 comprises two sets of extensions: Intel SSE4.1 and SSE4.2. Intel SSE4.1 is targeted to improve the performance of media, imaging, and 3D workloads. Intel SSE4.1 adds instructions that improve compiler vectoriza-
tion and significantly increase support for packed dword computation. The technology also provides a hint that can improve memory throughput when reading from uncacheable WC memory type.

The 47 Intel SSE4.1 instructions include:
- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction performs a load with a streaming hint.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

The Intel SSE4.2 instructions operating on XMM registers improve performance in the following areas:
- String and text processing that can take advantage of single-instruction multiple-data programming techniques.
- A SIMD integer instruction that enhances the capability of the 128-bit integer SIMD capability in Intel SSE4.1.

## 12.10 Intel® SSE4.1 Instruction Set

### 12.10.1 Dword Multiply Instructions

Intel SSE4.1 adds two dword multiply instructions that aid vectorization. They allow four simultaneous 32-bit by 32-bit multiplies. PMULLD returns a low 32-bits of the result and PMULDQ returns a 64-bit signed result. These represent the most common integer multiply operation. See Table 12-2.

<table>
<thead>
<tr>
<th>Result</th>
<th>32-Bit Integer Operation</th>
<th>Signed x Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low 32-bit</td>
<td>(not available)</td>
<td>PMULLD</td>
</tr>
<tr>
<td>High 32-bit</td>
<td>(not available)</td>
<td>(not available)</td>
</tr>
<tr>
<td>64-bit</td>
<td>PMULUDQ*</td>
<td>PMULDQ</td>
</tr>
</tbody>
</table>

* Available prior to Intel SSE4.1.

### 12.10.2 Floating-Point Dot Product Instructions

Intel SSE4.1 adds two instructions for double precision (for up to 2 elements; DPPD) and single precision dot products (for up to 4 elements; DPPS).

These dot-product instructions include source select and destination broadcast which generally improves the flexibility. For example, a single DPPS instruction can be used for a 2, 3, or 4 element dot product.
12.10.3 Streaming Load Hint Instruction

Historically, CPU read accesses of WC memory type regions have significantly lower throughput than accesses to cacheable memory.

The streaming load instruction in SSE4.1, MOVNTDQA, provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region of WC memory type (a streaming line) to be fetched and held in a small set of temporary buffers ("streaming load buffers"). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be satisfied from the streaming load buffer and can improve throughput.

Programmers are advised to use the following practices to improve the efficiency of MOVNTDQA streaming loads from WC memory:

• Streaming loads must be 16-byte aligned.

• Temporally group streaming loads of the same streaming cache line for effective use of the small number of streaming load buffers. If loads to the same streaming line are excessively spaced apart, it may cause the streaming line to be re-fetched from memory.

• Temporally group streaming loads from at most a few streaming lines together. The number of streaming load buffers is small; grouping a modest number of streams will avoid running out of streaming load buffers and the resultant re-fetching of streaming lines from memory.

• Avoid writing to a streaming line until all 16-byte-aligned reads from the streaming line have occurred. Reading a 16-byte item from a streaming line that has been written, may cause the streaming line to be re-fetched.

• Avoid reading a given 16-byte item within a streaming line more than once; repeated loads of a particular 16-byte item are likely to cause the streaming line to be re-fetched.

• The streaming load buffers, reflecting the WC memory type characteristics, are not required to be snooped by operations from other agents. Software should not rely upon such coherency actions to provide any data coherency with respect to other logical processors or bus agents. Rather, software must ensure the consistency of WC memory accesses between producers and consumers.

• Streaming loads may be weakly ordered and may appear to software to execute out of order with respect to other memory operations. Software must explicitly use MFENCE if it needs to preserve order among streaming loads or between streaming loads and other memory operations.

• Streaming loads must not be used to reference memory addresses that are mapped to I/O devices having side effects or when reads to these devices are destructive. This is because MOVNTDQA is speculative in nature.

Example 12-1 provides a sketch of the basic assembly sequences that illustrate the principles of using MOVNTDQA in a situation with a producer-consumer accessing a WC memory region.
Example 12-1. Sketch of MOVNTDQA Usage of a Consumer and a PCI Producer

// P0: producer is a PCI device writing into the WC space
# the PCI device updates status through a UC flag, "u_dev_status".
# the protocol for "u_dev_status": 0: produce; 1: consume; 2: all done

        mov eax, $0
        mov [u_dev_status], eax
producerStart:
        mov eax, [u_dev_status]  # poll status flag to see if consumer is requestion data
        cmp eax, $0              #
        jne done                # I no longer need to produce
        commence PCI writes to WC region..

        mov eax, $1  # producer ready to notify the consumer via status flag
        mov  [u_dev_status], eax
# now wait for consumer to signal its status
spinloop:
        cmp [u_dev_status], $1      # did I get a signal from the consumer ?
        jne producerStart          # yes I did
        jmp spinloop               # check again
done:
// producer is finished at this point

// P1: consumer check PCI status flag to consume WC data
        mov eax, $0  # request to the producer
        mov [u_dev_status], eax
consumerStart:
        mov; eax, [u_dev_status]  # reads the value of the PCI status
        cmp eax, $1              # has producer written
        jne consumerStart        # tight loop; make it more efficient with pause, etc.
        mfence                   # producer finished device writes to WC, ensure WC region is coherent
ntread:
        movntdqa xmm0, [addr]
        movntdqa xmm1, [addr + 16]
        movntdqa xmm2, [addr + 32]
        movntdqa xmm3, [addr + 48]
...  # do any more NT reads as needed
        mfence  # ensure PCI device reads the correct value of [u_dev_status]
# now decide whether we are done or we need the producer to produce more data
# if we are done write a 2 into the variable, otherwise write a 0 into the variable
        mov eax, $0/$2            # end or continue producing
        mov [u_dev_status], eax
# if I want to consume again I will jump back to consumerStart after storing a 0 into eax
# otherwise I am done
12.10.4 Packed Blending Instructions

Intel SSE4.1 adds 6 instructions used for blending (BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, PBLENDVB, PBLENDW).

Blending conditionally copies a data element in a source operand to the same element in the destination. Intel SSE4.1 instructions improve blending operations for most field sizes. A single new Intel SSE4.1 instruction can generally replace a sequence of 2 to 4 operations using previous architectures.

The variable blend instructions (BLENDVPS, BLENDVPD, PBLENDW) introduce the use of control bits stored in an implicit XMM register (XMM0). The most significant bit in each field (the sign bit, for 2’s complement integer or floating-point) is used as a selector. See Table 12-3.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Packed Double FP</th>
<th>Packed Single FP</th>
<th>Packed QWord</th>
<th>Packed DWord</th>
<th>Packed Word</th>
<th>Packed Byte</th>
<th>Blend Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLENDPS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imm8</td>
</tr>
<tr>
<td>BLENDPD</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imm8</td>
</tr>
<tr>
<td>BLENDVPS</td>
<td>X</td>
<td></td>
<td></td>
<td>X(1)</td>
<td></td>
<td></td>
<td>XMM0</td>
</tr>
<tr>
<td>BLENDVPD</td>
<td></td>
<td></td>
<td>X(1)</td>
<td></td>
<td>X(1)</td>
<td></td>
<td>XMM0</td>
</tr>
<tr>
<td>PBLENDVB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XMM0</td>
</tr>
<tr>
<td>PBLENDW</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**NOTE:**
1. Use of floating-point SIMD instructions on integer data types may incur performance penalties.
2. Byte variable blend can be used for larger sized fields by reformating (or shuffling) the blend control.

12.10.5 Packed Integer MIN/MAX Instructions

Intel SSE4.1 adds 8 packed integer MIN and MAX instructions: PMINUW, PMINUD, PMINSB, PMINSD; PMAXUW, PMAXUD, PMAXSB, and PMAXSD.

Four 32-bit integer packed MIN and MAX instructions operate on unsigned and signed dwords. Two instructions operate on signed bytes. Two instructions operate on unsigned words. See Table 12-4.

<table>
<thead>
<tr>
<th>Integer Format</th>
<th>Integer Width</th>
<th>Byte</th>
<th>Word</th>
<th>DWord</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PMINUB*</td>
<td>PMINUW</td>
<td>PMINUD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PMAXUB*</td>
<td>PMAXUW</td>
<td>PMAXUD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PMINSB</td>
<td>PMINSW*</td>
<td>PMINSD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PMAXSB</td>
<td>PMAXSW*</td>
<td>PMAXSD</td>
</tr>
</tbody>
</table>

**NOTE:**
* Available prior to Intel SSE4.1.

12.10.6 Floating-Point Round Instructions with Selectable Rounding Mode

High level languages and libraries often expose rounding operations having a variety of numeric rounding and exception behaviors. Using Intel SSE, SSE2, and SSE3 instructions to mitigate the rounding-mode-related problem is sometimes not straight forward.

Intel SSE4.1 introduces four rounding instructions (ROUNDPS, ROUNDPD, ROUNDSS, and ROUNDSD) that cover scalar and packed single- and double precision floating-point operands. The rounding mode can be selected using an immediate from one of the IEEE-754 modes (Nearest, -Inf, +Inf, and Truncate) without changing the current...
rounding mode; or the instruction can be forced to use the current rounding mode. Another bit in the immediate is used to suppress inexact precision exceptions.

Rounding instructions in Intel SSE4.1 generally permit single-instruction solutions to C99 functions ceil(), floor(), trunc(), rint(), nearbyint(). These instructions simplify the implementations of half-way-away-from-zero rounding modes as used by C99 round() and F90’s nint().

12.10.7 Insertion and Extractions from XMM Registers

Intel SSE4.1 adds 7 instructions (corresponding to 9 assembly instruction mnemonics) that simplify data insertion and extraction between general-purpose register (GPR) and XMM registers: EXTRACTPS, INSERTPS, PINSRB, PINSRD, PINSRQ, PEXTRB, PEXTRW, PEXTRD, and PEXTRQ. When accessing memory, no alignment is required for any of these instructions (unless alignment checking is enabled).

EXTRACTPS extracts a single precision floating-point value from any dword offset in an XMM register and stores the result to memory or a general-purpose register. INSERTPS inserts a single floating-point value from either a 32-bit memory location or from specified element in an XMM register to a selected element in the destination XMM register. In addition, INSERTPS allows the insertion of +0.0f into any destination elements using a mask.

PINSRB, PINSRD, and PINSRQ insert byte, dword, or qword integer values from a register or memory into an XMM register. Insertion of integer word values were already supported by Intel SSE2 (PINSRW).

PEXTRB, PEXTRW, PEXTRD, and PEXTRQ extract byte, word, dword, and qword from an XMM register and insert the values into a general-purpose register or memory.

12.10.8 Packed Integer Format Conversions

A common type of operation on packed integers is the conversion by zero- or sign-extension of packed integers into wider data types. Intel SSE4.1 adds 12 instructions that convert from a smaller packed integer type to a larger integer type: PMOVSXBW, PMOVZXBW, PMOVXSXB, PMOVZXBD, PMOVSXWD, PMOVZWXQ, PMOVZQWQ, PMOVZWXQ, PMOVZQDXQ, and PMOVZQDXQ.

The source operand is from either an XMM register or memory; the destination is an XMM register. See Table 12-5. When accessing memory, no alignment is required for any of the instructions unless alignment checking is enabled. In which case, all conversions must be aligned to the width of the memory reference. The number of elements converted (and width of memory reference) is illustrated in Table 12-6. The alignment requirement is shown in parenthesis.

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Byte</th>
<th>Word</th>
<th>Dword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signed Word</td>
<td>PMOVXSXBW</td>
<td>PMOVZXBW</td>
<td></td>
</tr>
<tr>
<td>Unsigned Word</td>
<td>PMOVXSXBW</td>
<td>PMOVZXBW</td>
<td></td>
</tr>
<tr>
<td>Signed Dword</td>
<td>PMOVXSXB</td>
<td>PMOVZXB</td>
<td>PMOVZXWD</td>
</tr>
<tr>
<td>Unsigned Dword</td>
<td>PMOVXSXB</td>
<td>PMOVZXB</td>
<td>PMOVZXWD</td>
</tr>
<tr>
<td>Signed Qword</td>
<td>PMOVXSXBQ</td>
<td>PMOVZXBQ</td>
<td>PMOVZXWD</td>
</tr>
<tr>
<td>Unsigned Qword</td>
<td>PMOVXSXBQ</td>
<td>PMOVZXBQ</td>
<td>PMOVZXWD</td>
</tr>
</tbody>
</table>

Table 12-5. New SIMD Integer Conversions Supported by Intel® SSE4.1
12.10.9 Improved Sums of Absolute Differences (SAD) for 4-Byte Blocks

Intel SSE4.1 adds an instruction (MPSADBW) that performs eight 4-byte wide SAD operations per instruction to produce eight results. Compared to PSADBW, MPSADBW operates on smaller chunks (4-byte instead of 8-byte chunks); this makes the instruction better suited to video coding standards such as VC.1 and H.264. MPSADBW performs four times the number of absolute difference operations than that of PSADBW (per instruction). This can improve performance for dense motion searches.

MPSADBW uses a 4-byte wide field from a source operand; the offset of the 4-byte field within the 128-bit source operand is specified by two immediate control bits. MPSADBW produces eight 16-bit SAD results. Each 16-bit SAD result is formed from overlapping pairs of 4 bytes in the destination with the 4-byte field from the source operand. MPSADBW uses eleven consecutive bytes in the destination operand, its offset is specified by a control bit in the immediate byte (i.e., the offset can be from byte 0 or from byte 4). Figure 12-4 illustrates the operation of MPSADBW. MPSADBW can simplify coding of dense motion estimation by providing source and destination offset control, higher throughput of SAD operations, and the smaller chunk size.

<table>
<thead>
<tr>
<th>Destination Type</th>
<th>Source Type</th>
<th>Word</th>
<th>Dword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>Word</td>
<td>8 (64 bits)</td>
<td></td>
</tr>
<tr>
<td>Dword</td>
<td>Dword</td>
<td>4 (32 bits)</td>
<td>4 (64 bits)</td>
</tr>
<tr>
<td>Qword</td>
<td>Qword</td>
<td>2 (16 bits)</td>
<td>2 (32 bits)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Destination Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Word</td>
</tr>
<tr>
<td>Dword</td>
<td></td>
</tr>
<tr>
<td>Qword</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12-4. MPSADBW Operation

12.10.10 Horizontal Search

Intel SSE4.1 adds a search instruction (PHMINPOSUW) that finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words. The resulting value and location (offset within the source) are packed into the low dword of the destination XMM register.

Rapid search is often a significant component of motion estimation. MPSADBW and PHMINPOSUW can be used together to improve video encode.
12.10.11 Packed Test

The packed test instruction PTEST is similar to a 128-bit equivalent to the legacy instruction TEST. With PTEST, the source argument is typically used like a bit mask.

PTEST performs a logical AND between the destination with this mask and sets the ZF flag if the result is zero. The CF flag (zero for TEST) is set if the inverted mask AND’d with the destination is all zero. Because the destination is not modified, PTEST simplifies branching operations (such as branching on signs of packed floating-point numbers, or branching on zero fields).

12.10.12 Packed Qword Equality Comparisons

Intel SSE4.1 adds a 128-bit packed qword equality test. The new instruction (PCMPEQQ) is identical to PCMPEQD, but has qword granularity.

12.10.13 Dword Packing With Unsigned Saturation

Intel SSE4.1 adds a new instruction PACKUSDW to complete the set of small integer pack instructions in the family of SIMD instruction extensions. PACKUSDW packs dword to word with unsigned saturation. See Table 12-7 for the complete set of packing instructions for small integers.

<table>
<thead>
<tr>
<th>Table 12-7. Enhanced SIMD Pack Support by Intel® SSE4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table" /></td>
</tr>
</tbody>
</table>

12.11 INTEL® SSE4.2 INSTRUCTION SET

Five of the seven Intel SSE4.2 instructions can use an XMM register as a source or destination. These include four text/string processing instructions and one packed quadword compare SIMD instruction. Programming these five Intel SSE4.2 instructions is similar to programming 128-bit Integer SIMD in Intel SSE2 or SSSE3. Intel SSE4.2 does not provide any 64-bit integer SIMD instructions.

12.11.1 String and Text Processing Instructions

String and text processing instructions in Intel SSE4.2 allocates four opcodes to provide a rich set of string and text processing capabilities that traditionally required many more opcodes. These four instructions use XMM registers to process string or text elements of up to 128-bits (16 bytes or 8 words). Each instruction uses an immediate byte to support a rich set of programmable controls. A string-processing Intel SSE4.2 instruction returns the result of processing each pair of string elements using either an index or a mask.

The capabilities of the string/text processing instructions include:

- Handling string/text fragments consisting of bytes or words, either signed or unsigned.
- Support for partial string or fragments less than 16 bytes in length, using either explicit length or implicit null-termination.
- Four types of string compare operations on word/byte elements.
- Up to 256 compare operations performed in a single instruction on all string/text element pairs.
- Built-in aggregation of intermediate results from comparisons.
• Programmable control of processing on intermediate results.
• Programmable control of output formats in terms of an index or mask.
• Bi-directional support for the index format.
• Support for two mask formats: bit or natural element width.
• Not requiring 16-byte alignment for memory operand.

The four Intel SSE4.2 instructions that process text/string fragments are:
• PCMPESTRI — Packed compare explicit-length strings, return index in ECX/RCX.
• PCMPESTRM — Packed compare explicit-length strings, return mask in XMM0.
• PCMPISTRI — Packed compare implicit-length strings, return index in ECX/RCX.
• PCMPISTRM — Packed compare implicit-length strings, return mask in XMM0.

All four of these instructions require the use of an immediate byte to control operation. The two source operands can be XMM registers or a combination of XMM register and memory address. The immediate byte provides programmable control with the following attributes:
• Input data format.
• Compare operation mode.
• Intermediate result processing.
• Output selection.

Depending on the output format associated with the instruction, the text/string processing instructions implicitly uses either a general-purpose register (ECX/RCX) or an XMM register (XMM0) to return the final result.

Two of the four text-string processing instructions specify string length explicitly. They use two general-purpose registers (EDX, EAX) to specify the number of valid data elements (either word or byte) in the source operands. The other two instructions specify valid string elements using null termination. A data element is considered valid only if it has a lower index than the least significant null data element.

12.11.1.1 Memory Operand Alignment
The text and string processing instructions in Intel SSE4.2 do not perform alignment checking on memory operands. This is different from most other 128-bit SIMD instructions accessing the XMM registers. The absence of an alignment check for these four instructions does not imply any modification to the existing definitions of other instructions.

12.11.2 Packed Comparison SIMD Integer Instruction
Intel SSE4.2 also provides a 128-bit integer SIMD instruction PCMPGTQ that performs logical compare of greater-than on packed integer quadwords.

12.12 WRITING APPLICATIONS WITH INTEL® SSE4 EXTENSIONS

12.12.1 Guidelines for Using Intel® SSE4 Extensions
The following guidelines describe how to maximize the benefits of using Intel SSE4 extensions:
• Check that the processor supports Intel SSE4 extensions.
• Ensure that the operating system supports SSSE3 and Intel SSE, SSE2, and SSE3. (Operating system support for Intel SSE implies sufficient support for SSSE3 and Intel SSE2, SSE3, and SSE4.)
• Employ the optimization and scheduling techniques described in the Intel® 64 and IA-32 Architectures Optimization Reference Manual (see Section 1.4, “Related Literature”).
12.12.2 Checking for Intel® SSE4.1 Support

Before an application attempts to use Intel SSE4.1 instructions, the application should follow the steps illustrated in Section 11.6.2, “Checking for Intel® SSE and SSE2 Support.” Next, use the additional step provided below:

Check that the processor supports Intel SSE4.1 (if CPUID.01H:ECX.SSE4_1[bit 19] = 1), Intel SSE3 (if CPUID.01H:ECX.SSE3[bit 0] = 1), and SSSE3 (if CPUID.01H:ECX.SSSE3[bit 9] = 1).

12.12.3 Checking for Intel® SSE4.2 Support

Before an application attempts to use the following Intel SSE4.2 instructions: PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM, PCMPGTQ; the application should follow the steps illustrated in Section 11.6.2, “Checking for Intel® SSE and SSE2 Support.” Next, use the additional steps provided below:

- Check that the processor supports Intel SSE4.2 (if CPUID.01H:ECX.SSE4_2[bit 20] = 1), Intel SSE4.1 (if CPUID.01H:ECX.SSE4_1[bit 19] = 1), and SSSE3 (if CPUID.01H:ECX.SSSE3[bit 9] = 1).

- Before an application attempts to use the CRC32 instruction, it must check that the processor supports Intel SSE4.2 (if CPUID.01H:ECX.SSE4_2[bit 20] = 1).

- Before an application attempts to use the POPCNT instruction, it must check that the processor supports Intel SSE4.2 (if CPUID.01H:ECX.SSE4_2[bit 20] = 1) and POPCNT (if CPUID.01H:ECX.POPCNT[bit 23] = 1).

12.13 INTEL® AES-NI OVERVIEW

Intel AES-NI provides six instructions to accelerate symmetric block encryption/decryption of 128-bit data blocks using the Advanced Encryption Standard (AES) specified by the NIST publication FIPS 197. Specifically, two instructions (AESENC and AESDECLAST) target the AES encryption rounds; and two instructions (AESDEC and AESDECLAST) target AES decryption rounds using the Equivalent Inverse Cipher. One instruction (AESIMC) targets the Inverse MixColumn transformation primitive, and one instruction (AESKEYGEN) targets generation of round keys from the cipher key for the AES encryption/decryption rounds.

AES supports encryption/decryption using cipher key lengths of 128, 192, and 256 bits by processing the data block in 10, 12, and 14 rounds of predefined transformations. Figure 12-5 depicts the cryptographic processing of a block of 128-bit plain text into cipher text.

![Figure 12-5. AES State Flow](image)

The predefined AES transformation primitives are described in the next few sections, they are also referenced in the operation flow of instruction reference page of these instructions.

12.13.1 Little-Endian Architecture and Big-Endian Specification (FIPS 197)

FIPS 197 document defines the Advanced Encryption Standard (AES) and includes a set of test vectors for testing all of the steps in the algorithm, and can be used for testing and debugging.
The following observation is important for using the AES instructions offered in Intel 64 Architecture: FIPS 197 text convention is to write hex strings with the low-memory byte on the left and the high-memory byte on the right. Intel’s convention is the reverse. It is similar to the difference between Big Endian and Little Endian notations.

In other words, a 128 bits vector in the FIPS document, when read from left to right, is encoded as [7:0, 15:8, 23:16, 31:24, …127:120]. Note that inside the byte, the encoding is [7:0], so the first bit from the left is the most significant bit. In practice, the test vectors are written in hexadecimal notation, where pairs of hexadecimal digits define the different bytes. To translate the FIPS 197 notation to an Intel 64 architecture compatible (“Little Endian”) format, each test vector needs to be byte-reflected to [127:120, … 31:24, 23:16, 15:8, 7:0].

Example A:
FIPS Test vector: 000102030405060708090a0b0c0d0e0fH
Intel AES Hardware: 0f0e0d0c0b0a09080706050403020100H

It should be pointed out that the only thing at issue is a textual convention, and programmers do not need to perform byte-reversal in their code, when using the AES instructions.

12.13.1.1 AES Data Structure in Intel® 64 Architecture

The AES instructions that are defined in this document operate on one or on two 128 bits source operands: State and Round Key. From the architectural point of view, the state is input in an xmm register and the Round key is input either in an xmm register or a 128-bit memory location.

In AES algorithm, the state (128 bits) can be viewed as four 32-bit doublewords (“Words” in AES terminology): X3, X2, X1, and X0.

The state may also be viewed as a set of 16 bytes. The 16 bytes can also be viewed as a 4x4 matrix of bytes where S(i, j) with i, j = 0, 1, 2, 3 compose the 32-bit “words” as follows:

\[
\begin{align*}
X0 & = S (3, 0) S (2, 0) S (1, 0) S (0, 0) \\
X1 & = S (3, 1) S (2, 1) S (1, 1) S (0, 1) \\
X2 & = S (3, 2) S (2, 2) S (1, 2) S (0, 2) \\
X3 & = S (3, 3) S (2, 3) S (1, 3) S (0, 3)
\end{align*}
\]

The following tables, Table 12-8 through Table 12-11, illustrate various representations of a 128-bit state.

### Table 12-8. Byte and 32-Bit Word Representation of a 128-Bit State

<table>
<thead>
<tr>
<th>Byte #</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Word</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Byte</td>
<td>P</td>
<td>O</td>
<td>N</td>
<td>M</td>
<td>L</td>
<td>K</td>
<td>J</td>
<td>I</td>
<td>H</td>
<td>G</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

### Table 12-9. Matrix Representation of a 128-Bit State

|     |     |     |     |     | S(0, 0) | S(0, 1) | S(0, 2) | S(0, 3) | S(1, 0) | S(1, 1) | S(1, 2) | S(1, 3) | S(2, 0) | S(2, 1) | S(2, 2) | S(2, 3) | S(3, 0) | S(3, 1) | S(3, 2) | S(3, 3) |
|-----|-----|-----|-----|-----|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| A   | E   | I   | M   |     | S(0, 0) | S(0, 1) | S(0, 2) | S(0, 3) | S(1, 0) | S(1, 1) | S(1, 2) | S(1, 3) | S(2, 0) | S(2, 1) | S(2, 2) | S(2, 3) | S(3, 0) | S(3, 1) | S(3, 2) | S(3, 3) |
| B   | F   | J   | N   |     | S(1, 0) | S(1, 1) | S(1, 2) | S(1, 3) | S(2, 0) | S(2, 1) | S(2, 2) | S(2, 3) | S(3, 0) | S(3, 1) | S(3, 2) | S(3, 3) |
| C   | G   | K   | O   |     | S(2, 0) | S(2, 1) | S(2, 2) | S(2, 3) | S(3, 0) | S(3, 1) | S(3, 2) | S(3, 3) |
| D   | H   | L   | P   |     | S(3, 0) | S(3, 1) | S(3, 2) | S(3, 3) |

Example:
FIPS vector: d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5
This vector has the "least significant" byte d4 and the significant byte e5 (written in Big Endian format in the FIPS document). When it is translated to IA notations, the encoding is:

| Table 12-10. Little Endian Representation of a 128-Bit State |
|----------------------------------|---|---|---|---|---|---|---|---|---|---|---|---|
| Byte # | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| State Byte | P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A |
| State Value | e5 | 98 | 27 | 1e | f1 | 11 | 41 | b8 | ae | 52 | b4 | e0 | 30 | 5d | bf | d4 |

| Table 12-11. Little Endian Representation of a 4x4 Byte Matrix |
|----------------------------------|---|---|---|---|---|---|---|---|---|---|---|---|
| A | E | I | M | d4 | e0 | b8 | 1e |
| B | F | J | N | bf | b4 | 41 | 27 |
| C | G | K | O | 5d | 52 | 11 | 98 |
| D | H | L | P | 30 | ae | f1 | e5 |

### 12.13.2 AES Transformations and Functions

The following functions and transformations are used in the algorithmic descriptions of AES instruction extensions AESDEC, AESDECLAST, AESENCLAST, AESIMC, and AESKEYGENASSIST.

Note that these transformations are expressed here in a Little Endian format (and not as in the FIPS 197 document).

- **MixColumns():** A byte-oriented 4x4 matrix transformation on the matrix representation of a 128-bit AES state. A FIPS-197 defined 4x4 matrix is multiplied to each 4x1 column vector of the AES state. The columns are considered polynomials with coefficients in the Finite Field that is used in the definition of FIPS 197, the operations ("multiplication" and "addition") are in that Finite Field, and the polynomials are reduced modulo \( x^4+1 \).

The MixColumns() transformation defines the relationship between each byte of the result state, represented as \( S'(i, j) \) of a 4x4 matrix (see Section 12.13.1), as a function of input state bytes, \( S(i, j) \), as follows:

\[
\begin{align*}
S'(0, 0) &= \text{FF_MUL}(02H, S(0, 0)) \text{ XOR } \text{FF_MUL}(03H, S(1, 0)) \text{ XOR } S(2, 0) \text{ XOR } S(3, 0) \\
S'(0, 1) &= S(0, 1) \text{ XOR } \text{FF_MUL}(02H, S(1, 1)) \text{ XOR } \text{FF_MUL}(03H, S(2, 1)) \text{ XOR } S(3, 1) \\
S'(0, 2) &= S(0, 2) \text{ XOR } S(1, 1) \text{ XOR } \text{FF_MUL}(02H, S(2, 2)) \text{ XOR } \text{FF_MUL}(03H, S(3, 2)) \\
S'(0, 3) &= \text{FF_MUL}(03H, S(0, 0)) \text{ XOR } S(1, 1) \text{ XOR } S(2, 2) \text{ XOR } \text{FF_MUL}(02H, S(3, 3))
\end{align*}
\]

where \( j = 0, 1, 2, 3 \). \text{FF_MUL(Byte1, Byte2)} denotes the result of multiplying two elements (represented by Byte1 and Byte2) in the Finite Field representation that defines AES. The result of produced byte \text{FF_MUL(Byte1, Byte2)} is an element in the Finite Field (represented as a byte). A Finite Field is a field with a finite number of elements, and when this number can be represented as a power of 2 (2\( n \)), its elements can be represented as the set of 2\( n \) binary strings of length \( n \). AES uses a finite field with \( n=8 \) (having 256 elements). With this representation, “addition” of two elements in that field is a bit-wise XOR of their binary-string representation, producing another element in the field. Multiplication of two elements in that field is defined using an irreducible polynomial (for AES, this polynomial is \( m(x) = x^8 + x^4 + x^3 + x + 1 \)). In this Finite Field representation, the bit value of bit position \( k \) of a byte represents the coefficient of a polynomial of order \( k \), e.g., 1010_1101B (ADH) is represented by the polynomial \((x^7 + x^5 + x^3 + x^2 + 1)\). The byte value of the polynomial is obtained by a carry-less multiplication of the two corresponding polynomials, followed by reduction modulo the polynomial, where the remainder is calculated using operations defined in the field. For example, \text{FF_MUL}(57H, 83H) = C1H, because the carry-less polynomial multiplication of the polynomials represented by 57H and 83H produces \((x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1)\), and the remainder modulo \( m(x) \) is \((x^6 + x^5 + 1)\).

- **RotWord():** performs a byte-wise cyclic permutation (rotate right in little-endian byte order) on a 32-bit AES word.
The output word $X'[j]$ of RotWord($X[j]$) where $X[j]$ represent the four bytes of column $j$, $S(i, j)$, in descending order $X[j] = (S(3, j), S(2, j), S(1, j), S(0, j))$; $X'[j] = (S'(3, j), S'(2, j), S'(1, j), S'(0, j)) := (S(0, j), S(3, j), S(2, j), S(1, j))$

- **ShiftRows():** A byte-oriented matrix transformation that processes the matrix representation of a 16-byte AES state by cyclically shifting the last three rows of the state by different offset to the left, see Table 12-12.

<table>
<thead>
<tr>
<th>Matrix Representation of Input State</th>
<th>Output of ShiftRows</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  E  I  M</td>
<td>A  E  I  M</td>
</tr>
<tr>
<td>B  F  J  N</td>
<td>F  J  N  B</td>
</tr>
<tr>
<td>C  G  K  O</td>
<td>K  O  C  G</td>
</tr>
<tr>
<td>D  H  L  P</td>
<td>P  D  H  L</td>
</tr>
</tbody>
</table>

  - **SubBytes():** A byte-oriented transformation that processes the 128-bit AES state by applying a non-linear substitution table (S-BOX) on each byte of the state.

    The SubBytes() function defines the relationship between each byte of the result state $S'(i, j)$ as a function of input state byte $S(i, j)$, by

    $S'(i, j) := S\text{-Box}(S(i, j)[7:4], S(i, j)[3:0])$

    where $S\text{-Box}(S[7:4], S[3:0])$ represents a look-up operation on a 16x16 table to return a byte value, see Table 12-13.

<table>
<thead>
<tr>
<th>Table 12-13. Look-up Table Associated with S-Box Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

  - **SubWord():** produces an output AES word (four bytes) from the four bytes of an input word using a non-linear substitution table (S-BOX).
X'[j] = ( S'(3, j), S'(2, j), S'(1, j), S'(0, j) ) := ( S-Box (S(3, j)), S-Box( S(2, j) ), S-Box( S(1, j) ), S-Box( S(0, j) ) )

• InvMixColumns(): The inverse transformation of MixColumns().

The InvMixColumns() transformation defines the relationship between each byte of the result state S'(i, j) as a function of input state bytes, S(i, j), by

S'(0, j) := FF_MUL( 0eH, S(0, j) ) XOR FF_MUL(0bH, S(1, j) ) XOR FF_MUL(0dH, S(2, j) ) XOR FF_MUL( 09H, S(3, j) )
S'(1, j) := FF_MUL(09H, S(0, j) ) XOR FF_MUL( 0eH, S(1, j) ) XOR FF_MUL(0bH, S(2, j) ) XOR FF_MUL( 0dH, S(3, j) )
S'(2, j) := FF_MUL(0dH, S(0, j) ) XOR FF_MUL( 09H, S(1, j) ) XOR FF_MUL( 0eH, S(2, j) ) XOR FF_MUL(0bH, S(3, j) )
S'(3, j) := FF_MUL(0bH, S(0, j) ) XOR FF_MUL(0dH, S(1, j) ) XOR FF_MUL( 09H, S(2, j) ) XOR FF_MUL( 0eH, S(3, j) ), where j = 0, 1, 2, 3.

• InvShiftRows(): The inverse transformation of InvShiftRows(). The InvShiftRows() transforms the matrix representation of a 16-byte AES state by cyclically shifting the last three rows of the state by different offset to the right, see Table 12-14.

Table 12-14. The InvShiftRows Transformation

<table>
<thead>
<tr>
<th>Matrix Representation of Input State</th>
<th>Output of ShiftRows</th>
</tr>
</thead>
<tbody>
<tr>
<td>A    E    I    M</td>
<td>A    E    I    M</td>
</tr>
<tr>
<td>B    F    J    N</td>
<td>N    B    F    J</td>
</tr>
<tr>
<td>C    G    K    O</td>
<td>K    O    C    G</td>
</tr>
<tr>
<td>D    H    L    P</td>
<td>H    L    P    D</td>
</tr>
</tbody>
</table>

• InvSubBytes(): The inverse transformation of SubBytes().

The InvSubBytes() transformation defines the relationship between each byte of the result state S'(i, j) as a function of input state byte S(i, j), by

S'(i, j) := InvS-Box (S(i, j)[7:4], S(i, j)[3:0])

where InvS-BOX (S[7:4], S[3:0]) represents a look-up operation on a 16x16 table to return a byte value, see Table 12-15.
12.13.3 PCLMULQDQ

The PCLMULQDQ instruction performs carry-less multiplication of two 64-bit data into a 128-bit result. Carry-less multiplication of two 128-bit data into a 256-bit result can use PCLMULQDQ as building blocks.

Carry-less multiplication is a component of many cryptographic systems. It is an important piece of implementing Galois Counter Mode (GCM) operation of block ciphers. GCM operation can be used in conjunction with AES algorithms to add authentication capability. GCM usage models also include IPsec, storage standard, and security protocols over fiber channel. Additionally, PCLMULQDQ can be used in calculations of hash functions and CRC using arbitrary polynomials.

12.13.4 Checking for Intel® AES-NI Support

Before an application attempts to use AESNI instructions or PCLMULQDQ, the application should follow the steps illustrated in Section 11.6.2, "Checking for Intel® SSE and SSE2 Support." Next, use the additional step provided below:

Check that the processor supports Intel AES-NI (if CPUID.01H:ECX.AESNI[bit 25] = 1); check that the processor supports PCLMULQDQ (if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1).
The XSAVE feature set extends the functionality of the FXSAVE and FXRSTOR instructions (see Section 10.5, "FXSAVE and FXRSTOR Instructions") by supporting the saving and restoring of processor state in addition to the x87 execution environment (x87 state) and the registers used by the streaming SIMD extensions (SSE state).

The XSAVE feature set comprises eight instructions. XGETBV and XSETBV allow software to read and write the extended control register XCR0, which controls the operation of the XSAVE feature set. XSAVE, XSAVEOPT, XSAVEC, and XSAVES are four instructions that save processor state to memory; XRSTOR and XRSTORS are corresponding instructions that load processor state from memory. XGETBV, XSAVE, XSAXOPT, XSAVEC, and XRSTOR can be executed at any privilege level; XSETBV, XSAVES, and XRSTORS can be executed only if CPL = 0. In addition to XCR0, the XSAVES and XRSTORS instructions are controlled also by the IA32_XSS MSR (index DA0H).

The XSAVE feature set organizes the state that manages into state components. Operation of the instructions is based on state-component bitmaps that have the same format as XCR0 and as the IA32_XSS MSR: each bit corresponds to a state component. Section 13.1 discusses these state components and bitmaps in more detail.

Section 13.2 describes how the processor enumerates support for the XSAVE feature set and for XSAVE-enabled features (those features that require the use of the XSAVE feature set for their enabling). Section 13.3 explains how software can enable the XSAVE feature set and XSAVE-enabled features.

The XSAVE feature set allows saving and loading processor state from a region of memory called an XSAVE area. Section 13.4 presents details of the XSAVE area and its organization. Each XSAVE-managed state component is associated with a section of the XSAVE area. Section 13.5 describes in detail each of the XSAVE-managed state components.

Section 13.7 through Section 13.12 describe the operation of XSAVE, XRSTOR, XSAVEOPT, XSAVEC, XSAVES, and XRSTORS, respectively.

Section 13.13 provides some details about memory accesses performed by instructions in the XSAVE feature set, and Section 13.14 describes a facility called extended feature disable (XFD).

### 13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS

The XSAVE feature set supports the saving and restoring of state components, each of which is a discrete set of processor registers (or parts of registers). In general, each such state component corresponds to a particular CPU feature. Such a feature is XSAVE-supported. Some XSAVE-supported features use registers in multiple XSAVE-managed state components.

The XSAVE feature set organizes the state components of the XSAVE-supported features using state-component bitmaps. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. The following bits are defined in state-component bitmaps (details on individual state components are provided in subsections of Section 13.5):

- Bit 0 corresponds to the state component used for the x87 FPU execution environment (x87 state).
- Bit 1 corresponds to the state component used for registers used by the streaming SIMD extensions (SSE state).
- Bit 2 corresponds to the state component used for the additional register state used by the Intel® Advanced Vector Extensions (AVX state).
- Bits 4:3 correspond to the two state components used for the additional register state used by Intel® Memory Protection Extensions (MPX state):
  - State component 3 is used for the 4 128-bit bounds registers BND0–BND3 (BNDREGS state).
  - State component 4 is used for the 64-bit user-mode MPX configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (BNDCSR state).
- Bits 7:5 correspond to the three state components used for the additional register state used by Intel® Advanced Vector Extensions 512 (AVX-512 state):
State component 5 is used for the 8 64-bit opmask registers k0–k7 (opmask state).

State component 6 is used for the upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0_H–ZMM15_H (ZMM_Hi256 state).

State component 7 is used for the 16 512-bit registers ZMM16–ZMM31 (Hi16_ZMM state).

- Bit 8 corresponds to the state component used for the Intel Processor Trace MSRs (PT state).
- Bit 9 corresponds to the state component used for the protection-key feature’s register PKRU (PKRU state).
- Bit 10 corresponds to the state component used for the IA32_PASID MSR used by the ENQCMD instruction for a process address space identifiers (PASID state).
- Bits 12:11 correspond to the two state components used for the additional register state used by Control-Flow Enforcement Technology (CET state):
  - State component 11 is used for the 2 MSRs controlling user-mode functionality for CET (CET_U state).
  - State component 12 is used for the 3 MSRs containing shadow-stack pointers for privilege levels 0–2 (CET_S state).
- Bit 13 corresponds to the state component used for an MSR used to control hardware duty cycling (HDC state).
- Bit 14 corresponds to the state component used for user interrupts (UINTR state).
- Bit 15 corresponds to the state component used for last-branch record configuration (LBR state).
- Bit 16 corresponds to the state component used for an MSR used to control hardware P-states (HWP state).
- Bits 18:17 correspond to the two state components used for the additional register state used by Intel® Advanced Matrix Extensions (AMX state):
  - State component 17 is used for the 64-byte TILECFG register (TILECFG state).
  - State component 18 is used for the 8192 bytes of tile data (TILEDATA state).

Bits in the range 62:19 are not currently defined in state-component bitmaps and are reserved for future expansion. As individual state components are defined using those bits, additional sub-sections will be updated within Section 13.5 over time. Bit 63 is used for special functionality in some bitmaps and does not correspond to any state component.

The state component corresponding to bit \( i \) of state-component bitmaps is called state component \( i \). Thus, x87 state is state component 0; SSE state is state component 1; AVX state is state component 2; MPX state comprises state components 3–4; AVX-512 state comprises state components 5–7; PT state is state component 8; PKRU state is state component 9; PASID state is state component 10; CET state comprises state components 11–12; HDC state is state component 13; UINTR state is state component 14; LBR state is state component 15; HWP state is state component 16; AMX state comprises state components 17–18.

The XSAVE feature set uses state-component bitmaps in multiple ways. Most of the instructions use an implicit operand (in EDX:EAX), called the instruction mask, which is the state-component bitmap that specifies the state components on which the instruction operates.

Some state components are user state components, and they can be managed by the entire XSAVE feature set. Other state components are supervisor state components, and they can be managed only by XSAVES and XRSTORS. The state components corresponding to bit 9, to bits 18:17, and to bits in the range 7:0 are user state components; those corresponding to bit 8, to bits in the range 13:10, and to bits 16:14 are supervisor state components.

Extended control register XCR0 contains a state-component bitmap that specifies the user state components that software has enabled the XSAVE feature set to manage. If the bit corresponding to a state component is clear in XCR0, instructions in the XSAVE feature set will not operate on that state component, regardless of the value of the instruction mask.

The IA32_XSS MSR (index DA0H) contains a state-component bitmap that specifies the supervisor state components that software has enabled XSAVES and XRSTORS to manage (XSAVE, XSAVEC, XSAVEmP, and XRSTOR cannot manage supervisor state components). If the bit corresponding to a state component is clear in the IA32_XSS MSR, XSAVES and XRSTORS will not operate on that state component, regardless of the value of the instruction mask.
Some XSAVE-supported features can be used only if XCR0 has been configured so that the features’ state components can be managed by the XSAVE feature set. (This applies only to features with user state components.) Such state components and features are XSAVE-enabled. In general, the processor will not modify (or allow modification of) the registers of a state component of an XSAVE-enabled feature if the bit corresponding to that state component is clear in XCR0. (If software clears such a bit in XCR0, the processor preserves the corresponding state component.) If an XSAVE-enabled feature has not been fully enabled in XCR0, execution of any instruction defined for that feature causes an invalid-opcode exception (#UD).

As will be explained in Section 13.3, the XSAVE feature set is enabled only if CR4.OSXSAVE[bit 18] = 1. If CR4.OSXSAVE = 0, the processor treats XSAVE-enabled state features and their state components as if all bits in XCR0 were clear; the state components cannot be modified and the features’ instructions cannot be executed.

The state components for x87 state, for SSE state, for PT state, for PKRU state, for PASID state, for CET state, for HDC state, for UINTR state, for LBR state, and for HWP state are XSAVE-managed but the corresponding features are not XSAVE-enabled. Processors allow modification of this state, as well as execution of x87 FPU instructions and SSE instructions and use of Intel Processor Trace, protection keys, the ENQCMD instruction and the IA32_PASID MSR, CET, hardware duty cycling, user interrupts, LBRs, and hardware P-states, regardless of the value of CR4.OSXSAVE and XCR0.

### 13.2 Enumeration of CPU Support for XSAVE Instructions and XSAVE-Supported Features

A processor enumerates support for the XSAVE feature set and for features supported by that feature set using the CPUID instruction. The following items provide specific details:

- **CPUID.1:ECX.XSAVE[bit 26]** enumerates general support for the XSAVE feature set:
  - If this bit is 0, the processor does not support any of the following instructions: XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV; the processor provides no further enumeration through CPUID function 0DH (see below).
  - If this bit is 1, the processor supports the following instructions: XGETBV, XRSTOR, XSAVE, and XSETBV. Further enumeration is provided through CPUID function 0DH.

CR4.OSXSAVE can be set to 1 if and only if CPUID.1:ECX.XSAVE[bit 26] is enumerated as 1.

- **CPUID function 0DH** enumerates details of CPU support through a set of sub-functions. Software selects a specific sub-function by the value placed in the ECX register. The following items provide specific details:
  - **CPUID function 0DH, sub-function 0.**
    - EDX:EAX is a bitmap of all the user state components that can be managed using the XSAVE feature set. A bit can be set in XCR0 if and only if the corresponding bit is set in this bitmap. Every processor that supports the XSAVE feature set will set EAX[0] (x87 state) and EAX[1] (SSE state).
      - If EAX[i] = 1 (for 1 < i < 32) or EDX[i–32] = 1 (for 32 ≤ i < 63), sub-function i enumerates details for state component i (see below).
    - **ECX** enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components supported by this processor.
    - **EBX** enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components corresponding to bits currently set in XCR0.
  - **CPUID function 0DH, sub-function 1.**
    - **EAX[0]** enumerates support for the XSAVEOPT instruction. The instruction is supported if and only if this bit is 1. If EAX[0] = 0, execution of XSAVEOPT causes an invalid-opcode exception (#UD).
    - **EAX[1]** enumerates support for compaction extensions to the XSAVE feature set. The following are supported if this bit is 1:

1. If CPUID.1:ECX.XSAVE[bit 26] = 1, XGETBV and XSETBV may be executed with ECX = 0 (to read and write XCR0). Any support for execution of these instructions with other values of ECX is enumerated separately.
— The compacted format of the extended region of XSAVE areas (see Section 13.4.3).
— The XSAVEC instruction. If EAX[1] = 0, execution of XSAVEC causes a #UD.
— Execution of the compacted form of XRSTOR (see Section 13.8).

- EAX[2] enumerates support for execution of XGETBV with ECX = 1. This allows software to determine the state of the init optimization. See Section 13.6.
- EAX[3] enumerates support for XSAVES, XRSTORS, and the IA32_XSS MSR. If EAX[3] = 0, execution of XSAVES or XRSTORS causes a #UD; an attempt to access the IA32_XSS MSR using RDMSR or WRMSR causes a general-protection exception (#GP). Every processor that supports a supervisor state component sets EAX[3]. Every processor that sets EAX[3] (XSAVES, XRSTORS, IA32_XSS) will also set EAX[1] (the compaction extensions).
- EAX[31:5] are reserved.
- EBX enumerates the size (in bytes) defined as follows:
  - If EAX[3] is enumerated as 1, EBX enumerates the size required by the XSAVES instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0 | IA32_XSS.
  - If EAX[3] is enumerated as 0 and EAX[1] is enumerated as 1, EBX enumerates the size required by the XSAVEC instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0.
  - If EAX[1] and EAX[3] are both enumerated as 0, EBX enumerates zero.
- EDX:ECX is a bitmap of all the supervisor state components that can be managed by XSAVES and XRSTORS. A bit can be set in the IA32_XSS MSR if and only if the corresponding bit is set in this bitmap.

**NOTE**

In summary, the XSAVE feature set supports state component \(i\) (\(0 \leq i < 63\)) if one of the following is true: (1) \(i < 32\) and CPUID.(EAX=0DH,ECX=0):EAX[\(i\)] = 1; (2) \(i \geq 32\) and CPUID.(EAX=0DH,ECX=0):EAX[\(i-32\)] = 1; (3) \(i < 32\) and CPUID.(EAX=0DH,ECX=1):ECX[\(i\)] = 1; or (4) \(i \geq 32\) and CPUID.(EAX=0DH,ECX=1):EDX[\(i-32\)] = 1. The XSAVE feature set supports user state component \(i\) if (1) or (2) holds; if (3) or (4) holds, state component \(i\) is a supervisor state component and support is limited to XSAVES and XRSTORS.

- CPUID function 0DH, sub-function \(i\) (\(i > 1\)). This sub-function enumerates details for state component \(i\). If the XSAVE feature set supports state component \(i\) (see note above), the following items provide specific details:
  - EAX enumerates the size (in bytes) required for state component \(i\).
  - If state component \(i\) is a user state component, EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section used for state component \(i\). (This offset applies only when the standard format for the extended region of the XSAVE area is being used; see Section 13.4.3.)
  - If state component \(i\) is a supervisor state component, EBX returns 0.
  - If state component \(i\) is a user state component, ECX[0] return 0; if state component \(i\) is a supervisor state component, ECX[0] returns 1.
  - The value returned by ECX[1] indicates the alignment of state component \(i\) when the compacted format of the extended region of an XSAVE area is used (see Section 13.4.3). If ECX[1] returns 0, state component \(i\) is located immediately following the preceding state component; if ECX[1] returns 1, state component \(i\) is located on the next 64-byte boundary following the preceding state component.
  - If the processor supports XFD for state component \(i\), ECX[2] returns 1; otherwise, ECX[2] returns 0.
  - ECX[31:3] and EDX return 0.

If the XSAVE feature set does not support state component \(i\), sub-function \(i\) returns 0 in EAX, EBX, ECX, and EDX.
13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES

Software enables the XSAVE feature set by setting CR4.OSXSAVE[bit 18] to 1 (e.g., with the MOV to CR4 instruction). If this bit is 0, execution of any of XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV causes an invalid-opcode exception (#UD).

When CR4.OSXSAVE = 1 and CPL = 0, executing the XSETBV instruction with ECX = 0 writes the 64-bit value in EDX:EAX to XCR0 (EAX is written to XCR0[31:0] and EDX to XCR0[63:32]). (Execution of the XSETBV instruction causes a general-protection fault — #GP — if CPL > 0.) The following items provide details regarding individual bits in XCR0:

- **XCR0[0]** is associated with x87 state (see Section 13.5.1). XCR0[0] is always 1. It has that value coming out of RESET. Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[0] is 0.

- **XCR0[1]** is associated with SSE state (see Section 13.5.2). Software can use the XSAVE feature set to manage SSE state only if XCR0[1] = 1. The value of XCR0[1] in no way determines whether software can execute SSE instructions (these instructions can be executed even if XCR0[1] = 0).

XCR0[1] is 0 coming out of RESET. As noted in Section 13.2, every processor that supports the XSAVE feature set allows software to set XCR0[1].

- **XCR0[2]** is associated with AVX state (see Section 13.5.3). Software can use the XSAVE feature set to manage AVX state only if XCR0[2] = 1. In addition, software can execute Intel AVX instructions only if CR4.OSXSAVE = XCR0[2] = 1. Otherwise, any execution of an Intel AVX instruction causes an invalid-opcode exception (#UD).

XCR0[2] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[2] if and only if CPUID.(EAX=0DH,ECX=0):EAX[2] = 1. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[2:1] has the value 10b; that is, software cannot enable the XSAVE feature set for AVX state but not for SSE state.

As noted in Section 13.1, the processor will preserve AVX state unmodified if software clears XCR0[2]. However, clearing XCR0[2] while AVX state is not in its initial configuration may cause SSE instructions to incur a power and performance penalty. See Section 14.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State Components," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for how system software can avoid this penalty.

- **XCR0[4:3]** are associated with MPX state (see Section 13.5.4). Software can use the XSAVE feature set to manage MPX state only if XCR0[4:3] = 11b. In addition, MPX instructions operate as defined only if CR4.OSXSAVE = XCR0[2] = 11b. Otherwise, execution of an MPX instruction causes no operation (as a NOP instruction); in addition, executions of CALL, RET, JMP, and Jcc do not initialize the bounds registers, and they ignore any F2H (BND) prefix.\(^1\)

XCR0[4:3] have value 00b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[4:3] to 11b if and only if CPUID.(EAX=0DH,ECX=0):EAX[4:3] = 11b. Otherwise, execution of an MPX instruction causes a general-protection fault (#GP) if ECX = 0, EAX[4:3] is neither 00b nor 11b; that is, software can enable the XSAVE feature set for MPX state only if it does so for both state components.

As noted in Section 13.1, the processor will preserve MPX state unmodified if software clears XCR0[4:3].

- **XCR0[7:5]** are associated with AVX-512 state (see Section 13.5.5). Software can use the XSAVE feature set to manage AVX-512 state only if XCR0[7:5] = 111b. In addition, software can execute Intel AVX-512 instructions only if CR4.OSXSAVE = XCR0[2] = 111b. Otherwise, any execution of an Intel AVX-512 instruction causes an invalid-opcode exception (#UD).

XCR0[7:5] have value 00b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[7:5] to 11b if and only if CPUID.(EAX=0DH,ECX=0):EAX[7:5] = 11b. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0, EAX[7:5] is not 00b, and any bit is clear in EAX[2:1] or EAX[7:5]; that is, software can enable the XSAVE feature set for AVX-512 state only if it does so for all three state components, and only if it also does so for AVX state and SSE state. This implies that the value of XCR0[7:5] is always either 000b or 111b.

As noted in Section 13.1, the processor will preserve AVX-512 state unmodified if software clears XCR0[7:5]. However, clearing XCR0[7:5] while AVX-512 state is not in its initial configuration may cause SSE and Intel AVX instructions to incur a power and performance penalty. See Section 14.5.3, "Enable the Use Of XSAVE Feature

---

1. Prior to the introduction of MPX, the opcodes defining MPX instructions operated as NOP, and the CALL, RET, JMP, and Jcc instructions ignored any F2H prefix.
Set And XSAVE State Components,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for how system software can avoid this penalty.

- **XCR0[9]** is associated with PKRU state (see Section 13.5.7). Software can use the XSAVE feature set to manage PKRU state only if XCR0[9] = 1. The value of XCR0[9] in no way determines whether software can use protection keys or execute other instructions that access PKRU state (these instructions can be executed even if XCR0[9] = 0).

  XCR0[9] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[9] if and only if CPUID.(EAX=0DH,ECX=0):EAX[9] = 1.

- **XCR0[18:17]** are associated with AMX state (see Section 13.5.5). Software can use the XSAVE feature set to manage AMX state only if XCR0[18:17] = 11b. In addition, software can execute Intel AMX instructions only if CR4.OSXSAVE = 1 and XCR0[18:17] = 11b. Otherwise, any execution of an Intel AMX instruction causes an invalid-opcode exception (#UD).

  XCR0[18:17] have value 00b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[18:17] to 11b if and only if CPUID.(EAX=0DH,ECX=0):EAX[18:17] = 11b. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[17] ≠ EAX[18] (TILECFG and TILEDATA must be enabled together). This implies that the value of XCR0[18:17] is always either 00b or 11b.

  While Intel AMX instructions can be executed only in 64-bit mode, instructions of the XSAVE feature set can operate on TILECFG and TILEDATA in any mode. It is recommended that only 64-bit operating systems enable Intel AMX by setting XCR0[18:17].

- **XCR0[63:19], XCR0[16:10], and XCR0[8]** are reserved.1 Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

Software operating with CPL > 0 may need to determine whether the XSAVE feature set and certain XSAVE-enabled features have been enabled. If CPL > 0, execution of the MOV from CR4 instruction causes a general-protection fault (#GP). The following alternative mechanisms allow software to discover the enabling of the XSAVE feature set regardless of CPL:

- The value of CR4.OSXSAVE is returned in CPUID.1:ECX.OSXSAVE[bit 27]. If software determines that CPUID.1:ECX.OSXSAVE = 1, the processor supports the XSAVE feature set and the feature set has been enabled in CR4.

- Executing the XGETBV instruction with ECX = 0 returns the value of XCR0 in EDX:EAX. XGETBV can be executed if CR4.OSXSAVE = 1 (if CPUID.1:ECX.OSXSAVE = 1), regardless of CPL.

Thus, software can use the following algorithm to determine the support and enabling for the XSAVE feature set:

1. Use CPUID to discover the value of CPUID.1:ECX.OSXSAVE.
   - If the bit is 0, either the XSAVE feature set is not supported by the processor or has not been enabled by software. Either way, the XSAVE feature set is not available, nor are XSAVE-enabled features such as AVX.
   - If the bit is 1, the processor supports the XSAVE feature set — including the XGETBV instruction — and it has been enabled by software. The XSAVE feature set can be used to manage x87 state (because XCR0[0] is always 1). Software requiring more detailed information can go on to the next step.

2. Execute XGETBV with ECX = 0 to discover the value of XCR0. If XCR0[1] = 1, the XSAVE feature set can be used to manage SSE state. If XCR0[2] = 1, the XSAVE feature set can be used to manage AVX state and software can execute Intel AVX instructions. If XCR0[4:3] is 11b, the XSAVE feature set can be used to manage MPX state and software can execute Intel MPX instructions. If XCR0[7:5] is 11b, the XSAVE feature set can be used to manage AVX-512 state and software can execute Intel AVX-512 instructions. If XCR0[9] = 1, the XSAVE feature set can be used to manage PKRU state.

The IA32_XSS MSR (with MSR index DA0H) is zero coming out of RESET. If CR4.OSXSAVE = 1, CPUID.(EAX=0DH,ECX=1):EAX[3] = 1, and CPL = 0, executing the WRMSR instruction with ECX = DA0H writes the 64-bit value in EDX:EAX to the IA32_XSS MSR (EAX is written to IA32_XSS[31:0] and EDX to IA32_XSS[63:32]). The following items provide details regarding individual bits in the IA32_XSS MSR:

---

1. Bit 8 and bits 16:10 correspond to supervisor state components. Since bits can be set in XCR0 only for user state components, those bits of XCR0 must be 0.
MANAGING STATE USING THE XSAVE FEATURE SET

• IA32_XSS[8] is associated with PT state (see Section 13.5.6). Software can use XSAVES and XRSTORS to manage PT state only if IA32_XSS[8] = 1. The value of IA32_XSS[8] does not determine whether software can use Intel Processor Trace (the feature can be used even if IA32_XSS[8] = 0).

• IA32_XSS[10] is associated with PASID state (see Section 13.5.8). Software can use the XSAVES and XRSTORS to manage PASID state only if IA32_XSS[10] = 1. The value of IA32_XSS[10] does not determine whether software can use the ENQCMD instruction, which uses the IA32_PASID MSR. (ENQCMD can be used even if IA32_XSS[10] is 0.)


• IA32_XSS[13] is associated with HDC state (see Section 13.5.10). Software can use XSAVES and XRSTORS to manage HDC state only if IA32_XSS[13] = 1. The value of IA32_XSS[13] does not determine whether software can use hardware duty cycling (the feature can be used even if IA32_XSS[13] = 0).

• IA32_XSS[14] is associated with UINTR state (see Section 13.5.11). Software can use XSAVES and XRSTORS to manage UINTR state only if IA32_XSS[14] = 1. The value of IA32_XSS[14] does not determine whether software can use user interrupts (the feature can be used even if IA32_XSS[14] = 0).

• IA32_XSS[15] is associated with LBR state (see Section 13.5.12). Software can use XSAVES and XRSTORS to manage LBR state only if IA32_XSS[15] = 1. The value of IA32_XSS[15] does not determine whether software can use LBRs (the feature can be used even if IA32_XSS[15] = 0).

• IA32_XSS[16] is associated with HWP state (see Section 13.5.13). Software can use XSAVES and XRSTORS to manage HWP state only if IA32_XSS[16] = 1. The value of IA32_XSS[16] does not determine whether software can use hardware P-states (the feature can be used even if IA32_XSS[16] = 0).

• IA32_XSS[63:17], IA32_XSS[9] and IA32_XSS[7:0] are reserved. Executing the WRMSR instruction causes a general-protection fault (#GP) if ECX = DA0H and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

The IA32_XSS MSR is 0 coming out of RESET.

There is no mechanism by which software operating with CPL > 0 can discover the value of the IA32_XSS MSR.

13.4 XSAVE AREA

The XSAVE feature set includes instructions that save and restore the XSAVE-managed state components to and from memory: XSAVE, XSAVEOPT, XSAVEC, and XSAVES (for saving); and XRSTOR and XRSTORS (for restoring). The processor organizes the state components in a region of memory called an XSAVE area. Each of the save and restore instructions takes a memory operand that specifies the 64-byte aligned base address of the XSAVE area on which it operates.

Every XSAVE area has the following format:

• The legacy region. The legacy region of an XSAVE area comprises the 512 bytes starting at the area’s base address. It is used to manage the state components for x87 state and SSE state. The legacy region is described in more detail in Section 13.4.1.

• The XSAVE header. The XSAVE header of an XSAVE area comprises the 64 bytes starting at an offset of 512 bytes from the area’s base address. The XSAVE header is described in more detail in Section 13.4.2.

• The extended region. The extended region of an XSAVE area starts at an offset of 576 bytes from the area’s base address. It is used to manage the state components other than those for x87 state and SSE state. The extended region is described in more detail in Section 13.4.3. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 and IA32_XSS (see Section 13.3).

1. Bit 9 and bits 7:0 correspond to user state components. Since bits can be set in the IA32_XSS MSR only for supervisor state components, those bits of the MSR must be 0.
13.4.1 Legacy Region of an XSAVE Area

The legacy region of an XSAVE area comprises the 512 bytes starting at the area’s base address. It has the same format as the FXSAVE area (see Section 10.5.1). The XSAVE feature set uses the legacy area for x87 state (state component 0) and SSE state (state component 1). Table 13-1 illustrates the format of the first 416 bytes of the legacy region of an XSAVE area.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIP[63:48] or reserved</td>
<td>FCS or FIP[47:32]</td>
<td>FIP[31:0]</td>
<td>FOP</td>
<td>Rsvd.</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td>FDP[63:48] or reserved</td>
<td>FDS or FDP[47:32]</td>
<td>FDP[31:0]</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Reserved | ST0/MM0 | 32 |
| Reserved | ST1/MM1 | 48 |
| Reserved | ST2/MM2 | 64 |
| Reserved | ST3/MM3 | 80 |
| Reserved | ST4/MM4 | 96 |
| Reserved | ST5/MM5 | 112 |
| Reserved | ST6/MM6 | 128 |
| Reserved | ST7/MM7 | 144 |

XMM0 160
XMM1 176
XMM2 192
XMM3 208
XMM4 224
XMM5 240
XMM6 256
XMM7 272
XMM8 288
XMM9 304
XMM10 320
XMM11 336
XMM12 352
XMM13 368
XMM14 384
XMM15 400

The x87 state component comprises bytes 23:0 and bytes 159:32. The SSE state component comprises bytes 31:24 and bytes 415:160. The XSAVE feature set does not use bytes 511:416; bytes 463:416 are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the legacy region of an XSAVE area.
13.4.2 XSAVE Header

The XSAVE header of an XSAVE area comprises the 64 bytes starting at offset 512 from the area’s base address:

- Bytes 7:0 of the XSAVE header is a state-component bitmap (see Section 13.1) called XSTATE_BV. It identifies the state components in the XSAVE area.
- Bytes 15:8 of the XSAVE header is a state-component bitmap called XCOMP_BV. It is used as follows:
  - XCOMP_BV[63] indicates the format of the extended region of the XSAVE area (see Section 13.4.3). If it is clear, the standard format is used. If it is set, the compacted format is used; XCOMP_BV[62:0] provide format specifics as specified in Section 13.4.3.
  - XCOMP_BV[63] determines which form of the XRSTOR instruction is used. If the bit is set, the compacted form is used; otherwise, the standard form is used. See Section 13.8.
  - All bits in XCOMP_BV should be 0 if the processor does not support the compaction extensions to the XSAVE feature set.
- Bytes 63:16 of the XSAVE header are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the XSAVE header of an XSAVE area.

13.4.3 Extended Region of an XSAVE Area

The extended region of an XSAVE area starts at byte offset 576 from the area’s base address. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 | IA32_XSS (see Section 13.3). The XSAVE feature set uses the extended area for each state component i, where i ≥ 2.

The extended region of an XSAVE area may have one of two formats. The standard format is supported by all processors that support the XSAVE feature set; the compacted format is supported by those processors that support the compaction extensions to the XSAVE feature set (see Section 13.2). Bit 63 of the XCOMP_BV field in the XSAVE header (see Section 13.4.2) indicates which format is used.

The following items describe the two possible formats of the extended region:

- **Standard format.** Each state component i (i ≥ 2) is located at the byte offset from the base address of the XSAVE area enumerated in CPUID.(EAX=0DH,ECX=i):EBX. (CPUID.(EAX=0DH,ECX=i):EAX enumerates the number of bytes required for state component i.)
- **Compacted format.** Each state component i (i ≥ 2) is located at a byte offset from the base address of the XSAVE area based on the XCOMP_BV field in the XSAVE header:
  - If XCOMP_BV[i] = 0, state component i is not in the XSAVE area.
  - If XCOMP_BV[i] = 1, state component i is located at a byte offset location_i from the base address of the XSAVE area, where location_i is determined by the following items:
    - If XCOMP_BV[j] = 0 for every j, 2 ≤ j < i, location_i is 576. (This item applies if i is the first bit set in bits 62:2 of the XCOMP_BV; it implies that state component i is located at the beginning of the extended region.)
    - Otherwise, let j, 2 ≤ j < i, be the greatest value such that XCOMP_BV[j] = 1. Then location_i is determined by the following values: location_j, size_j, as enumerated in CPUID.(EAX=0DH,ECX=j):EAX; and the value of align_i, as enumerated in CPUID.(EAX=0DH,ECX=i):ECX[1]:
      - If align_i = 0, location_i = location_j + size_j. (This item implies that state component i is located immediately following the preceding state component whose bit is set in XCOMP_BV.)
      - If align_i = 1, location_i = ceiling(location_j + size_j, 64). (This item implies that state component i is located on the next 64-byte boundary following the preceding state component whose bit is set in XCOMP_BV.)
13.5 XSAVE-MANAGED STATE

The section provides details regarding how the XSAVE feature set interacts with the various XSAVE-managed state components.

Unless otherwise stated, the state pertaining to a particular state component is saved beginning at byte 0 of the section of the XSAVE area corresponding to that state component.

13.5.1 x87 State

Instructions in the XSAVE feature set can manage the same state of the x87 FPU execution environment (x87 state) that can be managed using the FXSAVE and FXRSTOR instructions. They organize all x87 state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the x87 state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 1:0, 3:2, 7:6. These are used for the x87 FPU Control Word (FCW), the x87 FPU Status Word (FSW), and the x87 FPU Opcode (FOP), respectively.
- Byte 4 is used for an abridged version of the x87 FPU Tag Word (FTW). The following items describe its usage:
  - For each \( j \), \( 0 \leq j \leq 7 \), XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 0 into bit \( j \) of byte 4 if x87 FPU data register ST\( j \) has an empty tag; otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 1 into bit \( j \) of byte 4.
  - For each \( j \), \( 0 \leq j \leq 7 \), XRSTOR and XRSTORS establish the tag value for x87 FPU data register ST\( j \) as follows. If bit \( j \) of byte 4 is 0, the tag for ST\( j \) in the tag register for that data register is marked empty (11B); otherwise, the x87 FPU sets the tag for ST\( j \) based on the value being loaded into that register (see below).
- Bytes 15:8 are used as follows:
  - If the instruction has no REX prefix, or if REX.W = 0:
    - Bytes 11:8 are used for bits 31:0 of the x87 FPU Instruction Pointer Offset (FIP).
    - If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 0, bytes 13:12 are used for x87 FPU Instruction Pointer Selector (FCS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H, and XRSTOR and XRSTORS ignore them.
    - Bytes 15:14 are not used.
  - If the instruction has a REX prefix with REX.W = 1, bytes 15:8 are used for the full 64 bits of FIP.
- Bytes 23:16 are used as follows:
  - If the instruction has no REX prefix, or if REX.W = 0:
    - Bytes 19:16 are used for bits 31:0 of the x87 FPU Data Pointer Offset (FDP).
    - If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 0, bytes 21:20 are used for x87 FPU Data Pointer Selector (FDS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H; and XRSTOR and XRSTORS ignore them.
    - Bytes 23:22 are not used.
  - If the instruction has a REX prefix with REX.W = 1, bytes 23:16 are used for the full 64 bits of FDP.
- Bytes 31:24 are used for SSE state (see Section 13.5.2).
- Bytes 159:32 are used for the registers ST0–ST7 (MM0–MM7). Each of the 8 register is allocated a 128-bit region, with the low 80 bits used for the register and the upper 48 bits unused.

x87 state is XSAVE-managed but the x87 FPU feature is not XSAVE-enabled. The XSAVE feature set can operate on x87 state only if the feature set is enabled (CR4.OSXSAVE = 1). Software can otherwise use x87 state even if the XSAVE feature set is not enabled.

1. The processor ensures that XCR0[0] is always 1.
13.5.2 SSE State

Instructions in the XSAVE feature set can manage the registers used by the streaming SIMD extensions (SSE state) just as the FXSAVE and FXRSTOR instructions do. They organize all SSE state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the SSE state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 23:0 are used for x87 state (see Section 13.5.1).
- Bytes 27:24 are used for the MXCSR register. XRSTOR and XRSTORS generate general-protection faults (#GP) in response to attempts to set any of the reserved bits of the MXCSR register.1
- Bytes 31:28 are used for the MXCSR_MASK value. XRSTOR and XRSTORS ignore this field.
- Bytes 159:32 are used for x87 state.
- Bytes 287:160 are used for the registers XMM0–XMM7.
- Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify these bytes; executions of XRSTOR and XRSTORS outside 64-bit mode do not update XMM8–XMM15. See Section 13.13.

SSE state is XSAVE-managed but the SSE feature is not XSAVE-enabled. The XSAVE feature set can operate on SSE state only if the feature set is enabled (CR4.OSXSAVE= 1) and has been configured to manage SSE state (XCR0[1] = 1). Software can otherwise use SSE state even if the XSAVE feature set is not enabled or has not been configured to manage SSE state.

13.5.3 AVX State

The register state used by the Intel® Advanced Vector Extensions (Intel AVX) comprises the MXCSR register and 16 256-bit vector registers called YMM0–YMM15. The low 128 bits of each register YMMi is identical to the SSE register XMMi. Thus, the new state register state added by Intel AVX comprises the upper 128 bits of the registers YMM0–YMM15. These 16 128-bit values are denoted YMM0_H–YMM15_H and are collectively called AVX state.

As noted in Section 13.1, the XSAVE feature set manages AVX state as user state component 2. Thus, AVX state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=2):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for AVX state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=2):EAX enumerates the size (in bytes) required for AVX state.

The XSAVE feature set partitions YMM0_H–YMM15_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 127:0 of the AVX-state section are for YMM0_H–YMM7_H. Bytes 255:128 are used for YMM8_H–YMM15_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 255:128; executions of XRSTOR and XRSTORS outside 64-bit mode do not update YMM8_H–YMM15_H. See Section 13.13. In general, bytes 16i+15:16i are used for YMMi_H (for 0 ≤ i ≤ 15).

AVX state is XSAVE-managed and the Intel AVX feature is XSAVE-enabled. The XSAVE feature set can operate on AVX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX state (XCR0[2] = 1). Intel AVX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX state.

13.5.4 MPX State

The register state used by the Intel® Memory Protection Extensions (MPX) comprises the 4 128-bit bounds registers BND0–BND3 (BNDREGS state); and the 64-bit user-mode configuration register BNDCFGU and the 64-bit MPX status register BNDCSTATUS (collectively, BNDCSR state). Together, these two user state components compose MPX state.

1. While MXCSR and MXCSR_MASK are part of SSE state, their treatment by the XSAVE feature set is not the same as that of the XMM registers. See Section 13.7 through Section 13.11 for details.
As noted in Section 13.1, the XSAVE feature set manages MPX state as state components 3-4. Thus, MPX state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **BNDREGS state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=3):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for BNDREGS state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=3):EAX enumerates the size (in bytes) required for BNDREGS state. The BNDREGS section is used for the 4 128-bit bound registers BND0–BND3, with bytes 16+i:15:16/i being used for BNDi.

- **BNDCSR state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=4):EBX enumerates the offset of the section of the extended region of the XSAVE area used for BNDCSR state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=4):EAX enumerates the size (in bytes) required for BNDCSR state. In the BNDCSR section, bytes 7:0 are used for BNDCFGU and bytes 15:8 are used for BNDSTATUS.

Both components of MPX state are XSAVE-managed and the Intel MPX feature is XSAVE-enabled. The XSAVE feature set can operate on MPX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage MPX state (XCR0[4:3] = 11b). Intel MPX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage MPX state.

### 13.5.5 AVX-512 State

The register state used by the Intel® Advanced Vector Extensions 512 (Intel AVX-512) comprises the MXCSR register, the 8 64-bit opmask registers k0–k7, and 32 512-bit vector registers called ZMM0–ZMM31. For each \( i \), \( 0 \leq i \leq 15 \), the low 256 bits of register ZMM\( i \) is identical to the Intel AVX register YMM\( i \). Thus, the new state register state added by Intel AVX-512 comprises the following user state components:

- The opmask registers, collectively called **opmask state**.
- The upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0_H–ZMM15_H and are collectively called **ZMM_Hi256 state**.
- The 16 512-bit registers ZMM16–ZMM31, collectively called **Hi16_ZMM state**.

Together, these three state components compose **AVX-512 state**.

As noted in Section 13.1, the XSAVE feature set manages AVX-512 state as state components 5–7. Thus, AVX-512 state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **Opmask state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=5):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for opmask state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes) required for opmask state. The opmask section is used for the 8 64-bit opmask registers k0–k7, with bytes 8+i:7:8+i being used for k\( i \).

- **ZMM_Hi256 state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=6):EBX enumerates the offset of the section of the extended region of the XSAVE area used for ZMM_Hi256 state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for ZMM_Hi256 state.
  The XSAVE feature set partitions ZMM0_H–ZMM15_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 255:0 of the ZMM_Hi256-state section are used for ZMM0_H–ZMM7_H. Bytes 511:256 are used for ZMM8_H–ZMM15_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVEES outside 64-bit mode do not modify bytes 511:256; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM8_H–ZMM15_H. See Section 13.13. In general, bytes 32+i:31:32+i are used for ZMM\( i \) (for \( 0 \leq i \leq 15 \)).

- **Hi16_ZMM state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=7):EBX enumerates the offset of the section of the extended region of the XSAVE area used for Hi16_ZMM state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=7):EAX enumerates the size (in bytes) required for Hi16_ZMM state.
The XSAVE feature set accesses Hi16_ZMM state only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify the Hi16_ZMM section; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM16–ZMM31. See Section 13.13. In general, bytes 64/(i-16)+63:64/(i-16) are used for ZMMi (for 16 ≤ i ≤ 31).

All three components of AVX-512 state are XSAVE-managed and the Intel AVX-512 feature is XSAVE-enabled. The XSAVE feature set can operate on AVX-512 state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX-512 state (XCR0[7:5] = 111b). Intel AVX-512 instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX-512 state.

### 13.5.6 PT State

The register state used by Intel Processor Trace (PT state) comprises the following 9 MSRs: IA32_RTIT_CTL, IA32_RTIT_OUTPUT_BASE, IA32_RTIT_OUTPUT_MASK_PTRS, IA32_RTIT_STATUS, IA32_RTIT_CR3_MATCH, IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B, IA32_RTIT_ADDR1_A, and IA32_RTIT_ADDR1_B.1

As noted in Section 13.1, the XSAVE feature set manages PT state as supervisor state component 8. Thus, PT state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=8):EAX enumerates the size (in bytes) required for PT state. The MSRs are each allocated 8 bytes in the state component in the order given above. Thus, IA32_RTIT_CTL is at byte offset 0, IA32_RTIT_OUTPUT_BASE at byte offset 8, etc. Any locations in the state component at or beyond byte offset 72 are reserved.

PT state is XSAVE-managed but Intel Processor Trace is not XSAVE-enabled. The XSAVE feature set can operate on PT state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PT state (IA32_XSS[8] = 1). Software can otherwise use Intel Processor Trace and access its MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage PT state.

The following items describe special treatment of PT state by the XSAVE and XRSTORS instructions:

- If XSAVE saves PT state, the instruction clears IA32_RTIT_CTL.TraceEn (bit 0) after saving the value of the IA32_RTIT_CTL MSR and before saving any other PT state. If XSAVE causes a fault or a VM exit, it restores IA32_RTIT_CTL.TraceEn to its original value.
- If XSAVE saves PT state, the instruction saves zeroes in the reserved portions of the state component.
- If XRSTORS would restore (or initialize) PT state and IA32_RTIT_CTL.TraceEn = 1, the instruction causes a general-protection exception (#GP) before modifying PT state.
- If XRSTORS causes an exception or a VM exit, it does so before any modification to IA32_RTIT_CTL.TraceEn (even if it has loaded other PT state).

### 13.5.7 PKRU State

The register state used by the protection-key feature (PKRU state) is the 32-bit PKRU register. As noted in Section 13.1, the XSAVE feature set manages PKRU state as user state component 9. Thus, PKRU state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=9):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for PKRU state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=9):EAX enumerates the size (in bytes) required for PKRU state. The XSAVE feature set uses bytes 3:0 of the PK-state section for the PKRU register.

PKRU state is XSAVE-managed but the protection-key feature is not XSAVE-enabled. The XSAVE feature set can operate on PKRU state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PKRU state (XCR0[9] = 1). Software can otherwise use protection keys and access PKRU state even if the XSAVE feature set is not enabled or has not been configured to manage PKRU state.

---

1. These MSRs might not be supported by every processor that supports Intel Processor Trace. Software can use the CPUID instruction to discover which are supported; see Section 33.3.1, “Detection of Intel Processor Trace and Capability Enumeration,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.
The value of the PKRU register determines the access rights for user-mode linear addresses. (See Section 4.6, “Access Rights,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.) The access rights that pertain to an execution of the XRSTOR and XRSTORS instructions are determined by the value of the register before the execution and not by any value that the execution might load into the PKRU register.

### 13.5.8 PASID State

The register state used by the ENQCMD instruction and process address space identifiers (PASID state) comprises the IA32_PASID MSR.

As noted in Section 13.1, the XSAVE feature set manages PASID state as supervisor state component 10. Thus, PASID state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=10):EAX enumerates the size (in bytes) required for PASID state. The IA32_PASID MSR is allocated 8 bytes at byte offset 0 in the state component.

PASID state is XSAVE-managed but the ENQCMD instruction and process address space identifiers are not XSAVE-enabled. The XSAVE feature set can operate on PASID state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PASID state (IA32_XSS[10] = 1). Software can otherwise use the ENQCMD instruction and process address space identifiers, and access the IA32_PASID MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage PASID state.

### 13.5.9 CET State

The register state used by Control-Flow Enforcement Technology (CET) comprises the two 64-bit MSRs (IA32_U_CET and IA32_PL3_SSP) that manage CET when CPL = 3 (CET_U state); and the three 64-bit MSRs (IA32_PL0_SSP–IA32_PL2_SSP) that manage CET when CPL < 3 (CET_S state). Together, these two supervisor state components compose CET state.1

As noted in Section 13.1, the XSAVE feature set manages CET state as supervisor state components 11–12. Thus, CET state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **CET_U state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=11):EAX enumerates the size (in bytes) required for CET_U state. The CET_U section is used for the 64-bit MSRs IA32_U_CET and IA32_PL3_SSP, with bytes 7:0 being used for IA32_U_CET and bytes 15:8 being used for IA32_PL3_SSP.

- **CET_S state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=12):EAX enumerates the size (in bytes) required for CET_S state. The CET_S section is used for the three 64-bit MSRs IA32_PL0_SSP–IA32_PL2_SSP, with bytes 8i+7:8i being used for IA32_PLi_SSP.

The two components of CET state are XSAVE-managed and CET is not XSAVE-enabled. The XSAVE feature set can operate on CET_U state (respectively, CET_S state) only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage CET_U state (respectively, CET_S state) by setting IA32_XSS[11] (respectively, IA32_XSS[12]). Software can otherwise use CET and access the CET MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage CET state.

### 13.5.10 HDC State

The register state used by hardware duty cycling (HDC state) comprises the IA32_PM_CTL1 MSR.

As noted in Section 13.1, the XSAVE feature set manages HDC state as supervisor state component 13. Thus, HDC state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=13):EAX enumerates the size (in bytes) required for HDC state. The IA32_PM_CTL1 MSR is allocated 8 bytes at byte offset 0 in the state component.

---

1. The IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs also control CET when CPL < 3. However, they are not managed by the XSAVE feature set and are thus not considered in this chapter.
HDC state is XSAVE-managed but hardware duty cycling is not XSAVE-enabled. The XSAVE feature set can operate on HDC state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HDC state (IA32_XSS[13] = 1). Software can otherwise use hardware duty cycling and access the IA32_PM_CTL1 MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HDC state.

13.5.11 UINTR State

The register state used by user interrupts (UINTR state) comprises 48 bytes in memory with the following layout:

- Bytes 7:0 are for the IA32_UINTR_HANDLER MSR.
- Bytes 15:8 are for the IA32_UINTR_STACKADJUST MSR.
- Bytes 23:16 are for the IA32_UINTR_MISC MSR with exception of the last bit (bit 7 of byte 23), which is used for UIF. (Because UIF is not part of the IA32_UINTR_MISC MSR, software that reads a value from bytes 23:16 should clear bit 63 of that 64-bit value before attempting to write it to the IA32_UINTR_MISC MSR.)
- Bytes 31:24 are for the IA32_UINTR_PD MSR.
- Bytes 39:32 are for the IA32_UINTR_RR MSR.
- Bytes 47:40 are for the IA32_UINTR_TT MSR.

UINTR state is XSAVE-managed but user interrupts are not XSAVE-enabled. The XSAVE feature set can operate on UINTR state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage UINTR state (IA32_XSS[14] = 1). Software can otherwise use user interrupts and access the MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage UINTR state.

The management of the UINTR state component by XSAVES follows the architecture of the XSAVE feature set. The following items identify points that are specific to saving the UINTR state component:

- XSAVES writes the user-interrupt registers to the user-interrupt state component using the format specified above.
- XSAVES stores zeros to bits and bytes identified above as reserved.
- The values saved for the IA32_UINTR_HANDLER, IA32_UINTR_STACKADJUST, IA32_UINTR_PD, and IA32_UINTR_TT MSRs are always canonical relative to the maximum linear-address width enumerated by CPUID.
- After saving the user-interrupt state component, XSAVES clears UINV. (UINV is IA32_UINTR_MISC[39:32]; XSAVES does not modify the remainder of that MSR.)

The management of the user-interrupt state component by XRSTORS follows the architecture of the XSAVE feature set. The following items identify points that are specific to restoring the user-interrupt state component:

- Before restoring the user-interrupt state component, XRSTORS verifies that UINV is 0. If it is not, XRSTORS causes a general-protection fault (#GP) before loading any part of the user-interrupt state component. (UINV is IA32_UINTR_MISC[39:32]; XRSTORS does not check the contents of the remainder of that MSR.)
- If the instruction mask and XSAVE area used by XRSTORS indicates that the user-interrupt state component should be loaded from the XSAVE area, XRSTORS reads the user-interrupt registers from the XSAVE area using the format identified above. The values read cause a general-protection fault (#GP) in any of the following cases:
  - If the value to be loaded into any one of the IA32_UINTR_HANDLER, IA32_UINTR_STACKADJUST, IA32_UINTR_PD, or IA32_UINTR_TT MSRs is not canonical relative to the maximum linear-address width enumerated by CPUID.
  - If the value to be loaded into the IA32_UINTR_MISC MSR sets any of bits 62:40. These bits are reserved in the MSR. (Bit 63 is also reserved in the MSR, but the XSAVE feature set uses bit 63 of this value for UIF.)

1. They might not be canonical relative to the current paging mode if it supports only smaller linear addresses.
— If the value to be loaded into the IA32_UINTR_PD MSR sets any of bits 5:0. These bits are reserved in the MSR.
— If the value to be loaded into the IA32_UINTR_TT MSR sets any of bits 3:1. These bits are reserved in the MSR.

• If XRSTORS causes a fault or a VM exit after loading any part of the user-interrupt state component, XRSTORS clears UINV before delivering the fault or VM exit. (Other elements of user-interrupt state, including other parts of the IA32_UINTR_MISC MSR, may retain the values that were loaded by XRSTORS.)

• After an execution of XRSTORS that loads the user-interrupt state component, the logical processor recognizes a pending user interrupt if and only if some bit is set in the IA32_UINTR_RR MSR (see Section 7.4.1 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

13.5.12 LBR State

The register state used by last-branch records (LBR state) comprises 101 MSRs organized as follows:
IA32_LBR_CTL; IA32_LBR_DEPTH; IA32_LER_FROM_IP; IA32_LER_TO_IP; IA32_LER_INFO; and 32 triples of MSRs, IA32_LBR_i_FROM_IP, IA32_LBR_i_TO_IP, IA32_LBR_i_INFO, for each value of \( i \), \( 0 \leq i \leq 31 \).

As noted in Section 13.1, the XSAVE feature set manages LBR state as supervisor state component 15. Thus, LBR state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=ODH,ECX=15):EAX enumerates the size (in bytes) required for LBR state. The IA32_LBR_CTL MSR is allocated 8 bytes at byte offset 0 in the state component. The remaining MSRs are each allocated 8 bytes in the state component in the order given above. Thus, IA32_LBR_DEPTH is at byte offset 8, \ldots, IA32_LBR_0_FROM_IP at byte offset 40, IA32_LBR_0_TO_IP at byte offset 48, IA32_LBR_0_INFO at byte offset 56, IA32_LBR_1_FROM_IP at byte offset 64, \ldots, and IA32_LBR_31_INFO at byte offset 800. Any locations in the state component at or beyond byte offset 808 are reserved.

LBR state is XSAVE-managed but LBRs are not XSAVE-enabled. The XSAVE feature set can operate on LBR state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage LBR state (IA32_XSS[15] = 1). Software can otherwise use LBRs and access the MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage LBR state.

The following items describe special treatment of LBR state by the XSAVES and XRSTORS instructions:

• If XSAVES would save LBR state and that state is not in its initial configuration (see Section 13.6), the instruction always saves IA32_LBR_CTL, IA32_LBR_DEPTH, IA32_LER_FROM_IP, IA32_LER_TO_IP, and IA32_LER_INFO. It saves the triples IA32_LBR_i_FROM_IP, IA32_LBR_i_TO_IP, IA32_LBR_i_INFO, for each value of \( i \), \( 0 \leq i < D \), where \( D \) is the value of IA32_LBR_DEPTH. It will not save the values of the remaining triples, although it may access the corresponding fields in the XSAVE area.

• If XSAVES would save LBR state and that state is in its initial configuration, the instruction does not save any LBR state and will not access that component of the XSAVE area.

• If XRSTORS would initialize LBR state, IA32_LBR_DEPTH is not modified and zero is written to the other MSRs that compose LBR state.

• If XRSTORS would restore LBR state, behavior depends on the current value of IA32_LBR_DEPTH and the value of corresponding field in the XSAVE area:
  — If the current value of IA32_LBR_DEPTH equals the value of corresponding field in the XSAVE area, the instruction restores IA32_LBR_CTL, IA32_LER_FROM_IP, IA32_LER_TO_IP, IA32_LER_INFO, and the triples IA32_LBR_i_FROM_IP, IA32_LBR_i_TO_IP, IA32_LBR_i_INFO, for each value of \( i \), \( 0 \leq i < D \), where \( D \) is the value of IA32_LBR_DEPTH. It will not restore the values of the remaining triples, although it may access the corresponding fields in the XSAVE area.
  — If the IA32_LBR_DEPTH field in the XSAVE area sets any reserved bits, the instruction causes a general-protection exception (#GP).
  — If neither of the previous items apply, the instruction restores IA32_LBR_CTL, IA32_LER_FROM_IP, IA32_LER_TO_IP, and IA32_LER_INFO, but it writes zero to the triples IA32_LBR_i_FROM_IP, IA32_LBR_i_TO_IP, IA32_LBR_i_INFO, for each value of \( i \), \( 0 \leq i \leq 31 \). Such an execution does not modify XINUSE[15] (see Section 13.6 and Section 13.12).
13.5.13 HWP State

The register state used by hardware P-states (HWP state) comprises the IA32_HWP_REQUEST MSR.

As noted in Section 13.1, the XSAVE feature set manages HWP state as supervisor state component 16. Thus, HWP state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=16):EAX enumerates the size (in bytes) required for HWP state. The IA32_HWP_REQUEST MSR is allocated 8 bytes at byte offset 0 in the state component.

HWP state is XSAVE-managed but the hardware P-states feature is not XSAVE-enabled. The XSAVE feature set can operate on HWP state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage HWP state (IA32_XSS[16] = 1). Software can otherwise use hardware P-states and access the IA32_HWP_REQUEST MSR (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage HWP state.

13.5.14 AMX State

The register state used by the Intel® Advanced Matrix Extensions (Intel AMX) comprises two state components, TILECFG and TILEDATA. Together, these two state components compose AMX state.

As noted in Section 13.1, the XSAVE feature set manages AMX state as state components 17–18. Thus, AMX state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **TILECFG state.**
  As noted in Section 13.1, the XSAVE feature set manages TILECFG state as user state component 17. Thus, TILECFG state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=17):EAX enumerates the size (in bytes) required for TILECFG state.

- **TILEDATA state.**
  As noted in Section 13.1, the XSAVE feature set manages TILEDATA state as user state component 18. Thus, TILEDATA state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=18):EAX enumerates the size (in bytes) required for TILEDATA state.

Both components of AMX state are XSAVE-managed, and the AMX feature is XSAVE-enabled. The XSAVE feature set can operate on AMX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AMX state (XCR0[18:17] = 11b). Intel AMX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AMX state.

The following items describe special treatment of TILECFG and TILEDATA by the XSAVE feature set:

- **Loading of TILECFG and TILEDATA by XRSTOR and XRSTORS:**
  - While the LDTILECFG instruction generates a general-protection fault (#GP) if it would load the TILECFG register with an unsupported value executions of XRSTOR and XRSTORS do not do so. Instead, they initialize the register (resulting in TILES_CONFIGURED = 0).
  - While executions of LDTILECFG initialize TILEDATA, executions of XRSTOR and XRSTORS do not modify TILEDATA unless loading it from memory.
  - While the value of the TILECFG register can limit how Intel AMX instructions access TILEDATA, such limitations do not apply to XRSTOR and XRSTORS. An execution of either of those instructions loads all 8 KBytes of TILEDATA regardless of the value in the TILECFG register (or the value that the instruction may be loading into that register).

- **Saving of TILEDATA by XSAVE, XSAVEC, XSAVEOPT, and XSAVES:**
  - While the value of the TILECFG register can limit how Intel AMX instructions access TILEDATA, such limitations do not apply to XSAVE, XSAVEC, XSAVEOPT, and XSAVES. An execution of any of those instructions saves all 8 KBytes of TILEDATA regardless of the value in the TILECFG register.
13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE

The XSAVEOPT, XSAVEC, and XSAVES instructions use two optimizations to reduce the amount of data that they write to memory. They avoid writing data for any state component known to be in its initial configuration (the init optimization). In addition, if either XSAVEOPT or XSAVES is using the same XSAVE area as that used by the most recent execution of XRSTOR or XRSTORS, it may avoid writing data for any state component whose configuration is known not to have been modified since then (the modified optimization). (XSAVE does not use these optimizations, and XSAVEC does not use the modified optimization.) The operation of XSAVEOPT, XSAVEC, and XSAVES are described in more detail in Section 13.9 through Section 13.11.

A processor can support the init and modified optimizations with special hardware that tracks the state components that might benefit from those optimizations. Other implementations might not include such hardware; such a processor would always consider each such state component as not in its initial configuration and as modified since the last execution of XRSTOR or XRSTORS.

The following notation describes the state of the init and modified optimizations:

- XINUSE denotes the state-component bitmap corresponding to the init optimization. If XINUSE[i] = 0, state component i is known to be in its initial configuration; otherwise XINUSE[i] = 1. It is possible for XINUSE[i] to be 1 even when state component i is in its initial configuration. On a processor that does not support the init optimization, XINUSE[i] is always 1 for every value of i.

  Executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. Such an execution of XGETBV always sets EAX[1] to 1 if XCR0[1] = 1 and MXCSR does not have its RESET value of 1F80H. Section 13.2 explains how software can determine whether a processor supports this use of XGETBV.

- XMODIFIED denotes the state-component bitmap corresponding to the modified optimization. If XMODIFIED[i] = 0, state component i is known not to have been modified since the most recent execution of XRSTOR or XRSTORS; otherwise XMODIFIED[i] = 1. It is possible for XMODIFIED[i] to be 1 even when state component i has not been modified since the most recent execution of XRSTOR or XRSTORS. On a processor that does not support the modified optimization, XMODIFIED[i] is always 1 for every value of i.

A processor that implements the modified optimization saves information about the most recent execution of XRSTOR or XRSTORS in a quantity called XRSTOR_INFO, a 4-tuple containing the following: (1) the CPL; (2) whether the logical processor was in VMX non-root operation; (3) the linear address of the XSAVE area; and (4) the XCOMP_BV field in the XSAVE area. An execution of XSAVEOPT or XSAVES uses the modified optimization only if that execution corresponds to XRSTOR_INFO on these four parameters.

This mechanism implies that, depending on details of the operating system, the processor might determine that an execution of XSAVEOPT by one user application corresponds to an earlier execution of XRSTOR by a different application. For this reason, Intel recommends the application software not use the XSAVEOPT instruction.

The following items specify the initial configuration each state component (for the purposes of defining the XINUSE bitmap):

- **x87 state.** x87 state is in its initial configuration if the following all hold: FCW is 037FH; FSW is 0000H; FTW is FFFFH; FCS and FDS are each 0000H; FIP and FDP are each 00000000_00000000H; each of ST0–ST7 is 0000_00000000_00000000H.

- **SSE state.** In 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM15 is 0. Outside 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM7 is 0. XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update XMM8–XMM15. (See Section 13.13.)

- **AVX state.** In 64-bit mode, AVX state is in its initial configuration if each of YMM0_H–YMM15_H is 0. Outside 64-bit mode, AVX state is in its initial configuration if each of YMM0_H–YMM7_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update YMM8_H–YMM15_H. (See Section 13.13.)

- **BNDREGS state.** BNDREGS state is in its initial configuration if the value of each of BND0–BND3 is 0.

- **BNDCSR state.** BNDCSR state is in its initial configuration if BNDCFGU and BNDCSR each has value 0.

- **Opmask state.** Opmask state is in its initial configuration if each of the opmask registers k0–k7 is 0.

- **ZMM_Hi256 state.** In 64-bit mode, ZMM_Hi256 state is in its initial configuration if each of ZMM0_H–ZMM15_H is 0. Outside 64-bit mode, ZMM_Hi256 state is in its initial configuration if each of ZMM0_H–ZMM7_H is 0.
is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM8_H–ZMM15_H. (See Section 13.13.)

- **Hi16_ZMM state.** In 64-bit mode, Hi16_ZMM state is in its initial configuration if each of ZMM16–ZMM31 is 0. Outside 64-bit mode, Hi16_ZMM state is always in its initial configuration. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM31–ZMM31. (See Section 13.13.)

- **PT state.** PT state is in its initial configuration if each of the 9 MSRs is 0.

- **PKRU state.** PKRU state is in its initial configuration if the value of the PKRU is 0.

- **PASID state.** PASID state is in its initial configuration if the value of the IA32_PASID MSR is 0.

- **CET_U state.** CET_U state is in its initial configuration if both of the MSRs are 0.

- **CET_S state.** CET_S state is in its initial configuration if each of the three MSRs is 0.

- **HDC state.** HDC state is in its initial configuration if the value of the IA32_PM_CTL1 MSR is 1.

- **UINTR state.** UINTR state is in its initial configuration if all user-interrupt registers (including UIF) are zero.

- **LBR state.** LBR state is in its initial configuration if the value of each of the MSRs is 0, with the exception of IA32_LBR_DEPTH. XINUSE[15] does not pertain to IA32_LBR_DEPTH.

- **HWP state.** HWP state is in its initial configuration if the value of the IA32_HWP_REQUEST MSR is 8000FF01H.

- **AMX state.** AMX state is in its initial configuration if the TILECFG register is zero and all tile data are zero.

### 13.7 OPERATION OF XSAVE

The XSAVE instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask.** The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap** (RFBM) of the user state components to be saved.

The following conditions cause execution of the XSAVE instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.  

If none of these conditions cause a fault, execution of XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting XSTATE_BV[i] (0 ≤ i ≤ 63) as follows:

- If RFBM[i] = 0, XSTATE_BV[i] is not changed.
- If RFBM[i] = 1, XSTATE_BV[i] is set to the value of XINUSE[i]. Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component i is in its initial configuration, XINUSE[i] may be either 0 or 1, and XSTATE_BV[i] may be written with either 0 or 1.
  - XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. Thus, XSTATE_BV[1] may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If state component i is not in its initial configuration, XINUSE[i] = 1 and XSTATE_BV[i] is written with 1.

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVE instruction does not write any part of the XSAVE header other than the XSTATE_BV field; in particular, it does not write to the XCOMP_BV field.

Execution of XSAVE saves into the XSAVE area those state components corresponding to bits that are set in RFBM. State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i, 2 ≤ i ≤ 62, is located in the extended region; the XSAVE instruction always uses the standard format for the extended region (see Section 13.4.3).

---

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.
The MXCSR register and MXCSR_MASK are part of SSE state (see Section 13.5.2) and are thus associated with RFBM[1]. However, the XSAVE instruction also saves these values when RFBM[2] = 1 (even if RFBM[1] = 0). See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

13.8 OPERATION OF XRSTOR

The XRSTOR instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the instruction mask. The logical-AND of XCR0 and the instruction mask is the requested-feature bitmap (RFBM) of the user state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

• If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
• If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
• If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.¹

After checking for these faults, the XRSTOR instruction reads the XCOMP_BV field in the XSAVE area’s XSAVE header (see Section 13.4.2). If XCOMP_BV[63] = 0, the standard form of XRSTOR is executed (see Section 13.8.1); otherwise, the compacted form of XRSTOR is executed (see Section 13.8.2).²

See Section 13.2 for details of how to determine whether the compacted form of XRSTOR is supported.

13.8.1 Standard Form of XRSTOR

The standard form of XRSTOR performs additional fault checking. Either of the following conditions causes a general-protection exception (#GP):

• The XSTATE_BV field of the XSAVE header sets a bit that is not set in XCR0.
• Bytes 23:8 of the XSAVE header are not all 0 (this implies that all bits in XCOMP_BV are 0).³

If none of these conditions cause a fault, the processor updates each state component i for which RFBM[i] = 1. XRSTOR updates state component i based on the value of bit i in the XSTATE_BV field of the XSAVE header:

• If XSTATE_BV[i] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

The initial configuration of state component 1 pertains only to the XMM registers and not to MXCSR. See below for the treatment of MXCSR:

• If XSTATE_BV[i] = 1, the state component is loaded with data from the XSAVE area. See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i, 2 ≤ i ≤ 62, is located in the extended region; the standard form of XRSTOR uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register is part of state component 1, SSE state (see Section 13.5.2). However, the standard form of XRSTOR loads the MXCSR register from memory whenever the RFBM[1] (SSE) or RFBM[2] (AVX) is set, regardless

---

¹ If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.
² If the processor does not support the compacted form of XRSTOR, it may execute the standard form of XRSTOR without first reading the XCOMP_BV field. A processor supports the compacted form of XRSTOR only if it enumerates CPUID.(EAX=0DH,ECX=1):EAX[1] as 1.
³ Bytes 63:24 of the XSAVE header are also reserved. Software should ensure that bytes 63:16 of the XSAVE header are all 0 in any XSAVE area. (Bytes 15:8 should also be 0 if the XSAVE area is to be used on a processor that does not support the compaction extensions to the XSAVE feature set.)
of the values of XSTATE_BV[1] and XSTATE_BV[2]. The standard form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

### 13.8.2 Compacted Form of XRSTOR

The compacted form of XRSTOR performs additional fault checking. Any of the following conditions causes a #GP:

- The XCOMP_BV field of the XSAVE header sets a bit in the range 62:0 that is not set in XCR0.
- The XSTATE_BV field of the XSAVE header sets a bit (including bit 63) that is not set in XCOMP_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component \( i \) for which \( RFBM[i] = 1 \). XRSTOR updates state component \( i \) based on the value of bit \( i \) in the XSTATE_BV field of the XSAVE header:

- If \( XSTATE_BV[i] = 0 \), the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.
  
  If \( XSTATE_BV[1] = 0 \), the compacted form XRSTOR initializes MXCSR to 1F80H. (This differs from the standard form of XRSTOR, which loads MXCSR from the XSAVE area whenever either \( RFBM[1] \) or \( RFBM[2] \) is set.) State component \( i \) is set to its initial configuration as indicated above if \( RFBM[i] = 1 \) and \( XSTATE_BV[i] = 0 \) — even if \( XCOMP_BV[i] = 0 \). This is true for all values of \( i \), including 0 (x87 state) and 1 (SSE state).

- If \( XSTATE_BV[i] = 1 \), the state component is loaded with data from the XSAVE area.\(^1\) See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component \( i \), \( 2 \leq i \leq 62 \), is located in the extended region; the compacted form of the XRSTOR instruction uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if \( RFBM[1] = XSTATE_BV[1] = 1 \). The compacted form of XRSTOR does not consider \( RFBM[2] \) (AVX) when determining whether to update MXCSR. (This is a difference from the standard form of XRSTOR.) The compacted form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

### 13.8.3 XRSTOR and the Init and Modified Optimizations

Execution of the XRSTOR instruction causes the processor to update its tracking for the init and modified optimizations (see Section 13.6). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  
  - If \( RFBM[i] = 0 \), XINUSE\( [i] \) is not changed.
  
  - If \( RFBM[i] = 1 \) and \( XSTATE_BV[i] = 0 \), state component \( i \) may be tracked as init; XINUSE\( [i] \) may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the init optimization for state component \( i \); a processor that does not do so implicitly maintains XINUSE\( [i] = 1 \) at all times.)
  
  - If \( RFBM[i] = 1 \) and \( XSTATE_BV[i] = 1 \), state component \( i \) is tracked as not init; XINUSE\( [i] \) is set to 1.

- The processor updates its tracking for the modified optimization and records information about the XRSTOR execution for future interaction with the XSAVEOPT and XSAVES instructions (see Section 13.9 and Section 13.11) as follows:
  
  - If \( RFBM[i] = 0 \), state component \( i \) is tracked as modified; XMODIFIED\( [i] \) is set to 1.
  
  - If \( RFBM[i] = 1 \), state component \( i \) may be tracked as unmodified; XMODIFIED\( [i] \) may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the modified optimization for state component \( i \); a processor that does not do so implicitly maintains XMODIFIED\( [i] = 1 \) at all times.)

---

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and \( XSTATE_BV[i] = 1 \), then \( XCOMP_BV[i] = 1 \) is also 1.
— XRSTOR_INFO is set to the 4-tuple \((w, x, y, z)\), where \(w\) is the CPL (0); \(x\) is 1 if the logical processor is in VMX non-root operation and 0 otherwise; \(y\) is the linear address of the XSAVE area; and \(z\) is XCOMP_BV. In particular, the standard form of XRSTOR always sets \(z\) to all zeroes, while the compacted form of XRSTORS never does so (because it sets at least bit 63 to 1).

Note that, if RFBM is entirely zero (e.g., because the instruction mask in EDX:EAX is zero), no state components are modified, the XINUSE bitmap is not modified, and all bits are set in the XMODIFIED bitmap. Thus, if EDX:EAX was zero for the most recent execution of XRSTOR, an execution of XSAVEOPT or XSAVES will identify all state components as modified and will thus not use the modified optimization.

### 13.9 OPERATION OF XSAVEOPT

The operation of XSAVEOPT is similar to that of XSAVE. Unlike XSAVE, XSAVEOPT uses the init optimization (by which it may omit saving state components that are in their initial configuration) and the modified optimization (by which it may omit saving state components that have not been modified since the last execution of XRSTOR); see Section 13.6. See Section 13.2 for details of how to determine whether XSAVEOPT is supported.

The XSAVEOPT instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the instruction mask. The logical (bitwise) AND of XCR0 and the instruction mask is the requested-feature bitmap (RFBM) of the user state components to be saved.

The following conditions cause execution of the XSAVEOPT instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.\(^1\)

If none of these conditions cause a fault, execution of XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting XSTATE_BV\([i]\) (\(0 \leq i \leq 63\)) as follows:

- If RFBM\([i]\) = 0, XSTATE_BV\([i]\) is not changed.
- If RFBM\([i]\) = 1, XSTATE_BV\([i]\) is set to the value of XINUSE\([i]\). Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If the state component is in its initial configuration, XINUSE\([i]\) may be either 0 or 1, and XSTATE_BV\([i]\) may be written with either 0 or 1.
  - XINUSE\([1]\) pertains only to the state of the XMM registers and not to MXCSR. Thus, XSTATE_BV\([1]\) may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If the state component is not in its initial configuration, XSTATE_BV\([i]\) is written with 1.

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEOPT instruction does not write any part of the XSAVE header other than the XSTATE_BV field; in particular, it does not write to the XCOMP_BV field.

Execution of XSAVEOPT saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component \(i, 2 \leq i \leq 62\), is located in the extended region; the XSAVEOPT instruction always uses the standard format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEOPT performs two optimizations that reduce the amount of data written to memory:

---

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.
• **Init optimization.**
  If XINUSE\[i\] = 0, state component \(i\) is not saved to the XSAVE area (even if RFBM\[i\] = 1). (See below for exceptions made for MXCSR.)

• **Modified optimization.**
  Each execution of XRSTOR and XRSTORS establishes XRSTOR_INFO as a 4-tuple \(\langle w, x, y, z \rangle\) (see Section 13.8.3 and Section 13.12). Execution of XSAVEOPT uses the modified optimization only if the following all hold for the current value of XRSTOR_INFO:
  - \(w = CPL\);
  - \(x = 1\) if and only if the logical processor is in VMX non-root operation;
  - \(y\) is the linear address of the XSAVE area being used by XSAVEOPT; and
  - \(z = 00000000_00000000H\). (This last item implies that XSAVEOPT does not use the modified optimization if the last execution of XRSTOR used the compacted form, or if an execution of XRSTORS followed the last execution of XRSTOR.)

If XSAVEOPT uses the modified optimization and XMODIFIED\[i\] = 0 (see Section 13.6), state component \(i\) is not saved to the XSAVE area.

(In practice, the benefit of the modified optimization for state component \(i\) depends on how the processor is tracking state component \(i\); see Section 13.6. Limitations on the tracking ability may result in state component \(i\) being saved even though it is in the same configuration that was loaded by the previous execution of XRSTOR.)

Depending on details of the operating system, an execution of XSAVEOPT by a user application might use the modified optimization when the most recent execution of XRSTOR was by a different application. Because of this, Intel recommends the application software not use the XSAVEOPT instruction.

The MXCSR register and MXCSR_MASK are part of SSE state (see Section 13.5.2) and are thus associated with bit 1 of RFBM. However, the XSAVEOPT instruction also saves these values when RFBM\[2\] = 1 (even if RFBM\[1\] = 0). The init and modified optimizations do not apply to the MXCSR register and MXCSR_MASK.

### 13.10 OPERATION OF XSAVEC

The operation of XSAVEC is similar to that of XSAVE. Two main differences are (1) XSAVEC uses the compacted format for the extended region of the XSAVE area; and (2) XSAVEC uses the init optimization (see Section 13.6). Unlike XSAVEOPT, XSAVEC does not use the modified optimization. See Section 13.2 for details of how to determine whether XSAVEC is supported.

The XSAVEC instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the *instruction mask*. The logical (bitwise) AND of XCR0 and the instruction mask is the *requested-feature bitmap* (RFBM) of the user state components to be saved.

The following conditions cause execution of the XSAVEC instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.¹

If none of these conditions cause a fault, execution of XSAVEC writes the XSTATE_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE_BV\[i\] (0 ≤ \(i\) ≤ 63) as follows:\²

- If RFBM\[i\] = 0, XSTATE_BV\[i\] is written as 0.
- If RFBM\[i\] = 1, XSTATE_BV\[i\] is set to the value of XINUSE\[i\] (see below for an exception made for XSTATE_BV\[1\]).

Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:

- If state component \(i\) is in its initial configuration, XSTATE_BV\[i\] may be written with either 0 or 1.

¹. If CR0.AM = 1, CPL = 3, and EFLAGS.AC = 1, an alignment-check exception (#AC) may occur instead of #GP.
². Unlike the XSAVE and XSAVEOPT instructions, the XSAVEC instruction does not read the XSTATE_BV field of the XSAVE header.
— If state component \( i \) is not in its initial configuration, \( XSTATE_{BV}[i] \) is written with 1.

\( XINUSE[1] \) pertains only to the state of the XMM registers and not to MXCSR. However, if \( RFBM[1] = 1 \) and MXCSR does not have the value 1F80H, XSAVEC writes \( XSTATE_{BV}[1] \) as 1 even if \( XINUSE[1] = 0 \).

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEC instruction sets bit 63 of the XCOMP_BV field of the XSAVE header while writing \( RFBM[62:0] \) to \( XCOMP_BV[62:0] \). The XSAVEC instruction does not write any part of the XSAVE header other than the \( XSTATE_{BV} \) and \( XCOMP_BV \) fields.

Execution of XSAVE saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the init optimization described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component \( i, 2 \leq i \leq 62 \), is located in the extended region; the XSAVEC instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEC performs the init optimization to reduce the amount of data written to memory. If \( XINUSE[i] = 0 \), state component \( i \) is not saved to the XSAVE area (even if \( RFBM[i] = 1 \)). However, if \( RFBM[1] = 1 \) and MXCSR does not have the value 1F80H, XSAVEC saves all of state component 1 (SSE — including the XMM registers) even if \( XINUSE[1] = 0 \). Unlike the XSAVE instruction, \( RFBM[2] \) does not determine whether XSAVEC saves MXCSR and MXCSR_MASK.

### 13.11 OPERATION OF XSAVES

The operation of XSAVES is similar to that of XSAVEC. The main differences are (1) XSAVES can be executed only if \( CPL = 0 \); (2) XSAVES can operate on the state components whose bits are set in XCR0 | IA32_XSS and can thus operate on supervisor state components; and (3) XSAVES uses the modified optimization (see Section 13.6). See Section 13.2 for details of how to determine whether XSAVES is supported.

The XSAVES instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. EDX:EAX & (XCR0 | IA32_XSS) (the logical AND the instruction mask with the logical OR of XCR0 and IA32_XSS) is the **requested-feature bitmap** (RFBM) of the state components to be saved.

The following conditions cause execution of the XSAVES instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If CPL > 0 or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.

If none of these conditions cause a fault, execution of XSAVES writes the \( XSTATE_{BV} \) field of the XSAVE header (see Section 13.4.2), setting \( XSTATE_{BV}[i] \) (\( 0 \leq i \leq 63 \)) as follows:

- If \( RFBM[i] = 0 \), \( XSTATE_{BV}[i] \) is written as 0.
- If \( RFBM[i] = 1 \), \( XSTATE_{BV}[i] \) is set to the value of \( XINUSE[i] \) (see below for an exception made for \( XSTATE_{BV}[1] \)). Section 13.6 defines \( XINUSE \) to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component \( i \) is in its initial configuration, \( XSTATE_{BV}[i] \) may be written with either 0 or 1.
  - If state component \( i \) is not in its initial configuration, \( XSTATE_{BV}[i] \) is written with 1. \( XINUSE[1] \) pertains only to the state of the XMM registers and not to MXCSR. However, if \( RFBM[1] = 1 \) and MXCSR does not have the value 1F80H, XSAVES writes \( XSTATE_{BV}[1] \) as 1 even if \( XINUSE[1] = 0 \).

(As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)
The XSAVES instructions sets bit 63 of the XCOMP_BV field of the XSAVE header while writing RFBM[62:0] to XCOMP_BV[62:0]. The XSAVES instruction does not write any part of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.

Execution of XSAVES saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component \( i, 2 \leq i \leq 62 \), is located in the extended region; the XSAVES instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6, Section 13.5.11, Section 13.5.12, and Section 13.5.14 for special treatment by XSAVES of PT state, UINTR state, LBR state, and AMX state, respectively. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVES performs the init optimization to reduce the amount of data written to memory. If XINUSE[\( i \)] = 0, state component \( i \) is not saved to the XSAVE area (even if RFBM[\( i \)] = 1). However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVES saves all of state component 1 (SSE — including the XMM registers) even if XINUSE[1] = 0.

Like XSAVEOPT, XSAVES may perform the modified optimization. Each execution of XRSTOR and XRSTORS establishes XRSTOR_INFO as a 4-tuple \( \langle w, x, y, z \rangle \) (see Section 13.8.3 and Section 13.12). Execution of XSAVES uses the modified optimization only if the following all hold:

- \( w = CPL \);
- \( x = 1 \) if and only if the logical processor is in VMX non-root operation;
- \( y \) is the linear address of the XSAVE area being used by XSAVEOPT; and
- \( z[63] \) is 1 and \( z[62:0] = RFBM[62:0] \). (This last item implies that XSAVES does not use the modified optimization if the last execution of XRSTOR used the standard form and followed the last execution of XRSTORS.)

If XSAVES uses the modified optimization and XMODIFIED[\( i \)] = 0 (see Section 13.6), state component \( i \) is not saved to the XSAVE area.

### 13.12 OPERATION OF XRSTORS

The operation of XRSTORS is similar to that of XRSTOR. Three main differences are (1) XRSTORS can be executed only if CPL = 0; (2) XRSTORS can operate on the state components whose bits are set in XCR0 | IA32_XSS and can thus operate on supervisor state components; and (3) XRSTORS has only a compacted form (no standard form; see Section 13.8). See Section 13.2 for details of how to determine whether XRSTORS is supported.

The XRSTORS instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the instruction mask. EDX:EAX & (XCR0 | IA32_XSS) (the logical AND the instruction mask with the logical OR of XCR0 and IA32_XSS) is the requested-feature bitmap (RFBM) of the state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If CPL > 0 or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.

After checking for these faults, the XRSTORS instruction reads the first 64 bytes of the XSAVE header, including the XSTATE_BV and XCOMP_BV fields (see Section 13.4.2). A #GP occurs if any of the following conditions hold for the values read:

- \( XCOMP_BV[63] = 0 \).
- \( XCOMP_BV \) sets a bit in the range 62:0 that is not set in XCR0 | IA32_XSS.
- \( XSTATE_BV \) sets a bit (including bit 63) that is not set in XCOMP_BV.
- Bytes 63:16 of the XSAVE header are not all 0.
If none of these conditions cause a fault, the processor updates each state component \( i \) for which RFBM\( [i] \) = 1. XRSTORS updates state component \( i \) based on the value of bit \( i \) in the XSTATE_BV field of the XSAVE header:

- If XSTATE_BV\( [i] \) = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component. If XSTATE_BV\( [1] \) = 0, XRSTORS initializes MXCSR to 1F80H.
  
  State component \( i \) is set to its initial configuration as indicated above if RFBM\( [i] \) = 1 and XSTATE_BV\( [i] \) = 0 — even if XCOMP_BV\( [i] \) = 0. This is true for all values of \( i \), including 0 (x87 state) and 1 (SSE state).

- If XSTATE_BV\( [i] \) = 1, the state component is loaded with data from the XSAVE area.\(^1\) See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 and Section 13.5.12 for special treatment by XRSTORS of PT state and LBR state, respectively. See Section 13.13 for details regarding faults caused by memory accesses.

If XRSTORS is restoring a supervisor state component, the instruction causes a general-protection exception (#GP) if it would load any element of that component with an unsupported value (e.g., by setting a reserved bit in an MSR) or if a bit is set in any reserved portion of the state component in the XSAVE area.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component \( i, 2 \leq i \leq 62 \), is located in the extended region; XRSTORS uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if RFBM\( [1] \) = XSTATE_BV\( [1] \) = 1. XRSTORS causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

If an execution of XRSTORS causes an exception or a VM exit during or after restoring a supervisor state component, each element of that state component may have the value it held before the XRSTORS execution, the value loaded from the XSAVE area, or the element’s initial value (as defined in Section 13.6). See Section 13.5.6 for some special treatment of PT state for the case in which XRSTORS causes an exception or a VM exit.

Like XRSTOR, execution of XRSTORS causes the processor to update its tracking for the init and modified optimizations (see Section 13.6 and Section 13.8.3). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  - If RFBM\( [i] \) = 0, XINUSE\( [i] \) is not changed.
  - If RFBM\( [i] \) = 1 and XSTATE_BV\( [i] \) = 0, state component \( i \) may be tracked as init; XINUSE\( [i] \) may be set to 0 or 1.
  - If RFBM\( [i] \) = 1 and XSTATE_BV\( [i] \) = 1, state component \( i \) is tracked as not init; XINUSE\( [i] \) is set to 1.\(^2\)

- The processor updates its tracking for the modified optimization and records information about the XRSTORS execution for future interaction with the XSAVEOPT and XSAVES instructions as follows:
  - If RFBM\( [i] \) = 0, state component \( i \) is tracked as modified; XMODIFIED\( [i] \) is set to 1.
  - If RFBM\( [i] \) = 1, state component \( i \) may be tracked as unmodified; XMODIFIED\( [i] \) may be set to 0 or 1.
  - XRSTOR_INFO is set to the 4-tuple \( (w, x, y, z) \), where \( w \) is the CPL; \( x \) is 1 if the logical processor is in VMX non-root operation and 0 otherwise; \( y \) is the linear address of the XSAVE area; and \( z \) is XCOMP_BV (this implies that \( z[63] = 1 \)).

Note that, if RFBM is entirely zero (e.g., because the instruction mask in EDX:EAX is zero), no state components are modified, the XINUSE bitmap is not modified, and all bits are set in the XMODIFIED bitmap. Thus, if EDX:EAX was zero for the most recent execution of XRSTORS, an execution of XSAVEOPT or XSAVES will identify all state components as modified and will thus not use the modified optimization.

---

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and XSTATE_BV\( [i] \) is 1, then XCOMP_BV\( [i] \) is also 1.
13.13 MEMORY ACCESSES BY THE XSAVE FEATURE SET

Each instruction in the XSAVE feature set operates on a set of XSAVE-managed state components. The specific set of components on which an instruction operates is determined by the values of XCR0, the IA32_XSS MSR, EDX:EAX, and (for XRSTOR and XRSTORS) the XSAVE header.

Section 13.4 provides the details necessary to determine the location of each state component for any execution of an instruction in the XSAVE feature set. An execution of an instruction in the XSAVE feature set may access any byte of any state component on which that execution operates even when saving a state component is omitted because it is in its initial configuration; when restoring a state component to its initial configuration; or when XFD is enabled for the state components (see Section 13.14).

Section 13.5 provides details of the different XSAVE-managed state components. Some portions of some of these components are accessible only in 64-bit mode. Executions of XRSTOR and XRSTORS outside 64-bit mode will not update those portions; executions of XSAVE, XSAVEC, XSAVEOPT, and XSAVES will not modify the corresponding locations in memory.

Despite this fact, any execution of these instructions outside 64-bit mode may access any byte in any state component on which that execution operates — even those at addresses corresponding to registers that are accessible only in 64-bit mode. As a result, such an execution may incur a fault due to an attempt to access such an address.

For example, an execution of XSAVE outside 64-bit mode may incur a page fault if paging does not map as read/write the section of the XSAVE area containing state component 7 (Hi16_ZMM state) — despite the fact that state component 7 can be accessed only in 64-bit mode.

13.14 EXTENDED FEATURE DISABLE (XFD)

Extended feature disable (XFD) is an extension to the XSAVE feature set that allows an operating system to enable a feature while preventing specific user threads from using the feature. This section describes XFD.

As noted in Section 13.2, a processor that supports XFD enumerates CPUID.(EAX=0DH,ECX=1):EAX[4] as 1. Such a processor supports two new MSRs: IA32_XFD (MSR address 1C4H) and IA32_XFD_ERR (MSR address 1C5H). Each of these MSRs contains a state-component bitmap. Bit \( i \) of either MSR can be set to 1 only if CPUID.(EAX=0DH,ECX=i):ECX[2] is enumerated as 1 (see Section 13.2). An execution of WRMSR that attempts to set an unsupported bit in either MSR causes a general-protection fault (#GP). The reset values of both of these MSRs are zero.

XFD is enabled for state component \( i \) if XCR0[\( i \)] = IA32_XFD[\( i \)] = 1. (IA32_XFD[\( i \)] does not affect processor operations if XCR0[\( i \)] = 0.) In compacted format, the IA32_XFD MSR does not impact the computation of XCOMP_BV by the XSAVEC or XSAVES instructions and thus does not impact the format of the extended region of the XSAVE area. When XFD is enabled for a state component, any instruction that would access that state component does not execute and instead generates a device-not-available exception (#NM).

Exceptions are made for certain instructions (including those that initialize the state component). The following items provide details:

- **LDTILECFG** and **TILERELEASE** initialize the TILEDATA state component. An execution of either of these instructions does not generate #NM when XCR0[18] = IA32_XFD[18] = 1; instead, it initializes TILEDATA normally. (Note that STTILECFG does not use the TILEDATA state component. Thus, an execution of this instruction does not generate #NM when XCR0[18] = IA32_XFD[18] = 1.)
- If XRSTOR or XRSTORS is loading state component \( i \) and bit \( i \) of the XSTATE_BV field of the XSAVE header is 0, the instruction does not generate #NM when XCR0[\( i \)] = IA32_XFD[\( i \)] = 1; instead, it initializes the state component normally. (If bit \( i \) of the XSTATE_BV field of the XSAVE header is 1, the instruction does generate #NM.)
- If XSAVE, XSAVEC, XSAVEOPT, or XSAVES is saving the state component \( i \), the instruction does not generate #NM when XCR0[\( i \)] = IA32_XFD[\( i \)] = 1; instead, it operates as if XINUSE[\( i \)] = 0 (and the state component was in its initial state): it saves bit \( i \) of XSTATE_BV field of the XSAVE header as 0; in addition, XSAVE saves the initial configuration of the state component (the other instructions do not save state component \( i \)).
- Enclave entry instructions (ENCLU[EENTER] and ENCLU[ERESUME]) generate #NM if XCR0[\( i \)] = IA32_XFD[\( i \)] = 1 and bit \( i \) is set in the XFRM field in the attributes of the enclave being entered.
When XFD causes an instruction to generate #NM, the processor loads the IA32_XFD_ERR MSR to identify the disabled state component(s). Specifically, the MSR is loaded with the logical AND of the IA32_XFD MSR and the bitmap corresponding to the state component(s) required by the faulting instruction.

Device-not-available exceptions that are not due to XFD — those resulting from setting CR0.TS to 1 — do not modify the IA32_XFD_ERR MSR.
Intel® Advanced Vector Extensions (Intel® AVX) introduces 256-bit vector processing capability. The Intel AVX instruction set extends 128-bit SIMD instruction sets by employing a new instruction encoding scheme via a vector extension prefix (VEX). Intel AVX also offers several enhanced features beyond those available in prior generations of 128-bit SIMD extensions.

FMA (Fused Multiply Add) extensions enhances Intel AVX further in floating-point numeric computations. FMA provides high-throughput, arithmetic operations cover fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract.

Intel® Advanced Vector Extensions 2 (Intel® AVX2) provides 256-bit integer SIMD extensions that accelerate computation across integer and floating-point domains using 256-bit vector registers.

This chapter summarizes the key features of Intel AVX, FMA, and Intel AVX2.

14.1 INTEL® AVX OVERVIEW

Intel AVX introduces the following architectural enhancements:

• Support for 256-bit wide vectors with the YMM vector register set.
• 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
• Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
• VEX prefix-encoded instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
• Most VEX-encoded 128-bit and 256-bit AVX instructions (with both load and computational operation semantics) are not restricted to 16-byte or 32-byte memory alignment.
• Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code. With the exception of SIMD instructions operating on MMX registers, almost all legacy 128-bit SIMD instructions have AVX equivalents that support three operand syntax. 256-bit AVX instructions employ three-operand syntax and some with 4-operand syntax.

14.1.1 256-Bit Wide SIMD Register Support

Intel AVX introduces support for 256-bit wide SIMD registers (YMM0-YMM7 in operating modes that are 32-bit or less, YMM0-YMM15 in 64-bit mode). The lower 128-bits of the YMM registers are aliased to the respective 128-bit XMM registers.

Legacy SSE instructions (i.e., SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits beyond bit 128 of the YMM registers. AVX instructions with a VEX prefix and vector length of 128-bits zeroes the upper bits (above bit 128) of the YMM register.
14.1.2 Instruction Syntax Enhancements

Intel AVX employs an instruction encoding scheme using a new prefix (known as "VEX" prefix). Instruction encoding using the VEX prefix can directly encode a register operand within the VEX prefix. This support two new instruction syntax in Intel 64 architecture:

- A non-destructive operand (in a three-operand instruction syntax): The non-destructive source reduces the number of registers, register-register copies and explicit load operations required in typical SSE loops, reduces code size, and improves micro-fusion opportunities.

- A third source operand (in a four-operand instruction syntax) via the upper 4 bits in an 8-bit immediate field. Support for the third source operand is defined for selected instructions (e.g., VBLENDVPD, VBLENDVPS, PBLENDVB).

Two-operand instruction syntax previously expressed in legacy SSE instruction as

```
ADDPS xmm1, xmm2/m128
```

128-bit AVX equivalent can be expressed in three-operand syntax as

```
VADDPS xmm1, xmm2, xmm3/m128
```

In four-operand syntax, the extra register operand is encoded in the immediate byte.

Note SIMD instructions supporting three-operand syntax but processing only 128-bits of data are considered part of the 256-bit SIMD instruction set extensions of AVX, because bits 255:128 of the destination register are zeroed by the processor.

14.1.3 VEX Prefix Instruction Encoding Support

Intel AVX introduces a new prefix, referred to as VEX, in the Intel 64 and IA-32 instruction encoding format. Instruction encoding using the VEX prefix provides the following capabilities:

- Direct encoding of a register operand within VEX. This provides instruction syntax support for non-destructive source operand.

- Efficient encoding of instruction syntax operating on 128-bit and 256-bit register sets.
• Compaction of REX prefix functionality: The equivalent functionality of the REX prefix is encoded within VEX.
• Compaction of SIMD prefix functionality and escape byte encoding: The functionality of SIMD prefix (66H, F2H, F3H) on opcode is equivalent to an opcode extension field to introduce new processing primitives. This functionality is replaced by a more compact representation of opcode extension within the VEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the VEX prefix encoding.
• Most VEX-encoded SIMD numeric and data processing instruction semantics with memory operand have relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 14.9).

VEX prefix encoding applies to SIMD instructions operating on YMM registers, XMM registers, and in some cases with a general-purpose register as one of the operand. VEX prefix is not supported for instructions operating on MMX or x87 registers. Details of VEX prefix and instruction encoding are discussed in Chapter 2, “Instruction Format,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

14.2 FUNCTIONAL OVERVIEW

Intel AVX provides comprehensive functional improvements over previous generations of SIMD instruction extensions. The functional improvements include:
• 256-bit floating-point arithmetic primitives: Intel AVX enhances existing 128-bit floating-point arithmetic instructions with 256-bit capabilities for floating-point processing. Table 14-1 lists SIMD instructions promoted to Intel AVX.
• Enhancements for flexible SIMD data movements: Intel AVX provides a number of new data movement primitives to enable efficient SIMD programming in relation to loading non-unit-strided data into SIMD registers, intra-register SIMD data manipulation, conditional expression and branch handling, etc. Enhancements for SIMD data movement primitives cover 256-bit and 128-bit vector floating-point data, and across 128-bit integer SIMD data processing using VEX-encoded instructions.

![Table 14-1. Promoted SSSE3 and Intel® SSE, SSE2, SSE3, and SSE4 Instructions](image-url)
<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Group</th>
<th>Instruction</th>
<th>If No, Reason?</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>CVTPI2PS</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTSI2SS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>CVTPI2PD</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTSI2SD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVNTPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVNTPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>CVTTTPS2PI</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTTSS2SI</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>CVTTPD2PI</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTTSD2SI</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTPS2PI</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTSS2SI</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>CVTPD2PI</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTSD2SI</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>UCOMISS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>UCOMISD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>COMISS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>COMISD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY OF 5X</td>
<td>MOVMSKPS</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MOVMSKPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SQRTPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SQRTPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SQRSS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SQRSD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>RSQRTPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>RSQRTSS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>RCPSS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ANDPS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ANDPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ANDNPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ANDNPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ORPS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ORPD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>XORPS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>XORPD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ADDPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ADDSS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ADDPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ADDSD</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MULPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MULSS</td>
<td>scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MULPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MULSD</td>
<td>scalar</td>
<td></td>
</tr>
</tbody>
</table>

14-4 Vol. 1
<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Group</th>
<th>Instruction</th>
<th>If No, Reason?</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTPS2PD</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>CVTSS2SD</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTPD2PS</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>CVTSD2SS</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTDQ2PS</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTPS2DQ</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTTPS2DQ</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>SUBPS</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>SUBSS</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>SUBPD</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>SUBSD</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>MINPS</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>MINSS</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>MINPD</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>MINSD</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>DIVPS</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>DIVSS</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>DIVPD</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>DIVSD</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>MAXPS</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>MAXSS</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>MAXPD</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>MAXSD</td>
<td>scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKLBW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKLWD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKLDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PACKSSwB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PCMPGTB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PCMPGTW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PCMPGTD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PACKUSwB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKHbW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKHwD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKHDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PACKSSDw</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKLDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>PUNPCKHdq</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>MOVD</td>
<td>scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 6X</td>
<td>MOVQ</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY OF 7X</td>
<td>PSHUFD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY OF 7X</td>
<td>PSHUFLW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 7X</td>
<td>PSHUFD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 7X</td>
<td>PSHUFLW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY OF 7X</td>
<td>PSHUFD</td>
<td>VI</td>
</tr>
<tr>
<td>VEX.256 Encoding</td>
<td>VEX.128 Encoding</td>
<td>Group</td>
<td>Instruction</td>
<td>If No, Reason?</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
<td>-------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>Y0F AX</td>
<td>PCMPEQWB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>Y0F AX</td>
<td>PCMPEQW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>Y0F AX</td>
<td>PCMPEQD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F AX</td>
<td>HADDPD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F AX</td>
<td>HADDPD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F AX</td>
<td>HADDPS</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F AX</td>
<td>HADDPD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F AX</td>
<td>HADDPD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F AX</td>
<td>HADDPD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>Y0F AX</td>
<td>MOVD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>Y0F AX</td>
<td>MOVQ</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F CX</td>
<td>MOVDQA</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>Y0F CX</td>
<td>MOVDQU</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F AX</td>
<td>LDMXCSR</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F AX</td>
<td>STMXCSR</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F AX</td>
<td>CMPPS</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F AX</td>
<td>CMPPS</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>CMPSS</td>
<td>scalar</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>CMPSS</td>
<td>scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>CMPPSD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>CMPPSD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>PINSRW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>PEXTRW</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>SHUFPS</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>SHUFPS</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>SHUFPS</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F CX</td>
<td>SHUFPS</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>ADDSUBPD</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>ADDSUBPD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>ADDSUBPD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>ADDSUBPD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRLW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRLD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRLQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PADDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PADDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMULLW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMULLW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>MOVQ2DQ</td>
<td>MMX</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>MOVQ2DQ</td>
<td>MMX</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMOVMSKB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSUBUSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSUBUSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMINUB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PAND</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PADDUSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PADDUSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PADDUSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PADDUSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMAVGB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMAVGB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRAW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRAW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRAD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PSRAD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMULHUW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>YY 0F DX</td>
<td>PMULHUW</td>
<td>VI</td>
</tr>
<tr>
<td>VEX.256 Encoding</td>
<td>VEX.128 Encoding</td>
<td>Group</td>
<td>Instruction</td>
<td>If No, Reason?</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>-------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMULHW</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTPD2DQ</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTTPD2DQ</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td></td>
<td>CVTDQ2PD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>MOVNTDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSUBSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSUBSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMINSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>POR</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PADDSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PADDSP</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMAXSw</td>
<td>VI</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>YY OF FX</td>
<td>LDDQU</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSLLW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSLLD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSLLQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMULUOQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMADDWD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSADBW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>MASKMOVDQU</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSUBB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSUBW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSUBD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSUBQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PADDSP</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PADDW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PADDW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PHADDW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PHADDSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PHADD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PHSUBW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PHSUBW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PHSUBD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMADDUBSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PALIGNR</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSHUF</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PMULHRSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSIGNB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSIGNW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PSIGND</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PABSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PABSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PABSD</td>
<td>VI</td>
</tr>
</tbody>
</table>

**Notes:**
- If No, Reason? indicates the reason why the instruction is not supported in the specified encoding.
- VEX.256 Encoding and VEX.128 Encoding refer to the encoding formats supported by Intel AVX, FMA, and AVX2.
<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Group</th>
<th>Instruction</th>
<th>If No, Reason?</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.1</td>
<td>BLENDPS</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.1</td>
<td>BLENDPD</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.1</td>
<td>BLENDVPS Note 2</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.1</td>
<td>BLENDVPD Note 2</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.1</td>
<td>DPPD</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>DPPS</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>EXTRACTPS Note 3</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>INSERTPS Note 3</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>MOVNTDQA</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>MPSADBW VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PACKUSDW VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PBLENDVB VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PBLENDw VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PCMPEQQ VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PEXTRD VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PEXTRQ VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PEXTRB VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PEXTRW VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PHMINPOSUW VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PINSRB VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PINSRD VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PINSRQ VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMAXSB VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMAXSD VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMAXUD VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMAXUw VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMINSB VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMINSD VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMINUD VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMINUw VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMOVSTXxx VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMOVZXxx VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMULDQ VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.1</td>
<td>PMULLD VI</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.2</td>
<td>PTEST</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.2</td>
<td>ROUNDPD scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.2</td>
<td>ROUNDPS scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.2</td>
<td>ROUNDSD scalar</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>SSE4.2</td>
<td>ROUNDSS scalar</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.2</td>
<td>PCMPGTQ VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.2</td>
<td>CRC32c integer</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.2</td>
<td>PCMPESTRI VI</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.2</td>
<td>PCMPESTRM VI</td>
<td></td>
</tr>
</tbody>
</table>
14.2.1 256-Bit Floating-Point Arithmetic Processing Enhancements

Intel AVX provides 35 256-bit floating-point arithmetic instructions, see Table 14-2. The arithmetic operations cover add, subtract, multiply, divide, square-root, compare, max, min, round, etc., on single precision and double precision floating-point data.

The enhancement in AVX on floating-point compare operation provides 32 conditional predicates to improve programming flexibility in evaluating conditional expressions.

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Group</th>
<th>Instruction</th>
<th>If No, Reason?</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>yes</td>
<td>SSE4.2</td>
<td>PCMPISTRI</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>PCMPISTRM</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td></td>
<td>POPCNT</td>
<td>integer</td>
</tr>
</tbody>
</table>

Table 14-2. Promoted 256-Bit and 128-Bit Arithmetic Intel® AVX Instructions

14.2.2 256-Bit Non-Arithmetic Instruction Enhancements

Intel AVX provides primitives for handling data movement within 256-bit floating-point vectors and promotes many 128-bit floating data processing instructions to handle 256-bit floating-point vectors.

Intel AVX includes 39 256-bit data movement and processing instructions that are promoted from previous generations of SIMD instruction extensions, ranging from logical, blend, convert, test, unpacking, shuffling, load, and stores (see Table 14-3).

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Legacy Instruction Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>SQRTPS, SQRTPD, RSQRTPS, RCPPS</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ADDPS, ADDPD, SUBPS, SUBPD</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MULPS, MULPD, DIVPS, DIVPD</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>CVTPS2PD, CVTDP2PS</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>CVTDQ2PS, CVTPS2DQ</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>CVTTPS2DQ, CVTDP2DQ</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>CVTDP2DQ, CVTDQ2PD</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>MINPS, MINPD, MAXPS, MAXPD</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>HADDPD, HADDPS, HSUBPD, HSUBPS</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>CMPPS, CMPD</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ADDSUBPD, ADDSUBPS, DPBS</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>ROUNDPD, ROUNDPS</td>
</tr>
</tbody>
</table>

Table 14-3. Promoted 256-Bit and 128-Bit Data Movement Intel® AVX Instructions
Intel AVX introduces 18 data processing instructions that operate on 256-bit vectors, Table 14-4. These new primitives cover the following operations:

• Non-unit-strided fetching of SIMD data. Intel AVX provides several flexible SIMD floating-point data fetching primitives:
  — Broadcast of single or multiple data elements into a 256-bit destination.
  — Masked move primitives to load or store SIMD data elements conditionally.

• Intra-register manipulation of SIMD data elements. Intel AVX provides several flexible SIMD floating-point data manipulation primitives:
  — Insert/extract multiple SIMD floating-point data elements to/from 256-bit SIMD registers.
  — Permute primitives to facilitate efficient manipulation of floating-point data elements in 256-bit SIMD registers.

• Branch handling. Intel AVX provides several primitives to enable handling of branches in SIMD programming:
  — Variable blend instructions supports four-operand syntax with non-destructive source syntax. This is more flexible than the equivalent Intel SSE4 instruction syntax which uses the XMM0 register as the implied mask for blend selection.
  — Packed TEST instructions for floating-point data.

### Table 14-4. Promoted 256-Bit and 128-Bit Data Movement Intel® AVX Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBROADCASTF128 ymm1, m128</td>
<td>Broadcast 128-bit floating-point values in mem to low and high 128-bits in ymm1.</td>
</tr>
<tr>
<td>VBROADCASTSD ymm1, m64</td>
<td>Broadcast double precision floating-point element in mem to four locations in ymm1.</td>
</tr>
<tr>
<td>VBROADCASTSS ymm1, m32</td>
<td>Broadcast single precision floating-point element in mem to eight locations in ymm1.</td>
</tr>
<tr>
<td>VEXTRACTF128 xmm1/m128, ymm2, imm8</td>
<td>Extracts 128-bits of packed floating-point values from ymm2 and store results in xmm1/mem.</td>
</tr>
<tr>
<td>VINSERTF128 ymm1, ymm2, xmm3/m128, imm8</td>
<td>Insert 128-bits of packed floating-point values from xmm3/mem and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>VMASKMOVPS ymm1, ymm2, m256</td>
<td>Load packed single precision values from mem using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VMASKMOVPD ymm1, ymm2, m256</td>
<td>Load packed double precision values from mem using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VMASKMOVPS m256, ymm1, ymm2</td>
<td>Store packed single precision values from ymm2 mask in ymm1.</td>
</tr>
<tr>
<td>VMASKMOVPD m256, ymm1, ymm2</td>
<td>Store packed double precision values from ymm2 using mask in ymm1.</td>
</tr>
<tr>
<td>VPERMILPD ymm1, ymm2, ymm3/m256</td>
<td>Permute double precision floating-point values in ymm2 using controls from xmm3/mem and store result in ymm1.</td>
</tr>
</tbody>
</table>
14.2.3 Arithmetic Primitives for 128-Bit Vector and Scalar processing

Intel AVX provides a full complement of 128-bit numeric processing instructions that employ VEX-prefix encoding. These VEX-encoded instructions generally provide the same functionality over instructions operating on XMM register that are encoded using SIMD prefixes. The 128-bit numeric processing instructions in AVX cover floating-point and integer data processing; across 128-bit vector and scalar processing. Table 14-5 lists the state of promotion of legacy SIMD arithmetic ISA to VEX-128 encoding. Legacy SIMD floating-point arithmetic ISA promoted to VEX-256 encoding also support VEX-128 encoding (see Table 14-2).

The enhancement in Intel AVX on 128-bit floating-point compare operation provides 32 conditional predicates to improve programming flexibility in evaluating conditional expressions. This contrasts with floating-point SIMD compare instructions in Intel SSE and SSE2 supporting only eight conditional predicates.

### Table 14-5. Promotion of Legacy SIMD ISA to 128-Bit Arithmetic Intel® AVX Instructions

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Instruction</th>
<th>Reason Not Promoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>CVTP1PS, CVTP1PD, CVTPD2PI</td>
<td>MMX</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>CVTT1PS2PI, CVTT1PD2PI, CVTPS2PI</td>
<td>MMX</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTIS2SS, CVTIS2SD, CVTSD2SI</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTTSS2SI, CVTTSD2SI, CVTSS2SI</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>COMISD, RSQRTSS, RCPS</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>UCMISS, UCMISS, COMISS,</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>ADDSS, ADDSD, SUBSS, SUBSD</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MULSS, MULSD, DIVSS, DIVSD</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>SQRTSS, SQRTSD</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>CVTSS2SD, CVTSD2SS</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MINSS, MINSD, MAXSS, MAXSD</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PAND, PANDN, POR, PXOR</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PCMPGTB, PCMPGTW, PCMPGTD</td>
<td>VI</td>
</tr>
<tr>
<td>VEX.256 Encoding</td>
<td>VEX.128 Encoding</td>
<td>Instruction</td>
<td>Reason Not Promoted</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
<td>-------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PMADDWD, PMADDUBSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PAVGB, PAVGW, PMULUDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PCMPEQB, PCMPEQW, PCMPEQD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PMULLW, PMULHUiW, PMULHw</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PSUBSW, PADDSW, PSADBw</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PADDUSB, PADDUSW, PADDSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PSUBUSB, PSUBUSw, PSUBSB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PMINUB, PMINSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PMAXUB, PMAXSW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PADD, PADDw, PADDd, PADDq</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PSUBB, PSUBw, PSUBd, PSUBq</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PSLiW, PSLLd, PSLLq, PSRAW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PSRLW, PSRLd, PSRLq, PSRAD</td>
<td>VI</td>
</tr>
</tbody>
</table>

**CPUID.SSE3**

| no               | yes              | PHSUBw, PHSUBD, PHSUBSW | VI                  |
| no               | yes              | PHADDw, PHADDd, PHADDsw | VI                  |
| no               | yes              | PMULHRsw | VI                  |
| no               | yes              | PSIGNB, PSIGNw, PSIGNd | VI                  |
| no               | yes              | PABSB, PABSW, PABSD | VI                  |

**CPUID.SSE4_1**

| no               | yes              | DPPD |                   |
| no               | yes              | PHMINSUw, MPSADBw | VI                  |
| no               | yes              | PMAxSB, PMAxSD, PMAxUD | VI                  |
| no               | yes              | PMINSB, PMINSd, PMINUD | VI                  |
| no               | yes              | PMAxUw, PMINUw | VI                  |
| no               | yes              | PMOVXSxx, PMOVZXxx | VI                  |
| no               | yes              | PMLDQ, PMULLD | VI                  |
| no               | yes              | ROUNDSD, ROUNDSS | Scalar |

**CPUID.POPCNT**

| no               | yes              | POPCNT | Integer |

**CPUID.SSE4_2**

| no               | yes              | PCMPGTQ | VI                  |
| no               | no               | CRC32 | Integer |
| no               | yes              | PCMPESTRI, PCMPESTRM | VI                  |
| no               | yes              | PCMPISTRI, PCMPISTRM | VI                  |

**CPUID.CLMUL**

| no               | yes              | PCLMULQDQ | VI                  |

**CPUID.AESNI**
Description of Column "Reason not promoted":

- **MMX**: Instructions referencing MMX registers do not support VEX.
- **Scalar**: Scalar instructions are not promoted to 256-bit.
- **Integer**: Integer instructions are not promoted.
- **VI**: "Vector Integer" instructions are not promoted to 256-bit.

### 14.2.4 Non-Arithmetic Primitives for 128-Bit Vector and Scalar Processing

Intel AVX provides a full complement of data processing instructions that employ VEX-prefix encoding. These VEX-encoded instructions generally provide the same functionality over instructions operating on XMM register that are encoded using SIMD prefixes.

A subset of new functionalities listed in Table 14-4 is also extended via VEX.128 encoding. These enhancements in AVX on 128-bit data processing primitives include 11 new instructions (see Table 14-6) with the following capabilities:

- Non-unit-strided fetching of SIMD data. AVX provides several flexible SIMD floating-point data fetching primitives:
  - broadcast of single data element into a 128-bit destination,
  - masked move primitives to load or store SIMD data elements conditionally,
- Intra-register manipulation of SIMD data elements. AVX provides several flexible SIMD floating-point data manipulation primitives:
  - permute primitives to facilitate efficient manipulation of floating-point data elements in 128-bit SIMD registers
- Branch handling. AVX provides several primitives to enable handling of branches in SIMD programming:
  - new variable blend instructions supports four-operand syntax with non-destructive source syntax. Branching conditions dependent on floating-point data or integer data can benefit from Intel AVX. This is more flexible than non-VEX encoded instruction syntax that uses the XMM0 register as implied mask for blend selection. While variable blend with implied XMM0 syntax is supported in SSE4 using SIMD prefix encoding, VEX-encoded 128-bit variable blend instructions only support the more flexible four-operand syntax.
  - Packed TEST instructions for floating-point data.

#### Table 14-6. 128-Bit Intel® AVX Instruction Enhancement

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBROADCASTSS xmm1,m32</td>
<td>Broadcast single precision floating-point element in mem to four locations in xmm1.</td>
</tr>
<tr>
<td>VMASKMOVPS xmm1, xmm2, m128</td>
<td>Load packed single precision values from mem using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VMASKMOVPD xmm1, xmm2, m128</td>
<td>Load packed double precision values from mem using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VMASKMOVPS m128, xmm1, xmm2</td>
<td>Store packed single precision values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VMASKMOVPD m128, xmm1, xmm2</td>
<td>Store packed double precision values from xmm2 using mask in xmm1.</td>
</tr>
</tbody>
</table>
Programming with Intel® AVX, FMA, and Intel® AVX2

The 128-bit data processing instructions in AVX cover floating-point and integer data movement primitives. Legacy SIMD non-arithmetic ISA promoted to VEX-256 encoding also support VEX-128 encoding (see Table 14-3). Table 14-7 lists the state of promotion of the remaining legacy SIMD non-arithmetic ISA to VEX-128 encoding.

### Table 14-7. Promotion of Legacy SIMD ISA to 128-Bit Non-Arithmetic Intel® AVX instruction

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Instruction</th>
<th>Reason Not Promoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>MOVQ2DQ, MOVQ2DQ</td>
<td>MMX</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>LDMXCSR, STMXCSR</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVSS, MOVSD, CMPSS, CMPSD</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVHPS, MOVHPD</td>
<td>Note 1</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVLPS, MOVLPD</td>
<td>Note 1</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVLHPS, MOVHLPS</td>
<td>Redundant with VPERMILPS</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>MOVQ, MOVD</td>
<td>Scalar</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PACKUSWB, PACKSSDW, PACKSSWB</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PUNPCKHBW, PUNPCKHWd</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PUNPCKLBW, PUNPCKLDW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PUNPCKHDQ, PUNPCKLHDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PUNPCKLQDQ, PUNPCKHLDQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PSHUFW, PSHUFD</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PAND, PANDN, POR, PXOR</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PINSRW, PEXTRW</td>
<td>VI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPUID.SSSE3</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PALIGNR, PSHUFB</td>
<td>VI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPUID.SSSE4_1</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>EXTRACTPS, INSERTPS</td>
<td>Note 3</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PACKUSDW, PCMPEQQ</td>
<td>VI</td>
</tr>
</tbody>
</table>
### Table 14-7. Promotion of Legacy SIMD ISA to 128-Bit Non-Arithmetic Intel® AVX instruction (Contd.)

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Instruction</th>
<th>Reason Not Promoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>yes</td>
<td>PBLENDVB, PBLENDW</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PEXTRw, PEXTRB, PEXTRD, PEXTRQ</td>
<td>VI</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>PINSRB, PINSRD, PINSRQ</td>
<td>VI</td>
</tr>
</tbody>
</table>

Description of column "Reason not promoted":
- **MMX**: Instructions referencing MMX registers do not support VEX.
- **Scalar**: Scalar instructions are not promoted to 256-bit.
- **VI**: "Vector Integer" instructions are not promoted to 256-bit.
- **Note 1**: MOVLPD/PS and MOVHPD/PS are not promoted to 256-bit. The equivalent functionality are provided by VINSERTF128 and VEXTRACTF128 instructions as the existing instructions have no natural 256b extension
- **Note 3**: It is expected that using 128-bit INSERTPS followed by a VINSERTF128 would be better than promoting INSERTPS to 256-bit (for example).

### 14.3 DETECTION OF INTEL® AVX INSTRUCTIONS

Intel AVX instructions operate on the 256-bit YMM register state. Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 14-2.

Prior to using Intel AVX, the application must identify that the operating system supports the XGETBV instruction, the YMM register state, in addition to processor’s support for YMM state management using XSAVE/XRSTOR and AVX instructions. The following simplified sequence accomplishes both and is strongly recommended.

2. Issue XGETBV and verify that XCR0[2:1] = ‘11b’ (XMM state and YMM state are enabled by OS).  

(Step 3 can be done in any order relative to 1 and 2.)

![Figure 14-2. General Procedural Flow of Application Detection of Intel® AVX](image)

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XCR0. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.
The following pseudocode illustrates this recommended application Intel AVX detection process:

Example 14-1. Detection of Intel® AVX Instruction

```c
INT supports_AVX()
{
  mov  eax, 1
  cpuid
  and  ecx, 018000000H
  cmp  ecx, 018000000H; check both OSXSAVE and AVX feature flags
  jne  not_supported
  ; processor supports AVX instructions and XGETBV is enabled by OS
  mov  ecx, 0; specify 0 for XCR0 register
  XGETBV          ; result in EDX:EAX
  and  eax, 06H
  cmp  eax, 06H; check OS has enabled both XMM and YMM state support
  jne  not_supported
  mov  eax, 1
  jmp  done

NOT_SUPPORTED:
  mov  eax, 0
  done:
}
```

NOTE

It is unwise for an application to rely exclusively on CPUID.1:ECX.AVX[bit 28] or at all on CPUID.1:ECX.XSAVE[bit 26]: These indicate hardware support but not operating system support. If YMM state management is not enabled by an operating systems, Intel AVX instructions will #UD regardless of CPUID.1:ECX.AVX[bit 28]. “CPUID.1:ECX.XSAVE[bit 26] = 1” does not guarantee the OS actually uses the XSAVE process for state management.

These steps above also apply to enhanced 128-bit SIMD floating-pointing instructions in Intel AVX (using VEX prefix-encoding) that operate on the YMM states.
14.3.1 Detection of VEX-Encoded AES and VPCLMULQDQ

The VAESDEC/VAESDECLAST/VAESENC/VAESENCLAST/VAESEIMC/VAESKEYGENASSIST instructions operate on YMM states. The detection sequence must combine checking for CPUID.1:ECX.AES[bit 25] = 1 and the sequence for detection application support for Intel AVX.

Example 14-2. Detection of VEX-Encoded Intel® AES-NI Instructions

```assembly
INT supports_VAESNI()
{
    mov   eax, 1
    cpuid
    and   ecx, 01A000000H
    cmp   ecx, 01A000000H; check OSXSAVE AVX and AESNI feature flags
    jne   not_supported
    ; processor supports AVX and VEX-encoded AESNI and XGETBV is enabled by OS
    mov   ecx, 0; specify 0 for XCR0 register
    XGETBV ; result in EDX:EAX
    and   eax, 06H
    cmp   eax, 06H; check OS has enabled both XMM and YMM state support
    jne   not_supported
    mov   eax, 1
    jmp   done

NOT_SUPPORTED:
    mov   eax, 0
    done:
}
```

Similarly, the detection sequence for VPCLMULQDQ must combine checking for CPUID.1:ECX.PCLMULQDQ[bit 1] = 1 and the sequence for detection application support for Intel AVX.

This is shown in the pseudocode provided in Example 14-3.

Example 14-3. Detection of VEX-Encoded Intel® AES-NI Instructions

```assembly
INT supports_VPCLMULQDQ()
{
    mov   eax, 1
    cpuid
    and   ecx, 018000002H
    cmp   ecx, 018000002H; check OSXSAVE AVX and PCLMULQDQ feature flags
    jne   not_supported
    ; processor supports AVX and VEX-encoded PCLMULQDQ and XGETBV is enabled by OS
    mov   ecx, 0; specify 0 for XCR0 register
    XGETBV ; result in EDX:EAX
    and   eax, 06H
    cmp   eax, 06H; check OS has enabled both XMM and YMM state support
    jne   not_supported
    mov   eax, 1
    jmp   done

NOT_SUPPORTED:
    mov   eax, 0
    done:
}
```
14.4 HALF PRECISION FLOATING-POINT CONVERSION

VCVTPH2PS and VCVTPS2PH are two instructions supporting half precision floating-point data type conversion to and from single precision floating-point data types.

Half precision floating-point values are not used by the processor directly for arithmetic operations. But the conversion operation are subject to SIMD floating-point exceptions.

Additionally, the conversion operations of VCVTPS2PH allow programmer to specify rounding control using control fields in an immediate byte. The effects of the immediate byte are listed in Table 14-8.

Rounding control can use Imm[2] to select an override RC field specified in Imm[1:0] or use MXCSR setting.

Specific SIMD floating-point exceptions that can occur in conversion operations are shown in Table 14-9 and Table 14-10.

Table 14-8. Immediate Byte Encoding for 16-Bit Floating-Point Conversion Instructions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Name/value</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm[1:0]</td>
<td>RC=00B</td>
<td>Round to nearest even</td>
<td>If Imm[2] = 0</td>
</tr>
<tr>
<td></td>
<td>RC=01B</td>
<td>Round down</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=10B</td>
<td>Round up</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=11B</td>
<td>Truncate</td>
<td></td>
</tr>
<tr>
<td>Imm[2]</td>
<td>MS1=0</td>
<td>Use imm[1:0] for rounding</td>
<td>Ignore MXCSR.RC</td>
</tr>
<tr>
<td></td>
<td>MS1=1</td>
<td>Use MXCSR.RC for rounding</td>
<td></td>
</tr>
<tr>
<td>Imm[7:3]</td>
<td>Ignored</td>
<td>Ignored by processor</td>
<td></td>
</tr>
</tbody>
</table>

Table 14-9. Non-Numerical Behavior for VCVTPH2PS and VCVTPS2PH

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNaN</td>
<td>QNaN1(^1)</td>
<td>QNaN1(^1) (not an exception)</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN1(^2)</td>
<td>None</td>
</tr>
</tbody>
</table>

NOTES:
1. The half precision output QNaN1 is created from the single precision input QNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 5-bit exponent 1FH, and the 24-bit significand is truncated to an 11-bit significand by removing its 14 least significant bits.
2. The half precision output QNaN1 is created from the single precision input SNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 5-bit exponent 1FH, and the 24-bit significand is truncated to an 11-bit significand by removing its 14 least significant bits. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

Table 14-10. Invalid Operation for VCVTPH2PS and VCVTPS2PH

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCVTPH2PS</td>
<td>SRC = NaN</td>
<td>See Table 14-9</td>
<td>#I=1</td>
</tr>
<tr>
<td>VCVTPS2PH</td>
<td>SRC = NaN</td>
<td>See Table 14-9</td>
<td>#I=1</td>
</tr>
</tbody>
</table>

The VCVTPS2PH instruction can cause denormal exceptions if the value of the source operand is denormal relative to the numerical range represented by the source format (see Table 14-11).
The VCVTPS2PH instruction can cause an underflow exception if the result of the conversion is less than the underflow threshold for half precision floating-point data type, i.e., \(| x | < 1.0 \times 2^{-14}\).

### Table 14-12. Underflow Condition for VCVTPS2PH

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCVTPS2PH</td>
<td>Result &lt; smallest destination precision final normal value(^2)</td>
<td>Result = +0 or -0, denormal, normal. #UE =1, #PE = 1 if the result is inexact.</td>
<td>#UE=1, #PE = 1 if the result is inexact.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Masked and unmasked results are shown in Table 14-11.
2. MXCSR.FTZ is ignored, the processor behaves as if MXCSR.FTZ = 0.

The VCVTPS2PH instruction can cause an overflow exception if the result of the conversion is greater than the maximum representable value for half precision floating-point data type, i.e., \(| x | \geq 1.0 \times 2^{16}\).

### Table 14-13. Overflow Condition for VCVTPS2PH

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCVTPS2PH</td>
<td>Result ≥ largest destination precision final normal value(^1)</td>
<td>Result = +Inf or -Inf. #OE=1.</td>
<td>#OE=1.</td>
</tr>
</tbody>
</table>

The VCVTPS2PH instruction can cause an inexact exception if the result of the conversion is not exactly representable in the destination format.

### Table 14-14. Inexact Condition for VCVTPS2PH

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
</table>
| VCVTPS2PH   | The result is not representable in the destination format | res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked underflow or overflow conditions occur (this exception can occur in the presence of a masked underflow or overflow). #PE=1. | Only if no underflow/overflow condition occurred, or if the corresponding exceptions are masked:  
  - Set #OE if masked overflow and set result as described above for masked overflow.  
  - Set #UE if masked underflow and set result as described above for masked underflow.  
  If neither underflow nor overflow, result equals the result rounded to the destination precision and using the bounded exponent set #PE = 1. |

**NOTES:**
1. If a source is denormal relative to input format with DM masked and at least one of PM or UM unmasked, then an exception will be raised with DE, UE, and PE set.
14.4.1 Detection of F16C Instructions

Applications using float 16 instruction must follow a detection sequence similar to Intel AVX to ensure:

- The OS has enabled YMM state management support.
- The processor supports Intel AVX as indicated by the CPUID feature flag, i.e., CPUID.01H:ECX.AVX[bit 28] = 1.
- The processor supports 16-bit floating-point conversion instructions via a CPUID feature flag (CPUID.01H:ECX.F16C[bit 29] = 1).

Application detection of Float-16 conversion instructions follow the general procedural flow in Figure 14-3.

![Figure 14-3. General Procedural Flow of Application Detection of Float-16](image)

INT supports_f16c()
{
    ; result in eax
    mov eax, 1
    cpuid
    and ecx, 038000000H
    cmp ecx, 038000000H; check OSXSAVE, AVX, F16C feature flags
    jne not_supported
    ; processor supports AVX,F16C instructions and XGETBV is enabled by OS
    mov ecx, 0; specify 0 for XCR0 register
    XGETBV; result in EDX:EAX
    and eax, 06H
    jmp done
    cmp eax, 06H; check OS has enabled both XMM and YMM state support
    jne not_supported
    mov eax, 1
    jmp done
    not_supported:
    mov eax, 0
    done:
}
14.5 FUSED-MULTIPLY-ADD (FMA) EXTENSIONS

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

FMA extensions also provide 60 128-bit floating-point instructions to process 128-bit vector and scalar data. The arithmetic operations cover fused multiply-add, fused multiply-subtract, signed-reversed multiply on fused multiply-add and multiply-subtract.

Table 14-15. FMA Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMADD132PD/VFMADD213PD/VFMADD231PD xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Add of Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMADD132PS/VFMADD213PS/VFMADD231PS xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Add of Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMADD132SD/VFMADD213SD/VFMADD231SD xmm0, xmm1, xmm2/m64</td>
<td>Fused Multiply-Add of Scalar Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMADD132SS/VFMADD213SS/VFMADD231SS xmm0, xmm1, xmm2/m32</td>
<td>Fused Multiply-Add of Scalar Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMSUB132PD/VFMSUB213PD/VFMSUB231PD xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Subtract of Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMSUB132PS/VFMSUB213PS/VFMSUB231PS xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Multiply-Subtract of Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMSUB132SD/VFMSUB213SD/VFMSUB231SD xmm0, xmm1, xmm2/m64</td>
<td>Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFMSUB132SS/VFMSUB213SS/VFMSUB231SS xmm0, xmm1, xmm2/m32</td>
<td>Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFNMADD132PD/VFNADD213PD/VFNADD231PD xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFNMADD132PS/VFNADD213PS/VFNADD231PS xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFNMADD132SD/VFNADD213SD/VFNADD231SD xmm0, xmm1, xmm2/m64</td>
<td>Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFNMADD132SS/VFNADD213SS/VFNADD231SS xmm0, xmm1, xmm2/m32</td>
<td>Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS xmm0, xmm1, xmm2/m128; ymm0, ymm1, ymm2/m256</td>
<td>Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values</td>
</tr>
</tbody>
</table>
14.5.1 FMA Instruction Operand Order and Arithmetic Behavior

FMA instruction mnemonics are defined explicitly with an ordered three digits, e.g., VFMADD132PD. The value of each digit refers to the ordering of the three source operand as defined by instruction encoding specification:

- ‘1’: The first source operand (also the destination operand) in the syntactical order listed in this specification.
- ‘2’: The second source operand in the syntactical order. This is a YMM/XMM register, encoded using VEX prefix.
- ‘3’: The third source operand in the syntactical order. The first and third operand are encoded following ModR/M encoding rules.

The ordering of each digit within the mnemonic refers to the floating-point data listed on the right-hand side of the arithmetic equation of each FMA operation (see Table 14-17):

- The first position in the three digits of a FMA mnemonic refers to the operand position of the first FP data expressed in the arithmetic equation of FMA operation, the multiplicand.
- The second position in the three digits of a FMA mnemonic refers to the operand position of the second FP data expressed in the arithmetic equation of FMA operation, the multiplier.
- The third position in the three digits of a FMA mnemonic refers to the operand position of the FP data being added/subtracted to the multiplication result.

Note the non-numerical result of an FMA operation does not resemble the mathematically-defined commutative property between the multiplicand and the multiplier values (see Table 14-17). Consequently, software tools (such as an assembler) may support a complementary set of FMA mnemonics for each FMA instruction for ease of programming to take advantage of the mathematical property of commutative multiplications. For example, an assembler may optionally support the complementary mnemonic “VFMADD312PD” in addition to the true mnemonic “VFMADD132PD”. The assembler will generate the same instruction opcode sequence corresponding to VFMADD132PD. The processor executes VFMADD132PD and report any NAN conditions based on the definition of VFMADD132PD. Similarly, if the complementary mnemonic VFMADD123PD is supported by an assembler at source level, it must generate the opcode sequence corresponding to VFMADD123PD; the complementary mnemonic VFMADD321PD must produce the opcode sequence defined by VFMADD231PD. In the absence of FMA operations reporting a NAN result, the numerical results of using either mnemonic with an assembler supporting both mnemonics will match the behavior defined in Table 14-17. Support for the complementary FMA mnemonics by software tools is optional.

14.5.2 Fused-Multiply-ADD (FMA) Numeric Behavior

FMA instructions can perform fused-multiply-add operations (including fused-multiply-subtract, and other varieties) on packed and scalar data elements in the instruction operands. Separate FMA instructions are provided to handle different types of arithmetic operations on the three source operands.

FMA instruction syntax is defined using three source operands and the first source operand is updated based on the result of the arithmetic operations of the data elements of 128-bit or 256-bit operands, i.e., The first source operand is also the destination operand.

The arithmetic FMA operation performed in an FMA instruction takes one of several forms, \( r=(x*y)+z \), \( r=(x*y)-z \), \( r=-(x*y)+z \), or \( r=-(x*y)-z \). Packed FMA instructions can perform eight single precision FMA operations or four double precision FMA operations with 256-bit vectors.

Scalar FMA instructions only perform one arithmetic operation on the low order data element. The content of the rest of the data elements in the lower 128-bits of the destination operand is preserved. The upper 128-bits of the destination operand are filled with zero.
An arithmetic FMA operation of the form, \( r = (x \times y) + z \), takes two IEEE-754-2008 single (double) precision values and multiplies them to form an infinite precision intermediate value. This intermediate value is added to a third single (double) precision value (also at infinite precision) and rounded to produce a single (double) precision result.

Table 14-17 describes the numerical behavior of the FMA operation, \( r = (x \times y) + z \), \( r = (x \times y) - z \), \( r = -(x \times y) + z \), \( r = -(x \times y) - z \) for various input values. The input values can be 0, finite non-zero (F in Table 14-17), infinity of either sign (INF in Table 14-17), positive infinity (+INF in Table 14-17), negative infinity (-INF in Table 14-17), or NaN (including QNaN or SNaN). If any one of the input values is a NaN, the result of FMA operation, \( r \), may be a quietized NaN. The result can be either \( Q(x) \), \( Q(y) \), or \( Q(z) \), see Table 14-17. If \( x \) is a NaN, then:

- \( Q(x) = x \) if \( x \) is QNaN, or
- \( Q(x) = \) the quietized NaN obtained from \( x \) if \( x \) is SNaN.

The notation for the output value in Table 14-17 are:

- "+INF": positive infinity, "-INF": negative infinity. When the result depends on a conditional expression, both values are listed in the result column and the condition is described in the comment column.
- QNaNIndefinite represents the QNaN which has the sign bit equal to 1, the most significand field equal to 1, and the remaining significand field bits equal to 0.
- The summation or subtraction of 0s or identical values in FMA operation can lead to the following situations shown in Table 14-16.
- If the FMA computation represents an invalid operation (e.g., when adding two INF with opposite signs)), the invalid exception is signaled, and the MXCSR.IE flag is set.

### Table 14-16. Rounding Behavior of Zero Result in FMA Operation

<table>
<thead>
<tr>
<th>( x \times y )</th>
<th>( z )</th>
<th>( (x \times y) + z )</th>
<th>( (x \times y) - z )</th>
<th>( -(x \times y) + z )</th>
<th>( -(x \times y) - z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+0) (0)</td>
<td>+0 in all rounding modes</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>- 0 in all rounding modes</td>
<td></td>
</tr>
<tr>
<td>(+0) (-0)</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>+0 in all rounding modes</td>
<td>- 0 in all rounding modes</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td></td>
</tr>
<tr>
<td>(-0) (0)</td>
<td>- 0 in all rounding modes</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>+ 0 in all rounding modes</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td></td>
</tr>
<tr>
<td>(-0) (-0)</td>
<td>- 0 in all rounding modes</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>+ 0 in all rounding modes</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>-F</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>2*F</td>
<td>-2*F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>2*F</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>- 0 when rounding down, and +0 otherwise</td>
<td>-2*F</td>
</tr>
</tbody>
</table>

### Table 14-17. FMA Numeric Behavior

<table>
<thead>
<tr>
<th>( x ) (multiplicand)</th>
<th>( y ) (multiplier)</th>
<th>( z )</th>
<th>( r = (x \times y) + z )</th>
<th>( r = (x \times y) - z )</th>
<th>( r = -(x \times y) + z )</th>
<th>( r = -(x \times y) - z )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>0, F, INF, NaN</td>
<td>0, F, INF, NaN</td>
<td>Q(x)</td>
<td>Q(x)</td>
<td>Q(x)</td>
<td>Q(x)</td>
<td>Signal invalid exception if ( x ) or ( y ) or ( z ) is SNaN</td>
</tr>
<tr>
<td>0, F, INF</td>
<td>NaN</td>
<td>0, F, INF, NaN</td>
<td>Q(y)</td>
<td>Q(y)</td>
<td>Q(y)</td>
<td>Q(y)</td>
<td>Signal invalid exception if ( y ) or ( z ) is SNaN</td>
</tr>
<tr>
<td>0, F, INF</td>
<td>0, F, INF, NaN</td>
<td>NaN</td>
<td>Q(z)</td>
<td>Q(z)</td>
<td>Q(z)</td>
<td>Q(z)</td>
<td>Signal invalid exception if ( z ) is SNaN</td>
</tr>
<tr>
<td>INF</td>
<td>F, INF +INF</td>
<td>+INF</td>
<td>QNaNIndefinite</td>
<td>QNaNIndefinite</td>
<td>-INF</td>
<td>if ( x \times y ) and ( z ) have the same sign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>QNaNIndefinite -INF</td>
<td>+INF</td>
<td>QNaNIndefinite</td>
<td>QNaNIndefinite</td>
<td>+INF</td>
<td>if ( x \times y ) and ( z ) have opposite signs</td>
<td></td>
</tr>
</tbody>
</table>
If unmasked floating-point exceptions are signaled (invalid operation, denormal operand, overflow, underflow, or inexact result) the result register is left unchanged and a floating-point exception handler is invoked.

### 14.5.3 Detection of FMA

Hardware support for FMA is indicated by CPUID.1:ECX.FMA[bit 12]=1.

Application Software must identify that hardware supports AVX, after that it must also detect support for FMA by CPUID.1:ECX.FMA[bit 12]. The recommended pseudocode sequence for detection of FMA is:
INT supports_fma()
{
    ; result in eax
    mov eax, 1
    cpuid
    and ecx, 018001000H
    cmp ecx, 018001000H; check OSXSAVE, AVX, FMA feature flags
    jne not_supported
    ; processor supports AVX,FMA instructions and XGETBV is enabled by OS
    mov ecx, 0; specify 0 for XCR0 register
    XGETBV; result in EDX:EAX
    and eax, 06H
    cmp eax, 06H; check OS has enabled both XMM and YMM state support
    jne not_supported
    mov eax, 1
    jmp done
not_supported:
    mov eax, 0
done:
}

Note that FMA comprises 256-bit and 128-bit SIMD instructions operating on YMM states.

14.6 OVERVIEW OF INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions. In addition, Intel AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

14.6.1 Intel® AVX2 and 256-Bit Vector Integer Processing

Intel AVX2 promotes the vast majority of 128-bit integer SIMD instruction sets to operate with 256-bit wide YMM registers. Intel AVX2 instructions are encoded using the VEX prefix and require the same operating system support as Intel AVX. Generally, most of the promoted 256-bit vector integer instructions follow the 128-bit lane operation, similar to the promoted 256-bit floating-point SIMD instructions in Intel AVX.

Newer functionalities in Intel AVX2 generally fall into the following categories:

• Fetching non-contiguous data elements from memory using vector-index memory addressing. These “gather” instructions introduce a new memory-addressing form, consisting of a base register and multiple indices specified by a vector register (either XMM or YMM). Data elements sizes of 32 and 64-bits are supported, and data types for floating-point and integer elements are also supported.

• Cross-lane functionalities are provided with several new instructions for broadcast and permute operations. Some of the 256-bit vector integer instructions promoted from legacy SSE instruction sets also exhibit cross-lane behavior, e.g., VPMOVZ/VPMOVVS family.

• Intel AVX2 complements the Intel AVX instructions that are typed for floating-point operation with a full compliment of equivalent set for operating with 32/64-bit integer data elements.
- Vector shift instructions with per-element shift count. Data elements sizes of 32 and 64 bits are supported.

### 14.7 Promoted Vector Integer Instructions in Intel® AVX2

In Intel AVX2, most SSSE3 and Intel SSE, SSE2, SSE3, and SSE4 vector integer instructions have been promoted to support VEX.256 encodings. Table 14-18 summarizes the promotion status for existing instructions. The column “VEX.128” indicates whether the instruction using VEX.128 prefix encoding is supported. The column “VEX.256” indicates whether 256-bit vector form of the instruction using the VEX.256 prefix encoding is supported, and under which feature flag.

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Group</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>YY</td>
<td>PUNPCKLBW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PUNPCKLWD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PUNPCKLDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PACKSSWB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMPGTB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMPGTW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMPGTD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PACKUSWB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PUNPCKKBW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PUNPCKHDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PACKSSDW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PUNPCKLQDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PUNPCKHDQ</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>MOVQ</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>MOVQ</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
<td></td>
<td>MOVDQA</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
<td></td>
<td>MOVQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>YY</td>
<td>PSHUFD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>YY</td>
<td>PSHUFD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSHUFHW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSHUFHW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMPIB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMPIB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMQEQW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMQEQW</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
<td></td>
<td>MOVQ</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
<td></td>
<td>MOVQ</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PINSRW</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PEXTRW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSRLW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSRLD</td>
</tr>
<tr>
<td>VEX.256 Encoding</td>
<td>VEX.128 Encoding</td>
<td>Group</td>
<td>Instruction</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSRLQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMULLw</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMOVMSKB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBUSB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBUSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMINUB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PAND</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDUSB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDUSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMAXUB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PANDN</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>YY OF EX</td>
<td>PAVG</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSRAW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSRAD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PAVGW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMULHUW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMULHW</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
<td></td>
<td>MOVNTDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBSB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMINSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>POR</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDUSB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDUSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMAXSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PXOR</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX</td>
<td>YY OF FX</td>
<td>LDDQU</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSLLW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSLLD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSLLQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMULUDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMADDWd</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSADBw</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBw</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PBSBD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSUBQ</td>
</tr>
<tr>
<td>VEX.256 Encoding</td>
<td>VEX.128 Encoding</td>
<td>Group</td>
<td>Instruction</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
<td>---------</td>
<td>----------------------</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDDB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADDW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PADD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>SSSE3</td>
<td>PHADDB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PHADDSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PHADDD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PHSUBw</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PHSUBSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PHSUBD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMADDUBSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PALIGNR</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSHUFb</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMULHRSW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSIGNB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSIGNW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PSIGND</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PABS</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PABSw</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PABSD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>MOVNTDQA</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>MPSADBw</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PACKUSDW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PBLENDVB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PBLEND WD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PCMPEQQ</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PEXTRD</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PEXTRQ</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PEXTRB</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PEXTRW</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PHMINPOSUW</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PINSRB</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PINSRD</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PINSRQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMAXSB</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMAXSD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMAXUD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMAXUW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMINSB</td>
</tr>
</tbody>
</table>
Table 14-19 compares complementary SIMD functionalities introduced in Intel AVX and AVX2 instructions.

Table 14-19. Promoted Vector Integer SIMD Instructions in Intel® AVX2 (Contd.)

<table>
<thead>
<tr>
<th>VEX.256 Encoding</th>
<th>VEX.128 Encoding</th>
<th>Group</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMINSND</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td></td>
<td>PMINUW</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>AVX</td>
<td>PMOVSSXXX</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>AVX</td>
<td>PMOVZXX</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>AVX</td>
<td>PMULDQ</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>AVX</td>
<td>PULLLD</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>AVX</td>
<td>PTST</td>
</tr>
<tr>
<td>AVX2</td>
<td>AVX</td>
<td>SSE4.2</td>
<td>PCMPGQT</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PCMPSTRM</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PCMPSTRM</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td></td>
<td>PMPISTRM</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESDEC</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESDECLAST</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESENC</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESECN</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESIMC</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESKEYGEN</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>AESNI</td>
<td>AESKEYGEN</td>
</tr>
<tr>
<td>no</td>
<td>AVX</td>
<td>CLMUL</td>
<td>PCLMULQEQ</td>
</tr>
</tbody>
</table>

Table 14-19. VEX-Only SIMD Instructions in Intel® AVX and AVX2

<table>
<thead>
<tr>
<th>Intel® AVX2</th>
<th>Intel® AVX</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBROADCAST128</td>
<td>VBROADCAST128</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VBROADCAST</td>
<td>VBROADCASTSD</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VBROADCAST (from xmm)</td>
<td>VBROADCAST (from m64)</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VEXTRACT128</td>
<td>VEXTRACT128</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VINSERT128</td>
<td>VINSERT128</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VPMASKMOVQ</td>
<td>VPMASKMOVQ</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VPERMILPD</td>
<td>VPERMILPD</td>
<td>in-lane</td>
</tr>
<tr>
<td>VPERMILPS</td>
<td>VPERMILPS</td>
<td>in-lane</td>
</tr>
<tr>
<td>VPERM2I128</td>
<td>VPERM2I128</td>
<td>256-bit only</td>
</tr>
<tr>
<td>VPERMD</td>
<td>VPERMD</td>
<td>cross-lane</td>
</tr>
<tr>
<td>VPERMPS</td>
<td>VPERMPS</td>
<td>cross-lane</td>
</tr>
<tr>
<td>VPERMQL</td>
<td>VPERMQL</td>
<td>cross-lane</td>
</tr>
<tr>
<td>VPERMPD</td>
<td>VPERMPD</td>
<td>cross-lane</td>
</tr>
</tbody>
</table>
### Table 14-19. VEX-Only SIMD Instructions in Intel® AVX and AVX2 (Contd.)

<table>
<thead>
<tr>
<th>Intel® AVX2</th>
<th>Intel® AVX</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPBLENDD</td>
<td>VTESTPD</td>
<td></td>
</tr>
<tr>
<td>VPSLLVD/Q</td>
<td>VTESTPS</td>
<td></td>
</tr>
<tr>
<td>VPSRAVD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPSRLVD/Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGATHERDPD/QPD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGATHERDPS/QPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPGATHERDD/QD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPGATHERDQ/QQ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 14-20. New Primitive in Intel® AVX2 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPERMD ymm1, ymm2, ymm3/m256</td>
<td>Permute doublewords in ymm3/m256 using indexes in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>VPERMPD ymm1, ymm2/m256, imm8</td>
<td>Permute double precision FP elements in ymm2/m256 using indexes in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>VPERMPS ymm1, ymm2, ymm3/m256</td>
<td>Permute single precision FP elements in ymm3/m256 using indexes in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>VPERMQ ymm1, ymm2/m256, imm8</td>
<td>Permute quadwords in ymm2/m256 using indexes in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>VPSLLVD xmm1, xmm2, xmm3/m128</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSLLVQ xmm1, xmm2, xmm3/m128</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSLLVD ymm1, xmm2, ymm3/m256</td>
<td>Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSLLVQ ymm1, xmm2, ymm3/m256</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRAVD xmm1, xmm2, xmm3/m128</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in the sign bits.</td>
</tr>
<tr>
<td>VPSRLVD xmm1, xmm2, xmm3/m128</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVQ xmm1, xmm2, xmm3/m128</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVD ymm1, xmm2, ymm3/m256</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVQ ymm1, xmm2, ymm3/m256</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VGATHERDD xmm1, vm32x, xmm2</td>
<td>Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERQD xmm1, vm64x, xmm2</td>
<td>Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERDD ymm1, vm32y, ymm2</td>
<td>Using dword indices specified in vm32y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VGATHERQD ymm1, vm64y, ymm2</td>
<td>Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>
**14.7.1 Detection of Intel® AVX2**

Hardware support for Intel AVX2 is indicated by CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]=1. Application Software must identify that hardware supports Intel AVX, after that it must also detect support for Intel AVX2 by checking CPUID.(EAX=07H, ECX=0H):EBX.AVX2[bit 5]. The recommended pseudocode sequence for detection of Intel AVX2 is:

```
INT supports_avx2()
{
    ; result in eax
    mov eax, 1
    cpuid
    and ecx, 018000000H
    cmp ecx, 018000000H; check both OSXSAVE and AVX feature flags
    jne not_supported
    ; processor supports AVX instructions and XGETBV is enabled by OS
    mov eax, 7
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGATHERDPD xmm1, vm32x, xmm2</td>
<td>Using dword indices specified in vm32x, gather double precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERQPD xmm1, vm64x, xmm2</td>
<td>Using qword indices specified in vm64x, gather double precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERDPD ymm1, vm32x, ymm2</td>
<td>Using dword indices specified in vm32x, gather double precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VGATHERQPD ymm1, vm64y ymm2</td>
<td>Using qword indices specified in vm64y, gather double precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VGATHERDPs xmm1, vm32x, xmm2</td>
<td>Using dword indices specified in vm32x, gather single precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERQPS xmm1, vm64x, xmm2</td>
<td>Using qword indices specified in vm64x, gather single precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERDPs ymm1, vm32y, ymm2</td>
<td>Using dword indices specified in vm32y, gather single precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VGATHERQPS ymm1, vm64y, xmm2</td>
<td>Using qword indices specified in vm64y, gather single precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERDQ xmm1, vm32x, xmm2</td>
<td>Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERQQ xmm1, vm64x, xmm2</td>
<td>Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VGATHERDQ ymm1, vm32x, ymm2</td>
<td>Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VGATHERQQ ymm1, vm64y, ymm2</td>
<td>Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>
**14.8 ACCESSING YMM REGISTERS**

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e., SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (255:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register.

Upper bits of YMM registers (255:128) can be read and written by many instructions with a VEX.256 prefix. XSAVE and XRSTOR may be used to save and restore the upper bits of the YMM registers.

**14.9 MEMORY ALIGNMENT**

Memory alignment requirements on VEX-encoded instruction differs from non-VEX-encoded instructions. Memory alignment applies to non-VEX-encoded SIMD instructions in three categories:

- **Explicitly-aligned SIMD load and store instructions accessing 16 bytes of memory** (e.g., MOVAPD, MOVAPS, MOVDQA, etc.). These instructions always require memory address to be aligned on 16-byte boundary.
- **Explicitly-unaligned SIMD load and store instructions accessing 16 bytes or less of data from memory** (e.g., MOVUPD, MOVUPS, MOVDQU, MOVD, etc.). These instructions do not require memory address to be aligned on 16-byte boundary.
- The vast majority of arithmetic and data processing instructions in legacy SSE instructions (non-VEX-encoded SIMD instructions) support memory access semantics. When these instructions access 16 bytes of data from memory, the memory address must be aligned on 16-byte boundary.

Most arithmetic and data processing instructions encoded using the VEX prefix and performing memory accesses have more flexible memory alignment requirements than instructions that are encoded without the VEX prefix. Specifically,

- With the exception of explicitly aligned 16 or 32 byte SIMD load/store instructions, most VEX-encoded, arithmetic and data processing instructions operate in a flexible environment regarding memory address alignment, i.e., VEX-encoded instruction with 32-byte or 16-byte load semantics will support unaligned load operation by default. Memory arguments for most instructions with VEX prefix operate normally without
causing #GP(0) on any byte-granularity alignment (unlike Legacy SSE instructions). The instructions that require explicit memory alignment requirements are listed in Table 14-22.

Software may see performance penalties when unaligned accesses cross cacheline boundaries, so reasonable attempts to align commonly used data sets should continue to be pursued.

Atomic memory operation in Intel 64 and IA-32 architecture is guaranteed only for a subset of memory operand sizes and alignment scenarios. The list of guaranteed atomic operations are described in Section 9.1.1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. Intel AVX and FMA instructions do not introduce any new guaranteed atomic memory operations.

Intel AVX instructions can generate an #AC(0) fault on misaligned 4 or 8-byte memory references in Ring-3 when CR0.AM=1. 16 and 32-byte memory references will not generate #AC(0) fault. See Table 14-21 for details.

Certain Intel AVX instructions always require 16- or 32-byte alignment (see the complete list of such instructions in Table 14-22). These instructions will #GP(0) if not aligned to 16-byte boundaries (for 16-byte granularity loads and stores) or 32-byte boundaries (for 32-byte loads and stores).

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>EFLAGS.AC==1 &amp;&amp; Ring-3 &amp;&amp; CR0.AM == 1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AVX, FMA, SSE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16- or 32-byte “explicitly unaligned” loads and stores (see Table 14-23)</td>
<td>no fault</td>
<td>no fault</td>
<td></td>
</tr>
<tr>
<td>VEX op YMM, m256</td>
<td>no fault</td>
<td>no fault</td>
<td></td>
</tr>
<tr>
<td>VEX op XMM, m128</td>
<td>no fault</td>
<td>no fault</td>
<td></td>
</tr>
<tr>
<td>“explicitly aligned” loads and stores (see Table 14-22)</td>
<td>#GP(0)</td>
<td>#GP(0)</td>
<td></td>
</tr>
<tr>
<td>2, 4, or 8-byte loads and stores</td>
<td>no fault</td>
<td>#AC(0)</td>
<td></td>
</tr>
<tr>
<td>16 byte “explicitly unaligned” loads and stores (see Table 14-23)</td>
<td>no fault</td>
<td>no fault</td>
<td></td>
</tr>
<tr>
<td>op XMM, m128</td>
<td>#GP(0)</td>
<td>#GP(0)</td>
<td></td>
</tr>
<tr>
<td>“explicitly aligned” loads and stores (see Table 14-22)</td>
<td>#GP(0)</td>
<td>#GP(0)</td>
<td></td>
</tr>
<tr>
<td>2, 4, or 8-byte loads and stores</td>
<td>no fault</td>
<td>#AC(0)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 14-21. Alignment Faulting Conditions when Memory Access is Not Aligned**

**Table 14-22. Instructions Requiring Explicitly Aligned Memory**

<table>
<thead>
<tr>
<th>Require 16-byte alignment</th>
<th>Require 32-byte alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V)MOVDQA xmm, m128</td>
<td>VMOVQDA ymm, m256</td>
</tr>
<tr>
<td>(V)MOVDQA m128, xmm</td>
<td>VMOVQDA m256, ymm</td>
</tr>
<tr>
<td>(V)MOVAPS xmm, m128</td>
<td>VMOVAPS ymm, m256</td>
</tr>
<tr>
<td>(V)MOVAPS m128, xmm</td>
<td>VMOVAPS m256, ymm</td>
</tr>
<tr>
<td>(V)MOVAPD xmm, m128</td>
<td>VMOVAPD ymm, m256</td>
</tr>
<tr>
<td>(V)MOVAPD m128, xmm</td>
<td>VMOVAPD m256, ymm</td>
</tr>
<tr>
<td>(V)MOVNTPS m128, xmm</td>
<td>VMOVNTPS m256, ymm</td>
</tr>
<tr>
<td>(V)MOVNTPD m128, xmm</td>
<td>VMOVNTPD m256, ymm</td>
</tr>
<tr>
<td>(V)MOVNTDQ m128, xmm</td>
<td>VMOVNTDQ m256, ymm</td>
</tr>
<tr>
<td>(V)MOVNTDQA xmm, m128</td>
<td>VMOVNTDQA ymm, m256</td>
</tr>
</tbody>
</table>
14.10 SIMD FLOATING-POINT EXCEPTIONS

Intel AVX instructions can generate SIMD floating-point exceptions (#XM) and respond to exception masks in the same way as Legacy SSE instructions. When CR4.OSXMMEXCPT=0 any unmasked FP exceptions generate an Undefined Opcode exception (#UD).

Intel AVX FP exceptions are created in a similar fashion (differing only in number of elements) to Legacy SSE and SSE2 instructions capable of generating SIMD floating-point exceptions.

AVX introduces no new arithmetic operations (AVX floating-point are analogues of existing Legacy SSE instructions).

F16C, FMA instructions can generate SIMD floating-point exceptions (#XM). The requirements that apply to Intel AVX also apply to F16C and FMA.

The subset of Intel AVX2 instructions that operate on floating-point data do not generate #XM.

The detailed exception conditions for Intel AVX instructions and legacy SIMD instructions (excluding instructions that operates on MMX registers) are described in a number of exception class types, depending on the operand syntax and memory operation characteristics. The complete list of SIMD instruction exception class types are defined in Chapter 2, “Instruction Format,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

14.11 EMULATION

Setting the CR0.EMbit to 1 provides a technique to emulate Legacy SSE floating-point instruction sets in software. This technique is not supported with AVX instructions.

If an operating system wishes to emulate AVX instructions, set XCR0[2:1] to zero. This will cause AVX instructions to #UD. Emulation of F16C, AVX2, and FMA by operating system can be done similarly as with emulating AVX instructions.

14.12 WRITING INTEL® AVX FLOATING-POINT EXCEPTION HANDLERS

Intel AVX and FMA floating-point exceptions are handled in an entirely analogous way to Legacy SSE floating-point exceptions. To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. The section titled “SSE and SSE2 SIMD Floating-Point Exceptions” in Chapter 11, “Program-
ming with Streaming SIMD Extensions 2 (SSE2),” describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (#XM), the CR4.OSXM-MEXCPT flag (bit 10) must be set.

The guidelines for writing Intel AVX floating-point exception handlers also apply to F16C and FMA.

14.13 GENERAL PURPOSE INSTRUCTION SET ENHANCEMENTS

Enhancements in the general-purpose instruction set consist of several categories:

• A rich collection of instructions to manipulate integer data at bit-granularity. Most of the bit-manipulation instructions employ VEX-prefix encoding to support three-operand syntax with non-destructive source operands. Two of the bit-manipulating instructions (LZCNT, TZCNT) are not encoded using VEX. The VEX-encoded bit-manipulation instructions include: ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, PEXT, PDEP, SARX, SHLX, SHRX, and RORX.

• Enhanced integer multiply instruction (MULX) in conjunction with some of the bit-manipulation instructions allow software to accelerate calculation of large integer numerics (wider than 128-bits).

• INVPCID instruction targets system software that manages processor context IDs.
CHAPTER 15
PROGRAMMING WITH INTEL® AVX-512

15.1 OVERVIEW
The Intel AVX-512 family comprises a collection of instruction set extensions, including AVX-512 Foundation, AVX-512 Exponential and Reciprocal instructions, AVX-512 Conflict, AVX-512 Prefetch, and additional 512-bit SIMD instruction extensions, including AVX512-FP16. Intel AVX-512 instructions are natural extensions to Intel AVX and Intel AVX2. Intel AVX-512 introduces the following architectural enhancements:

- Support for 16 new, 512-bit SIMD registers (for a total of 32 SIMD registers, ZMM0 through ZMM31) in 64-bit mode. The extra 16 registers state is managed by the operating system using XSAVE/XRSTOR/XSAVEOPT.
- Support for 8 new opmask registers (k0 through k7) used for conditional execution and efficient merging of destination operands. The opmask register state is managed by the operating system using the XSAVE/XRSTOR/XSAVEOPT instructions.
- A new encoding prefix (referred to as EVEX) to support additional vector length encoding up to 512 bits. The EVEX prefix builds upon the foundations of the VEX prefix to provide compact, efficient encoding for functionality available to VEX encoding plus the following enhanced vector capabilities:
  - Opmasks.
  - Embedded broadcast.
  - Instruction prefix-embedded rounding control.
  - Compressed address displacements.

15.1.1 512-Bit Wide SIMD Register Support
Intel AVX-512 instructions support 512-bit wide SIMD registers (ZMM0-ZMM31). The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers and the lower 128-bit are aliased to the respective 128-bit XMM registers.

15.1.2 32 SIMD Register Support
Intel AVX-512 instructions also support 32 SIMD registers in 64-bit mode (XMM0-XMM31, YMM0-YMM31 and ZMM0-ZMM31). The number of available vector registers in 32-bit mode is still 8.

15.1.3 Eight Opmask Register Support
Intel AVX-512 instructions support 8 opmask registers (k0-k7). The width of each opmask register is architecturally defined as size MAX_KL (64 bits). Seven of the eight opmask registers (k1-k7) can be used in conjunction with EVEX-encoded AVX-512 Foundation instructions to provide conditional execution and efficient merging of data elements in the destination operand. The encoding of opmask register k0 is typically used when all data elements (unconditional processing) are desired. Additionally, the opmask registers are also used as vector flags/element-level vector sources to introduce novel SIMD functionality as seen in new instructions such as VCOMPRESSPS.
15.1.4 Instruction Syntax Enhancement

The architecture of EVEX encoding enhances the vector instruction encoding scheme in the following way:

- 512-bit vector-length, up to 32 ZMM registers, and enhanced vector programming environment are supported using the enhanced VEX (EVEX).

The EVEX prefix provides more encodable bit fields than the VEX prefix. In addition to encoding 32 ZMM registers in 64-bit mode, instruction encoding using the EVEX prefix can directly encode 7 (out of 8) opmask register operands to provide conditional processing in vector instruction programming. The enhanced vector programming environment can be explicitly expressed in the instruction syntax to include the following elements:

- An opmask operand: the opmask registers are expressed using the notation "k1" through "k7". An EVEX-encoded instruction supporting conditional vector operation using the opmask register k1 is expressed by attaching the notation {k1} next to the destination operand. The use of this feature is optional for most instructions. There are two types of masking (merging and zeroing) differentiated using the EVEX.z bit ({z} in instruction signature).

- Embedded broadcast may be supported for some instructions on the source operand that can be encoded as a memory vector. Data elements of a memory vector may be conditionally fetched or written to.

- For instruction syntax that operates only on floating-point data in SIMD registers with rounding semantics, the EVEX encoding can provide explicit rounding control within the EVEX bit fields at either scalar or 512-bit vector length.

In AVX-512 instructions, vector addition of all elements of the source operands can be expressed in the same syntax as AVX instruction:

\[
\text{VADDPS zmm1, zmm2, zmm3}
\]

Additionally, the EVEX encoding scheme of AVX-512 Foundation can express conditional vector addition as:

\[
\text{VADDPS zmm1 \{k1\}{z}, zmm2, zmm3}
\]

where:

- Conditional processing and updates to destination are expressed with an opmask register.

- Zeroing behavior of the opmask selected destination element is expressed by the \{z\} modifier (with merging as the default if no modifier is specified).
Note that some SIMD instructions supporting three-operand syntax but processing only less than or equal to 128-bits of data are considered part of the 512-bit SIMD instruction set extensions, because bits MAXVL-1:128 of the destination register are zeroed by the processor. The same rule applies to instructions operating on 256-bits of data where bits MAXVL-1:256 of the destination register are zeroed.

15.1.5 EVEX Instruction Encoding Support

Intel AVX-512 instructions employ a new encoding prefix, referred to as EVEX, in the Intel 64 and IA-32 instruction encoding format. Instruction encoding using the EVEX prefix provides the following capabilities:

- Direct encoding of a SIMD register operand within EVEX (similar to VEX). This provides instruction syntax support for three source operands.
- Compaction of REX prefix functionality and extended SIMD register encoding: the equivalent REX-prefix compaction functionality offered by the VEX prefix is provided within EVEX. Furthermore, EVEX extends the operand encoding capability to allow direct addressing of up to 32 ZMM registers in 64-bit mode.
- Compaction of SIMD prefix functionality and escape byte encoding: the functionality of a SIMD prefix (66H, F2H, F3H) on opcode is equivalent to an opcode extension field to introduce new processing primitives. This functionality is provided in the VEX prefix encoding scheme and employed within the EVEX prefix. Similarly, the functionality of the escape opcode byte (0FH) and two-byte escape (0F38H, 0F3AH) are also compacted within the EVEX prefix encoding.
- Most EVEX-encoded SIMD numeric and data processing instruction semantics with memory operands have more relaxed memory alignment requirements than instructions encoded using SIMD prefixes (see Section 15.7, “Memory Alignment”).
- Direct encoding of an opmask operand within the EVEX prefix. This provides instruction syntax support for conditional vector-element operation and merging of destination operand using an opmask register (k1-k7).
- Direct encoding of a broadcast attribute for instructions with a memory operand source. This provides instruction syntax support for elements broadcasting the second operand before being used in the actual operation.
- Compressed memory address displacements for a more compact instruction encoding byte sequence.

EVEX encoding applies to SIMD instructions operating on XMM, YMM, and ZMM registers. EVEX is not supported for instructions operating on MMX or x87 registers. Details of EVEX instruction encoding are discussed in Section 2.7, “Intel® AVX-512 Encoding,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

15.2 DETECTION OF AVX-512 FOUNDATION INSTRUCTIONS

The majority of AVX-512 Foundation instructions are encoded using the EVEX encoding scheme. EVEX-encoded instructions can operate on the 512-bit ZMM register state plus 8 opmask registers. The opmask instructions in AVX-512 Foundation instructions operate only on opmask registers or with a general purpose register. System software requirements to support the ZMM state and opmask instructions are described in Section 15.5, “Accessing XMM, YMM, AND ZMM Registers.”

Processor support of AVX-512 Foundation instructions is indicated by CPUID.(EAX=07H, ECX=0):EBX.AVX512F[bit 16] = 1. Detection of AVX-512 Foundation instructions operating on ZMM states and opmask registers needs to follow the general procedural flow in Figure 15-2.
Prior to using AVX-512 Foundation instructions, the application must identify that the operating system supports the XGETBV instruction and the ZMM register state, in addition to confirming the processor’s support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

2. Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
3. Detect CPUID.0x7.0:EBX.AVX512F[bit 16] = 1.

15.2.1 Additional 512-Bit Instruction Extensions of the Intel® AVX-512 Family

Processor support of the Intel AVX-512 Exponential and Reciprocal instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512ER[bit 27] = 1, the collection of VEXP2PD/VEXP2PS/VRCP28xx/VRSQRT28xx instructions are supported.

Processor support of the Intel AVX-512 Prefetch instructions are indicated by querying the feature flag:

- If CPUID.(EAX=07H, ECX=0):EBX.AVX512PF[bit 26] = 1, a collection of VGATHERPF0xxx/VGATHERPF1xxx/VSCATTERPF0xxx/VSCATTERPF1xxx instructions are supported.

Detection of 512-bit instructions operating on ZMM states and opmask registers, outside of AVX-512 Foundation, needs to follow the general procedural flow in Figure 15-3.

---

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XCR0 register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.
PREFETCHT1W does not require OS support for XMM/YMM/ZMM/k-reg, SIMD FP exception support.

Procedural Flow of Application Detection of other 512-bit extensions:

Prior to using the Intel AVX-512 Exponential and Reciprocal instructions, the application must identify that the operating system supports the XGETBV instruction and the ZMM register state, in addition to confirming the processor’s support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

2. Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
3. Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, and CPUID.0x7.0:EBX.AVX512ER[bit 27] = 1.

Prior to using the Intel AVX-512 Prefetch instructions, the application must identify that the operating system supports the XGETBV instruction and the ZMM register state, in addition to confirming the processor’s support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

2. Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
3. Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, and CPUID.0x7.0:EBX.AVX512PF[bit 26] = 1.

15.2.2 Detection of AVX512-FP16 Instructions

The AVX512-FP16 ISA extensions require that the AVX512BW feature be implemented since the instructions for manipulating 32b masks are associated with AVX512BW.
15.3 DETECTION OF 512-BIT INSTRUCTION GROUPS OF THE INTEL® AVX-512 FAMILY

In addition to the Intel AVX-512 Foundation instructions, the Intel AVX-512 family provides several groups of instruction extensions that can operate in vector lengths of 512/256/128 bits. Each group is enumerated by a CPUID leaf 7 feature flag and can be encoded via the EVEX.L'L field to support operation at vector lengths smaller than 512 bits. These instruction groups are listed in Table 15-1.

Table 15-1. 512-Bit Instruction Groups in the Intel® AVX-512 Family

<table>
<thead>
<tr>
<th>CPUID Leaf 07H Feature Flag Bit</th>
<th>Feature Flag Abbreviation of 512-Bit Instruction Group</th>
<th>SW Detection Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUID.(EAX=07H, ECX=0):EBX[bit 16]</td>
<td>AVX512F (AVX-512 Foundation)</td>
<td>Figure 15-2</td>
</tr>
<tr>
<td>CPUID.(EAX=07H, ECX=0):EBX[bit 28]</td>
<td>AVX512CD</td>
<td>Figure 15-4</td>
</tr>
<tr>
<td>CPUID.(EAX=07H, ECX=0):EBX[bit 17]</td>
<td>AVX512DQ</td>
<td>Figure 15-4</td>
</tr>
<tr>
<td>CPUID.(EAX=07H, ECX=0):EBX[bit 30]</td>
<td>AVX512BW</td>
<td>Figure 15-4</td>
</tr>
</tbody>
</table>

Software must follow the detection procedure for the 512-bit AVX-512 Foundation instructions as described in Section 15.2.

Detection of other 512-bit sibling instruction groups listed in Table 15-1 (excluding AVX512F) follows the procedure described in Figure 15-4.

To detect 512-bit instructions enumerated by AVX512CD, the following sequence is strongly recommended.

2. Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).
3. Verify both CPUID.0x7:0:EBX.AVX512F[bit 16] = 1, CPUID.0x7:0:EBX.AVX512CD[bit 28] = 1.

Similarly, the detection procedure for enumerating 512-bit instructions reported by AVX512DW follows the same flow.
15.4 DETECTION OF INTEL® AVX-512 INSTRUCTION GROUPS OPERATING AT 256 AND 128-BIT VECTOR LENGTHS

For each of the 512-bit instruction groups in the Intel AVX-512 family listed in Table 15-1, the EVEX encoding scheme may support a vast majority of these instructions operating at 256-bit or 128-bit (if applicable) vector lengths. Encoding support for vector lengths smaller than 512-bits is indicated by CPUID.(EAX=07H, ECX=0):EBX[bit 31], abbreviated as AVX512VL. The AVX512VL flag alone is never sufficient to determine a given Intel AVX-512 instruction may be encoded at vector lengths smaller than 512 bits. Software must use the procedure described in Figure 15-5 and Table 15-2.

To illustrate the procedure described in Figure 15-5 and Table 15-2 for software to use EVEX.256 encoded VPCONFLICT, the following sequence is provided. It is strongly recommended this sequence is followed.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use).
2) Execute XGETBV and verify that XCR0[7:5] = ‘111b’ (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = ‘11b’ (XMM state and YMM state are enabled by OS).
3) Verify CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512CD[bit 28] = 1, and CPUID.0x7.0:EBX.AVX512VL[bit 31] = 1.

Table 15-2. Feature Flag Collection Required of 256/128 Bit Vector Lengths for Each Instruction Group

<table>
<thead>
<tr>
<th>Usage of 256/128 Vector Lengths</th>
<th>Feature Flag Collection to Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX512F</td>
<td>AVX512F &amp; AVX512VL</td>
</tr>
<tr>
<td>AVX512CD</td>
<td>AVX512F &amp; AVX512CD &amp; AVX512VL</td>
</tr>
<tr>
<td>AVX512DQ</td>
<td>AVX512F &amp; AVX512DQ &amp; AVX512VL</td>
</tr>
<tr>
<td>AVX512BW</td>
<td>AVX512F &amp; AVX512BW &amp; AVX512VL</td>
</tr>
</tbody>
</table>

In some specific cases, AVX512VL may only support EVEX.256 encoding but not EVEX.128. These cases are listed in Table 15-3.
15.5 ACCESSING XMM, YMM, AND ZMM REGISTERS

The lower 128 bits of a YMM register is aliased to the corresponding XMM register. Legacy SSE instructions (i.e., SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAXVL-1:128) of the YMM registers. AVX and FMA instructions with a VEX prefix and vector length of 128-bits zeroes the upper 128 bits of the YMM register.

Upper bits of YMM registers (255:128) can be read and written to by many instructions with a VEX.256 prefix. XSAVE and XRSTOR may be used to save and restore the upper bits of the YMM registers.

The lower 256 bits of a ZMM register are aliased to the corresponding YMM register. Legacy SSE instructions (i.e., SIMD instructions operating on XMM state but not using the VEX prefix, also referred to non-VEX encoded SIMD instructions) will not access the upper bits (MAXVL-1:128) of the ZMM registers, where MAXVL is maximum vector length (currently 512 bits). AVX and FMA instructions with a VEX prefix and vector length of 128-bits zero the upper 384 bits of the ZMM register, while the VEX prefix and vector length of 256-bits zeroes the upper 256 bits of the ZMM register.

Upper bits of ZMM registers (511:256) can be read and written to by instructions with an EVEX.512 prefix.

15.6 ENHANCED VECTOR PROGRAMMING ENVIRONMENT USING EVEX ENCODING

EVEX-encoded AVX-512 instructions support an enhanced vector programming environment. The enhanced vector programming environment uses the combination of EVEX bit-field encodings and a set of eight opmask registers to provide the following capabilities:

- Conditional vector processing of an EVEX-encoded instruction. Opmask registers k1 through k7 can be used to conditionally govern the per-data-element computational operation and the per-element updates to the destination operand of an AVX-512 Foundation instruction. Each bit of the opmask register governs one vector element operation (a vector element can be 8 bits, 16 bits, 32 bits or 64 bits).

- In addition to providing predicate control on vector instructions via EVEX bit-field encoding, the opmask registers can also be used similarly on general-purpose registers as source/destination operands using modR/M encoding for non-mask-related instructions. In this case, an opmask register k0 through k7 can be selected.

- In 64-bit mode, 32 vector registers can be encoded using the EVEX prefix.

- Broadcast may be supported for some instructions on the operand that can be encoded as a memory vector. The data elements of a memory vector may be conditionally fetched or written to, and the vector size is dependent on the data transformation function.

- Flexible rounding control for the register-to-register flavor of EVEX encoded 512-bit and scalar instructions. Four rounding modes are supported by direct encoding within the EVEX prefix, overriding MXCSR settings.

- Broadcast of one element to the rest of the destination vector register.

- Compressed 8-bit displacement encoding scheme to increase the instruction encoding density for instructions that normally require disp32 syntax.
### 15.6.1 OPMASK Register to Predicate Vector Data Processing

AVX-512 instructions using EVEX encode a predicate operand to conditionally control per-element computational operation and updating of the result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers of size MAX_KL (64-bit). Note that from this set of eight architectural registers, only k1 through k7 can be addressed as a predicate operand. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand. Note also that a predicate operand can be used to enable memory fault-suppression for some instructions with a memory operand (source or destination).

As a predicate operand, the opmask registers contain one bit to govern the operation/update to each data element of a vector register. In general, opmask registers can support instructions with all element sizes: byte (int8), word (int16), single precision floating-point (float32), integer doubleword(int32), double precision floating-point (float64), integer quadword (int64). Therefore, a ZMM vector register can hold 8, 16, 32, or 64 elements in principle. The length of an opmask register, MAX_KL, is sufficient to handle up to 64 elements with one bit per element, i.e., 64 bits. Masking is supported in most of the AVX-512 instructions. For a given vector length, each instruction accesses only the number of least significant mask bits that are needed based on its data type. For example, AVX-512 Foundation instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 least significant bits of the opmask register.

An opmask register affects an AVX-512 instruction at per-element granularity. Any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in AVX-512 obeys the following properties:

- The instruction’s operation is not performed for an element if the corresponding opmask bit is not set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value must be preserved (merging-masking) or it must be zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a versatile construct to implement control-flow predication as the mask in effect provides a merging behavior for AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior is provided to remove the implicit dependency on the old value when it is not needed. Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory only accept merging-masking.

It’s important to note that the per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- Generated as a result of a vector instruction (e.g., CMP, FPCLASS, etc.).
- Loaded from memory.
- Loaded from a GPR register.
- Modified by mask-to-mask operations.

Opmask registers can be used for purposes outside of predication. For example, they can be used to manipulate sparse sets of elements from a vector, or used to set the EFLAGS based on the 0/0xFFFFFFFFFFFFFFFF/other status of the OR of two opmask registers.

#### 15.6.1.1 Opmask Register K0

The only exception to the opmask rules described above is that opmask k0 can not be used as a predicate operand. Opmask k0 cannot be encoded as a predicate operand for a vector operation; the encoding value that would select opmask k0 will instead select an implicit opmask value of 0xFFFFFFFFFFFFFFFF, thereby effectively disabling
masking. Opmask register k0 can still be used for any instruction that takes opmask register(s) as operand(s) (either source or destination).

Note that certain instructions implicitly use the opmask as an extra destination operand. In such cases, trying to use the "no mask" feature will translate into a #UD fault being raised.

15.6.1.2 Example of Opmask Usages

The example below illustrates the predicated vector add operation and predicated updates of added results into the destination operand. The initial state of vector registers zmm0, zmm1, and zmm2 and k3 are:

\[
\begin{align*}
\text{zmm0} &= [ \begin{array}{c}
0x00000003 \\
0x00000002 \\
0x00000001 \\
0x00000000 \\
0x00000007 \\
0x00000006 \\
0x00000005 \\
0x00000004 \\
0x0000000B \\
0x0000000A \\
0x00000009 \\
0x00000008 \\
0x0000000F \\
0x0000000E \\
0x0000000D \\
0x0000000C \\
\end{array} ] \text{ (bytes 15 through 0)} \\
\text{zmm1} &= [ \begin{array}{c}
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
\end{array} ] \text{ (bytes 15 through 0)} \\
\text{zmm2} &= [ \begin{array}{c}
0xAAAAAAAA \\
0xBBBBBBBB \\
0xCCCCCCCC \\
0xDDDDDDDD \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
0x0000000F \\
\end{array} ] \text{ (bytes 15 through 0)} \\
\text{k3} &= 0x8F03 \text{ (1000 1111 0000 0011)}
\end{align*}
\]

An opmask register serving as a predicate operand is expressed as a curly-braces-enclosed decorator following the first operand in the Intel assembly syntax. Given this state, we will execute the following instruction:

\[
vpaddd \text{ zmm2 } \{\text{k3}\}, \text{ zmm0, zmm1}
\]

The vpaddd instruction performs 32-bit integer additions on each data element conditionally based on the corresponding bit value in the predicate operand k3. Since per-element operations are not operated if the corresponding bit of the predicate mask is not set, the intermediate result is:

\[
\begin{align*}
\text{zmm2} &= [ \begin{array}{c}
********** \\
********** \\
0x00000010 \\
0x0000000F \\
********** \\
********** \\
0x00000019 \\
0x00000018 \\
0x00000017 \\
0x0000001E \\
********** \\
\end{array} ] \text{ (bytes 15 through 0)} \\
\text{zmm2} &= [ \begin{array}{c}
********** \\
********** \\
0x00000010 \\
0x0000000F \\
********** \\
********** \\
0x00000019 \\
0x00000018 \\
0x00000017 \\
0x0000001E \\
********** \\
\end{array} ] \text{ (bytes 15 through 0)} \\
\end{align*}
\]

where "**********" indicates that no operation is performed.

This intermediate result is then written into the destination vector register, zmm2, using the opmask register k3 as the writemask, producing the following final result:
zzm2 =
[ 0xAAAAAAAA 0xAAAAAAAA 0x00000010 0x0000000F ] (bytes 15 through 0)
[ 0xBBB88BBB 0xBBB88BBB 0xBBB88BBB 0xBBB88BBB ] (bytes 31 through 16)
[ 0x0000001A 0x00000018 0x00000018 0x00000017 ] (bytes 47 through 32)
[ 0x0000001E 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ] (bytes 63 through 48)

Note that for a 64-bit instruction (for example, vaddpd), only the 8 LSB of mask k3 (0x03) would be used to identify the predicate operation on each one of the 8 elements of the source/destination vectors.

15.6.2 OpMask Instructions

AVX-512 Foundation instructions provide a collection of opmask instructions that allow programmers to set, copy, or operate on the contents of a given opmask register. There are three types of opmask instructions:

- **Mask read/write instructions**: These instructions move data between a general-purpose integer register or memory and an opmask mask register, or between two opmask registers. For example:
  - kmovw k1, ebx; move lower 16 bits of ebx to k1.

- **Flag instructions**: This category consists of instructions that modify EFLAGS based on the content of opmask registers.
  - kortestw k1, k2; OR registers k1 and k2 and updated EFLAGS accordingly.

- **Mask logical instructions**: These instructions perform standard bitwise logical operations between opmask registers.
  - kandw k1, k2, k3; AND lowest 16 bits of registers k2 and k3, leaving the result in k1.

15.6.3 Broadcast

EVEX encoding provides a bit-field to encode data broadcast for some load-op instructions, i.e., instructions that load data from memory and perform some computational or data movement operation. A source element from memory can be broadcasted (repeated) across all the elements of the effective source operand (up to 16 times for a 32-bit data element, up to 8 times for a 64-bit data element). This is useful when reusing the same scalar operand for all the operations in a vector instruction. Note that some processors may perform multiple loads of the source element and thus software should not rely on atomicity of the data being broadcast (e.g., when the source element is simultaneously modified by another logical processor).

Broadcast is only enabled on instructions with an element size of 32 bits or 64 bits. Byte and word instructions do not support embedded broadcast.

The functionality of data broadcast is expressed as a curly-braces-enclosed decorator following the last register/memory operand in the Intel assembly syntax.

For instance:

vmulps zmm1, zmm2, [rax] {1to16}

The {1to16} primitive loads one float32 (single precision) element from memory, replicates it 16 times to form a vector of 16 32-bit floating-point elements, multiplies the 16 float32 elements with the corresponding elements in the first source operand vector, and puts each of the 16 results into the destination operand.

AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

vmovaps [rax] {k3}, zmm19
In contrast, the k3 opmask register is used as the predicate operand in the above example. Only the store operation on data elements corresponding to the non-zero bits in k3 will be performed.

### 15.6.4 Static Rounding Mode and Suppress All Exceptions

In previous SIMD instruction extensions (up to AVX and AVX2), rounding control is generally specified in MXCSR, with a handful of instructions providing per-instruction rounding override via encoding fields within the imm8 operand. AVX-512 offers a more flexible encoding attribute to override MXCSR-based rounding control for floating-pointing instructions with rounding semantics. This rounding attribute embedded in the EVEX prefix is called Static (per instruction) Rounding Mode or Rounding Mode override. This attribute allows programmers to statically apply a specific arithmetic rounding mode irrespective of the value of RM bits in MXCSR. It is available only to register-register flavors of EVEX-encoded floating-point instructions with rounding semantic. The differences between these three rounding control interfaces are summarized in Table 15-4.

The static rounding-mode override in Intel AVX-512 also implies the "suppress-all-exceptions" (SAE) attribute. The SAE effect is as if all the MXCSR mask bits are set, and none of the MXCSR flags will be updated. Using static rounding-mode via EVEX without SAE is not supported.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In such a case, vector length is assumed to be MAXVL (512-bit in case of AVX-512 packed vector instructions) or 128-bit for scalar instructions. Table 15-5 summarizes the possible static rounding-mode assignments in AVX-512 instructions.

Note that some instructions already allow specifying the rounding mode statically via immediate bits. In such cases, the immediate bits take precedence over the embedded rounding mode (in the same vein that they take precedence over whatever MXCSR.RM says).

#### Table 15-4. Characteristics of Three Rounding Control Interfaces

<table>
<thead>
<tr>
<th>Rounding Interface</th>
<th>Static Rounding Override</th>
<th>Imm8 Embedded Rounding Override</th>
<th>MXCSR Rounding Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic Requirement</td>
<td>FP rounding</td>
<td>FP rounding</td>
<td>FP rounding</td>
</tr>
<tr>
<td>Prefix Requirement</td>
<td>EVEX.B = 1</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Rounding Control</td>
<td>EVEX.L’L</td>
<td>IMM8[1:0] or MXCSR.RC (depending on IMM8[2])</td>
<td>MXCSR.RC</td>
</tr>
<tr>
<td>Suppress All Exceptions (SAE)</td>
<td>Implied</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>SIMD FP Exception #XM</td>
<td>All suppressed</td>
<td>Can raise #I, #P (unless SPE is set)</td>
<td>MXCSR masking controls</td>
</tr>
<tr>
<td>MXCSR flag update</td>
<td>No</td>
<td>yes (except PE if SPE is set)</td>
<td>Yes</td>
</tr>
<tr>
<td>Precedence</td>
<td>Above MXCSR.RC</td>
<td>Above EVEX.L’L</td>
<td>Default</td>
</tr>
<tr>
<td>Scope</td>
<td>512-bit, reg-reg, Scalar reg-reg</td>
<td>ROUNDPx, ROUNDSx, VCVTPS2PH, VRNDSCALEx</td>
<td>All SIMD operands, vector lengths</td>
</tr>
</tbody>
</table>

An example of use would be as follows:

```
vaddps zmm7 {k6}, zmm2, zmm4, {rd-sae}
```
This would perform the single precision floating-point addition of vectors zmm2 and zmm4 with round-towards-minus-infinity, leaving the result in vector zmm7 using k6 as conditional writemask.

Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

Examples of instruction instances where the static rounding-mode is not allowed are shown below:

; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2, 0x00

; instructions with memory operands
vmulps zmm7 {k6}, zmm2, [rax], {rd-sae}

; instructions with vector length different than MAXVL (512-bit)
vaddps ymm7 {k6}, ymm2, ymm4, {rd-sae}

15.6.5 Compressed Disp8*N Encoding

EVEX encoding supports a new displacement representation that allows for a more compact encoding of memory addressing commonly used in unrolled code, where an 8-bit displacement can address a range exceeding the dynamic range of an 8-bit value. This compressed displacement encoding is referred to as disp8*N, where N is a constant implied by the memory operation characteristic of each instruction.

The compressed displacement is based on the assumption that the effective displacement (of a memory operand occurring in a loop) is a multiple of the granularity of the memory access of each iteration. Since the base register in memory addressing already provides byte-granular resolution, the lower bits of the traditional disp8 operand become redundant, and can be implied from the memory operation characteristic.

The memory operation characteristics depend on the following:

• The destination operand is updated as a full vector, a single element, or multi-element tuples.
• The memory source operand (or vector source operand if the destination operand is memory) is fetched (or treated) as a full vector, a single element, or multi-element tuples.

For example:

vaddps zmm7, zmm2, disp8[membase + index*8]
The destination zmm7 is updated as a full 512-bit vector, and 64-bytes of data are fetched from memory as a full vector; the next unrolled iteration may fetch from memory in 64-byte granularity per iteration. There are 6 bits of lowest address that can be compressed, hence N = 2^6 = 64. The contribution of “disp8” to effective address calculation is 64*disp8.

vbroadcastf32x4 zmm7, disp8[membase + index*8]

In VBROADCASTF32x4, memory is fetched as a 4tuple of 4 32-bit entities. Hence the common lowest address bits that can be compressed are 4, corresponding to the 4tuple width of 2^4 = 16 bytes (4x32 bits). Therefore, N = 2^4.

For EVEX encoded instructions that update only one element in the destination, or the source element is fetched individually, the number of lowest address bits that can be compressed is generally the width in bytes of the data element, hence N = 2^(width).

15.7 MEMORY ALIGNMENT

Memory alignment requirements on EVEX-encoded SIMD instructions are similar to VEX-encoded SIMD instructions. Memory alignment applies to EVEX-encoded SIMD instructions in three categories:

• Explicitly-aligned SIMD load and store instructions accessing 64 bytes of memory with EVEX prefix encoded vector length of 512 bits (e.g., VMOVAPD, VMOVAPS, VMOVQ, etc.). These instructions always require the memory address to be aligned on a 64-byte boundary.
• Explicitly-unaligned SIMD load and store instructions accessing 64 bytes or less of data from memory (e.g., VMOVUPD, VMOVUPS, VMOVDQU, VMOVQ, VMOVQD, etc.). These instructions do not require the memory address to be aligned on a natural vector-length byte boundary.

• Most arithmetic and data processing instructions encoded using EVEX support memory access semantics. When these instructions access from memory, there are no alignment restrictions.

Software may see performance penalties when unaligned accesses cross cacheline boundaries or vector-length naturally-aligned boundaries, so reasonable attempts to align commonly used data sets should continue to be pursued.

Atomic memory operation in Intel 64 and IA-32 architecture is guaranteed only for a subset of memory operand sizes and alignment scenarios. The guaranteed atomic operations are described in Section 9.1.1, “Guaranteed Atomic Operations,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. Intel AVX and FMA instructions do not introduce any new guaranteed atomic memory operations.

Intel AVX-512 instructions may generate an #AC(0) fault on misaligned 4 or 8-byte memory references in Ring-3 when CR0.AM=1. 16, 32, and 64-byte memory references will not generate an #AC(0) fault. See Table 15-7 for details.

Certain AVX-512 Foundation instructions always require 64-byte alignment (see the complete list of VEX and EVEX encoded instructions in Table 15-6). These instructions will #GP(0) if not aligned to 64-byte boundaries.

### Table 15-6. SIMD Instructions Requiring Explicitly Aligned Memory

<table>
<thead>
<tr>
<th>Require 16-byte alignment</th>
<th>Require 32-byte alignment</th>
<th>Require 64-byte alignment*</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V)MOVDQA xmm, m128</td>
<td>VMODQA ymm, m256</td>
<td>VMODQA zmm, m512</td>
</tr>
<tr>
<td>(V)MOVDQA m128, xmm</td>
<td>VMODQA m256, ymm</td>
<td>VMODQA m512, zmm</td>
</tr>
<tr>
<td>(V)MOVAPS xmm, m128</td>
<td>VMOVAPS ymm, m256</td>
<td>VMOVAPS zmm, m512</td>
</tr>
<tr>
<td>(V)MOVAPS m128, xmm</td>
<td>VMOVAPS m256, ymm</td>
<td>VMOVAPS m512, zmm</td>
</tr>
<tr>
<td>(V)MOVAPD xmm, m128</td>
<td>VMOVAPD ymm, m256</td>
<td>VMOVAPD zmm, m512</td>
</tr>
<tr>
<td>(V)MOVAPD m128, xmm</td>
<td>VMOVAPD m256, ymm</td>
<td>VMOVAPD m512, zmm</td>
</tr>
<tr>
<td>(V)MOVNTDQA xmm, m128</td>
<td>VMOVNTPS m256, ymm</td>
<td>VMOVNTPD m512, zmm</td>
</tr>
<tr>
<td>(V)MOVNTPS m128, xmm</td>
<td>VMOVNTPD m256, ymm</td>
<td>VMOVNTPD m512, zmm</td>
</tr>
<tr>
<td>(V)MOVNTDP m128, xmm</td>
<td>VMOVNTPQD m256, ymm</td>
<td>VMOVNTPD m512, zmm</td>
</tr>
<tr>
<td>(V)MOVNTDQ m128, xmm</td>
<td>VMOVNTPDQA ymm, m256</td>
<td>VMOVNTPDQ m512, zmm</td>
</tr>
</tbody>
</table>

### Table 15-7. Instructions Not Requiring Explicit Memory Alignment

| (V)MOVDQU xmm, m128     | VMODQU ymm, m256          | VMODQU zmm, m512           |
| (V)MOVDQU m128, m128    | VMODQU m256, ymm          | VMODQU m512, zmm           |
| (V)MOVUPS xmm, m128     | MOVUPS ymm, m256          | MOVUPS zmm, m512           |
| (V)MOVUPS m128, xmm     | MOVUPS m256, ymm          | MOVUPS m512, zmm           |
| (V)MOVUPD xmm, m128     | MOVUPD ymm, m256          | MOVUPD zmm, m512           |
| (V)MOVUPD m128, xmm     | MOVUPD m256, ymm          | MOVUPD m512, zmm           |

## 15.8 SIMD FLOATING-POINT EXCEPTIONS

AVX-512 instructions can generate SIMD floating-point exceptions (#XM) if embedded “suppress all exceptions” (SAE) in EVEX is not set. When SAE is not set, these instructions will respond to exception masks of MXCSR in the same way as VEX-encoded AVX instructions. When CR4.OSXMMEXCPT=0, any unmasked FP exceptions generate an Undefined Opcode exception (#UD).
15.9 INSTRUCTION EXCEPTION SPECIFICATION

Exception behavior of VEX-encoded Intel AVX and Intel AVX2 instructions are described in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. Exception behavior of Intel AVX-512 Foundation instructions and additional 512-bit extensions are described in Section 2.8, “Exception Classifications of EVEX-Encoded instructions,” and Section 2.9, “Exception Classifications of Opmask instructions, Type K20 and Type K21.”

15.10 EMULATION

Setting the CR0.EM bit to 1 provides a technique to emulate legacy Intel SSE floating-point instruction sets in software. This technique is not supported with Intel AVX instructions, nor FMA instructions.

If an operating system wishes to emulate Intel AVX instructions, set XCR0[2:1] to zero. This will cause Intel AVX instructions to #UD. Emulation of FMA by the operating system can be done similarly as with emulating Intel AVX instructions.

15.11 WRITING FLOATING-POINT EXCEPTION HANDLERS

Intel AVX-512, Intel AVX, and FMA floating-point exceptions are handled in an entirely analogous way to legacy SSE floating-point exceptions. To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. Section 11.5.1, “SIMD Floating-Point Exceptions,” describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (#XM), the CR4.OSXM-MEXCPT flag (bit 10) must be set.
16.1 OVERVIEW

This chapter describes the software programming interface to the Intel® Transactional Synchronization Extensions of the Intel 64 architecture.

Multi-threaded applications take advantage of increasing number of cores to achieve high performance. However, writing multi-threaded applications requires programmers to reason about data sharing among multiple threads. Access to shared data typically requires synchronization mechanisms. These mechanisms ensure multiple threads update shared data by serializing operations on the shared data, often through the use of a critical section protected by a lock. Since serialization limits concurrency, programmers try to limit synchronization overheads. They do this either through minimizing the use of synchronization or through the use of fine-grain locks; where multiple locks each protect different shared data. Unfortunately, this process is difficult and error prone; a missed or incorrect synchronization can cause an application to fail. Conservatively adding synchronization and using coarser granularity locks, where a few locks each protect many items of shared data, helps avoid correctness problems but limits performance due to excessive serialization. While programmers must use static information to determine when to serialize, the determination as to whether actually to serialize is best done dynamically. Intel® Transactional Synchronization Extensions aim to improve the performance of lock-protected critical sections while maintaining the lock-based programming model.

16.2 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

Intel® Transactional Synchronization Extensions (Intel® TSX) allow the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to perform serialization only when required. This lets the hardware expose and exploit concurrency hidden in an application due to dynamically unnecessary synchronization through a technique known as lock elision.

With lock elision, the hardware executes the programmer-specified critical sections (also referred to as transactional regions) transactionally. In such an execution, the lock variable is only read within the transactional region; it is not written to (and therefore not acquired) with the expectation that the lock variable remains unchanged after the transactional region, thus exposing concurrency.

If the transactional execution completes successfully, then the hardware ensures that all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other logical processors, a process referred to as an atomic commit. Any updates performed within the transactional region are made visible to other processors only on an atomic commit.

Since a successful transactional execution ensures an atomic commit, the processor can execute the programmer-specified code section optimistically without synchronization. If synchronization was unnecessary for that specific execution, execution can commit without any cross-thread serialization.

If the transactional execution is unsuccessful, the processor cannot commit the updates atomically. When this happens, the processor will roll back the execution, a process referred to as a transactional abort. On a transactional abort, the processor will discard all updates performed in the region, restore architectural state to appear as if the optimistic execution never occurred, and resume execution non-transactionally. Depending on the policy in place, lock elision may be retried or the lock may be explicitly acquired to ensure forward progress.

Intel TSX provides two software interfaces for programmers.

- Hardware Lock Elision (HLE) is a legacy compatible instruction set extension comprising the XACQUIRE and XRELEASE prefixes.
- Restricted Transactional Memory (RTM) is an instruction set interface comprising the XBEGIN and XEND instructions.
Programmers who would like to run Intel TSX-enabled software on legacy hardware would use the HLE interface to implement lock elision. On the other hand, programmers who do not have legacy hardware requirements and who deal with more complex locking primitives would use the RTM software interface of Intel TSX to implement lock elision. In the latter case when using new instructions, the programmer must always provide a non-transactional path (which would have code to eventually acquire the lock being elided) to execute following a transactional abort and must not rely on the transactional execution alone.

Intel TSX provides the XTEST instruction to test whether a logical processor is executing transactionally, and the XABORT instruction to abort a transactional region.

A processor can perform a transactional abort for numerous reasons. A primary cause is due to conflicting accesses between the transactionally executing logical processor and another logical processor. Such conflicting accesses may prevent a successful transactional execution. Memory addresses read from within a transactional region constitute the write-set of the transactional region and addresses written to within the transactional region constitute the write-set of the transactional region. Intel TSX maintains the read- and write-sets at the granularity of a cache line.

A conflicting data access occurs if another logical processor either reads a location that is part of the transactional region’s write-set or writes a location that is a part of either the read- or write-set of the transactional region. We refer to this as a data conflict. Since Intel TSX detects data conflicts at the granularity of a cache line, unrelated data locations placed in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited transactional resources. For example, the amount of data accessed in the region may exceed an implementation-specific capacity. Additionally, some instructions and system events may cause transactional aborts.

Additionally, Intel TSX provides the XSUSLDTRK and XRESLDTRK instructions to suspend and resume load address tracking.

### 16.2.1 HLE Software Interface

HLE provides two instruction prefix hints: XACQUIRE and XRELEASE.

The programmer uses the XACQUIRE prefix in front of the instruction that is used to acquire the lock that is protecting the critical section. The processor treats the indication as a hint to elide the write associated with the lock acquire operation. Even though the lock acquire has an associated write operation to the lock, the processor does not add the address of the lock to the transactional region’s write-set nor does it issue any write requests to the lock. Instead, the address of the lock is added to the read-set. The logical processor enters transactional execution. If the lock was available before the XACQUIRE prefixed instruction, all other processors will continue to see it as available afterwards. Since the transactionally executing logical processor neither added the address of the lock to its write-set nor performed externally visible write operations to it, other logical processors can read the lock without causing a data conflict. This allows other logical processors to also enter and concurrently execute the critical section protected by the lock. The processor automatically detects any data conflicts that occur during the transactional execution and will perform a transactional abort if necessary.

Even though the eliding processor did not perform any external write operations to the lock, the hardware ensures program order of operations on the lock. If the eliding processor itself reads the value of the lock in the critical section, it will appear as if the processor had acquired the lock, i.e., the read will return the non-elided value. This behavior makes an HLE execution functionally equivalent to an execution without the HLE prefixes.

The programmer uses the XRELEASE prefix in front of the instruction that is used to release the lock protecting the critical section. This involves a write to the lock. If the instruction is restoring the value of the lock to the value it had prior to the XACQUIRE prefixed lock acquire operation on the same lock, then the processor elides the external write request associated with the release of the lock and does not add the address of the lock to the write-set. The processor then attempts to commit the transactional execution.

With HLE, if multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each other’s data, then the threads can execute concurrently and without serialization. Even though the software uses lock acquisition operations on a common lock, the hardware recognizes this, elides the lock, and executes the critical sections on the two threads without requiring any communication through the lock — if such communication was dynamically unnecessary.

If the processor is unable to execute the region transactionally, it will execute the region non-transactionally and without elision. HLE enabled software has the same forward progress guarantees as the underlying non-HLE lock-based execution. For successful HLE execution, the lock and the critical section code must follow certain guidelines.
These guidelines only affect performance; not following these guidelines will not cause a functional failure.

Hardware without HLE support will ignore the XACQUIRE and XRELEASE prefix hints and will not perform any elision since these prefixes correspond to the REPNE/REPE IA-32 prefixes which are ignored on the instructions where XACQUIRE and XRELEASE are valid. Importantly, HLE is compatible with the existing lock-based programming model. Improper use of hints will not cause functional bugs though it may expose latent bugs already in the code.

### 16.2.2 RTM Software Interface

RTM provides three instructions: XBEGIN, XEND, and XABORT.

Software uses the XBEGIN instruction to specify the start of the transactional region and the XEND instruction to specify the end of the transactional region. The XBEGIN instruction takes an operand that provides a relative offset to the fallback instruction address if the transactional region could not be successfully executed transactionally. Software using these instructions to implement lock elision must test the lock within the transactional region, and only if free should try to commit. Further, the software may also define a policy to retry if the lock is not free.

A processor may abort transactional execution for many reasons. The hardware automatically detects transactional abort conditions and restarts execution from the fallback instruction address with the architectural state corresponding to that at the start of the XBEGIN instruction and the EAX register updated to describe the abort status. The XABORT instruction allows programmers to abort the execution of a transactional region explicitly. The XABORT instruction takes an 8 bit immediate argument that is loaded into the EAX register and will thus be available to software following a transactional abort.

Hardware provides no guarantees as to whether a transactional execution will ever successfully commit. Programmers must always provide an alternative code sequence in the fallback path to guarantee forward progress. When using the instructions for lock elision, this may be as simple as acquiring a lock and executing the specified code region non-transactionally. Further, a transactional region that always aborts on a given implementation may complete transactionally on a future implementation. Therefore, programmers must ensure the code paths for the transactional region and the alternative code sequence are functionally tested.

If the RTM software interface is used for anything other than lock elision, the programmer must similarly ensure that the fallback path is inter-operable with the transactionally executing path.

### 16.3 INTEL® TSX APPLICATION PROGRAMMING MODEL

#### 16.3.1 Detection of Transactional Synchronization Support

##### 16.3.1.1 Detection of HLE Support

A processor supports HLE execution if CPUID.07H.EBX.HLE [bit 4] = 1. However, an application can use the HLE prefixes (XACQUIRE and XRELEASE) without checking whether the processor supports HLE. Processors without HLE support ignore these prefixes and will execute the code without entering transactional execution.

##### 16.3.1.2 Detection of RTM Support

A processor supports RTM execution if CPUID.07H.EBX.RTM [bit 11] = 1. An application must check if the processor supports RTM before it uses the RTM instructions (XBEGIN, XEND, and XABORT). These instructions will generate a #UD exception when used on a processor that does not support RTM.
16.3.1.3 Detection of XTEST Instruction
A processor supports the XTEST instruction if it supports either HLE or RTM. An application must check either of these feature flags before using the XTEST instruction. This instruction will generate a #UD exception when used on a processor that does not support either HLE or RTM.

16.3.1.4 Detection of Intel® TSX Suspend Load Address Tracking
A processor supports Intel TSX suspend/resume of load address tracking if CPUID.07H.EDX.TSXLDTRK [bit 16] = 1. An application must check if the processor supports Intel TSX suspend/resume of load address tracking before it uses the Intel TSX suspend/resume load address tracking instructions (XSUSLDTRK and XRESLDTRK). These instructions will generate a #UD exception when used on a processor that does not support Intel TSX suspend/resume load address tracking.

16.3.2 Querying Transactional Execution Status
The XTEST instruction can be used to determine the transactional status of a transactional region specified by HLE or RTM. Note, while the HLE prefixes are ignored on processors that do not support HLE, the XTEST instruction will generate a #UD exception when used on processors that do not support either HLE or RTM.

16.3.3 Requirements for HLE Locks
For HLE execution to successfully commit transactionally, the lock must satisfy certain properties and access to the lock must follow certain guidelines.

- An XRELEASE prefixed instruction must restore the value of the elided lock to the value it had before the lock acquisition. This allows hardware to safely elide locks by not adding them to the write-set. The data size and data address of the lock release (XRELEASE prefixed) instruction must match that of the lock acquire (XACQUIRE prefixed) and the lock must not cross a cache line boundary.
- Software should not write to the elided lock inside a transactional HLE region with any instruction other than an XRELEASE prefixed instruction, otherwise it may cause a transactional abort. In addition, recursive locks (where a thread acquires the same lock multiple times without first releasing the lock) may also cause a transactional abort. Note that software can observe the result of the elided lock acquire inside the critical section. Such a read operation will return the value of the write to the lock.

The processor automatically detects violations to these guidelines, and safely transitions to a non-transactional execution without elision. Since Intel TSX detects conflicts at the granularity of a cache line, writes to data collocated on the same cache line as the elided lock may be detected as data conflicts by other logical processors eliding the same lock.

16.3.4 Transactional Nesting
Both HLE- and RTM-based transactional executions support nested transactional regions. However, a transactional abort restores state to the operation that started transactional execution: either the outermost XACQUIRE prefixed HLE eligible instruction or the outermost XBEGIN instruction. The processor treats all nested transactional regions as one monolithic transactional region.

16.3.4.1 HLE Nesting and Elision
Programmers can nest HLE regions up to an implementation specific depth of MAX_HLE_NEST_COUNT. Each logical processor tracks the nesting count internally but this count is not available to software. An XACQUIRE prefixed HLE-eligible instruction increments the nesting count, and an XRELEASE prefixed HLE-eligible instruction decrements it. The logical processor enters transactional execution when the nesting count goes from zero to one. The logical processor attempts to commit only when the nesting count becomes zero. A transactional abort may occur if the nesting count exceeds MAX_HLE_NEST_COUNT.

In addition to supporting nested HLE regions, the processor can also elide multiple nested locks. The processor tracks a lock for elision beginning with the XACQUIRE prefixed HLE eligible instruction for that lock and ending with
the XRELEASE prefixed HLE eligible instruction for that same lock. The processor can, at any one time, track up to a MAX_HLE_ELIDED_LOCKS number of locks. For example, if the implementation supports a MAX_HLE_ELIDED_LOCKS value of two and if the programmer nests three HLE identified critical sections (by performing XACQUIRE prefixed HLE eligible instructions on three distinct locks without performing an intervening XRELEASE prefixed HLE eligible instruction on any one of the locks), then the first two locks will be elided, but the third won’t be elided (but will be added to the transaction’s write-set). However, the execution will still continue transactionally. Once an XRELEASE for one of the two elided locks is encountered, a subsequent lock acquired through the XACQUIRE prefixed HLE eligible instruction will be elided.

The processor attempts to commit the HLE execution when all elided XACQUIRE and XRELEASE pairs have been matched, the nesting count goes to zero, and the locks have satisfied the requirements described earlier. If execution cannot commit atomically, then execution transitions to a non-transactional execution without elision as if the first instruction did not have an XACQUIRE prefix.

### 16.3.4.2 RTM Nesting

Programmers can nest RTM-based transactional regions up to an implementation specific MAX_RTM_NEST_COUNT. The logical processor tracks the nesting count internally but this count is not available to software. An XBEGIN instruction increments the nesting count, and an XEND instruction decrements it. The logical processor attempts to commit only if the nesting count becomes zero. A transactional abort occurs if the nesting count exceeds MAX_RTM_NEST_COUNT.

### 16.3.4.3 Nesting HLE and RTM

HLE and RTM provide two alternative software interfaces to a common transactional execution capability. The behavior when HLE and RTM are nested together—HLE inside RTM or RTM inside HLE—is implementation specific. However, in all cases, the implementation will maintain HLE and RTM semantics. An implementation may choose to ignore HLE hints when used inside RTM regions, and may cause a transactional abort when RTM instructions are used inside HLE regions. In the latter case, the transition from transactional to non-transactional execution occurs seamlessly since the processor will re-execute the HLE region without actually doing elision, and then execute the RTM instructions.

### 16.3.5 RTM Abort Status Definition

RTM uses the EAX register to communicate abort status to software. Following an RTM abort the EAX register has the following definition.

<table>
<thead>
<tr>
<th>EAX Register Bit Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set if abort caused by XABORT instruction.</td>
</tr>
<tr>
<td>1</td>
<td>If set, the transactional execution may succeed on a retry. This bit is always clear if bit 0 is set.</td>
</tr>
<tr>
<td>2</td>
<td>Set if another logical processor conflicted with a memory address that was part of the transactional execution that aborted.</td>
</tr>
<tr>
<td>3</td>
<td>Set if an internal buffer to track transactional state overflowed.</td>
</tr>
<tr>
<td>4</td>
<td>Set if a debug exception (#DB) or breakpoint exception (#BP) was hit.</td>
</tr>
<tr>
<td>5</td>
<td>Set if an abort occurred during execution of a nested transactional execution.</td>
</tr>
<tr>
<td>23:6</td>
<td>Reserved.</td>
</tr>
<tr>
<td>31:24</td>
<td>XABORT argument (only valid if bit 0 set, otherwise reserved).</td>
</tr>
</tbody>
</table>

The EAX abort status for RTM only provides causes for aborts. It does not by itself encode whether an abort or commit occurred for the RTM region. The value of EAX can be 0 following an RTM abort. For example, a CPUID
instruction when used inside an RTM region causes a transactional abort and may not satisfy the requirements for setting any of the EAX bits. This may result in an EAX value of 0.

16.3.6 RTM Memory Ordering

A successful RTM commit causes all memory operations in the RTM region to appear to execute atomically. A successfully committed RTM region consisting of an XBEGIN followed by an XEND, even with no memory operations in the RTM region, has the same ordering semantics as a LOCK prefixed instruction.

The XBEGIN instruction does not have fencing semantics. However, if an RTM execution aborts, all memory updates from within the RTM region are discarded and never made visible to any other logical processor.

16.3.7 RTM-Enabled Debugger Support

Any debug exception (#DB) or breakpoint exception (#BP) inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address with architectural state recovered and bit 4 in EAX set. However, to allow software debuggers to intercept execution on debug or breakpoint exceptions, the RTM architecture provides additional capability called advanced debugging of RTM transactional regions.

Advanced debugging of RTM transactional regions is enabled if bit 11 of DR7 and bit 15 of the IA32_DEBUGCTL MSR are both 1. In this case, any RTM transactional abort due to a #DB or #BP causes execution to roll back to just before the XBEGIN instruction (EAX is restored to the value it had before XBEGIN) and then delivers a #DB. (A #DB is delivered even if the transactional abort was caused by a #BP.) DR6[16] is cleared to indicate that the exception resulted from a debug or breakpoint exception inside an RTM region. See also Section 18.3.3, “Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

16.3.8 Intel® TSX Suspend/Resume Load Address Tracking Support

Programmers can use Intel TSX suspend/resume of load address tracking to choose which memory accesses do not need to be tracked in the Intel TSX read set. A programmer who uses the suspend/resume load address tracking feature must ensure that there are no atomicity requirements related to the addresses they choose to exclude from the read set as hardware will not detect read-write conflicts for those addresses.

To prevent load addresses from being entered into the read set, the programmer should use the XSUSLDTRK and XRESLDTRK instructions. The XSUSLDTRK instruction suspends loads tracking and thus specifies the start of a suspend region; addresses of subsequent loads will not be added to the transaction read set. The XRESLDTRK instruction resumes load tracking and thus specifies the end of a suspend region; addresses of subsequent loads will be added to the transaction read set.

The execution of a suspend region is similar to transaction execution with the following exceptions:

- The addresses of loads in a suspend region are not tracked for read-write conflicts if the addresses are accessed inside the suspend region only (they are not added to the transaction read set). The addresses are still tracked if they are accessed outside of the suspend region inside the transaction.
- Transaction start/end inside a suspend region is not supported; any execution of XBEGIN or XEND inside a suspend region will cause the transaction to abort.
- There is no support for nesting of suspend regions; execution of XSUSLDTRK in a suspend region will cause a transaction to abort.

16.3.9 Programming Considerations

Typical programmer-identified regions are expected to execute transactionally and to commit successfully. However, Intel TSX does not provide any such guarantee. A transactional execution may abort for many reasons. To take full advantage of the transactional capabilities, programmers should follow certain guidelines to increase the probability of their transactional execution committing successfully.

This section discusses various events that may cause transactional aborts. The architecture ensures that updates performed within a transactional region that subsequently aborts execution will never become visible. Only a
committed transactional execution updates architectural state. Transactional aborts never cause functional failures and only affect performance.

16.3.9.1 Instruction Based Considerations

Programmers can use any instruction safely inside a transactional region. Further, programmers can use the Intel TSX instructions and prefixes at any privilege level. However, some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path.

Intel TSX allows for most common instructions to be used inside transactional regions without causing aborts. The following operations inside a transactional region do not typically cause an abort.

- Operations on the instruction pointer register, general purpose registers (GPRs) and the status flags (CF, OF, SF, PF, AF, and ZF).
- Operations on XMM and YMM registers and the MXCSR register

However, programmers must be careful when intermixing SSE and AVX operations inside a transactional region. Intermixing SSE instructions accessing XMM registers and AVX instructions accessing YMM registers may cause transactional regions to abort.

CLD and STD instructions when used inside transactional regions may cause aborts if they change the value of the DF flag. However, if DF is 1, the STD instruction will not cause an abort. Similarly, if DF is 0, the CLD instruction will not cause an abort.

Instructions not enumerated here as causing abort when used inside a transactional region will typically not cause the execution to abort (examples include but are not limited to MFENCE, LFENCE, SFENCE, RDTSC, RDTSCP, etc.).

The following instructions will abort transactional execution on any implementation:

- XABORT
- CPUID
- PAUSE
- ENCLS
- ENCLU

In addition, in some implementations, the following instructions may always cause transactional aborts. These instructions are not expected to be commonly used inside typical transactional regions. However, programmers must not rely on these instructions to force a transactional abort, since whether they cause transactional aborts is implementation dependent.

- Operations on X87 and MMX architecture state. This includes all MMX and X87 instructions, including the FXRSTOR and FXSAVE instructions.
- Update to non-status portion of EFLAGS or to UIF: CLI, CLUI, STI, STUI, POPFD, POPFQ, CLAC, and STAC.
- Instructions that update segment registers, debug registers and/or control registers: MOV to DS/ES/FS/GS/SS, POP DS/ES/FS/GS/SS, LDS, LES, LFS, LGS, LSS, SWAPGS, WRFSBASE, WRGSBASE, LGDT, SGDT, LIDT, SIDT, LLDT, SLDT, LTR, STR, Far CALL, Far JMP, Far RET, IRET, MOV to DRx, MOV to CR0/CR2/CR3/CR4/CR8, CLTS, and LMSW.
- Ring transitions: SYSENTER, SYSEXIT, and SYSRET.
- Extended state management: XRSTOR, XRSTORS, XSAVE, XSAVED, XSAVEOPT, XSAVES, and XSETBV.
- Interrupts: INT n, INTO, INT3, and INT1.
- I/O: IN, INS, REP INS, OUT, OUTS, REP OUTS and their variants.
- VMX: VMXPRD, VMXPRST, VMXCLAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, INVVPI, and VMFUNC.
- SMX: GETSEC.
- UD0, UD1, UD2, RSM, RDMSR, WRMSR, WRPKRU, HLT, MONITOR, MWAIT, and VZEROUPPER.
16.3.9.2 Runtime Considerations

In addition to the instruction-based considerations, runtime events may cause transactional execution to abort. These may be due to data access patterns or micro-architectural implementation causes. Keep in mind that the following list is not a comprehensive discussion of all abort causes.

Any fault or trap in a transactional region that must be exposed to software will be suppressed. Transactional execution will abort and execution will transition to a non-transactional execution, as if the fault or trap had never occurred. If any exception is not masked, that will result in a transactional abort and it will be as if the exception had never occurred.

When executed in VMX non-root operation, certain instructions may result in a VM exit. When such instructions are executed inside a transactional region, then instead of causing a VM exit, they will cause a transactional abort and the execution will appear as if instruction that would have caused a VM exit never executed.

Synchronous exception events (#DE, #OF, #NP, #SS, #GP, #BR, #UD, #AC, #XM, #PF, #NM, #TS, #MF, #DB, #BP/INT3) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred. With HLE, since the non-transactional code path is identical to the transactional code path, these events will typically re-appear when the instruction that caused the exception is re-executed non-transactionally, causing the associated synchronous events to be delivered appropriately in the non-transactional execution. The same behavior also applies to synchronous events (EPT violations, EPT misconfigurations, and accesses to the APIC-access page) that occur in VMX non-root operation.

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a non-transactional execution. The asynchronous events will be pended and handled after the transactional abort is processed. The same behavior also applies to asynchronous events (VMX-preemption timer expiry, virtual-interrupt delivery, and interrupt-window exiting) that occur in VMX non-root operation.

Transaction execution only supports write-back cacheable memory type operations. A transactional region may always abort if it includes operations on any other memory type. This includes instruction fetches to UC memory type.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. The behavior of how the processor handles this is implementation specific. Some implementations may allow the updates to these flags to become externally visible even if the transactional region subsequently aborts. Some Intel TSX implementations may choose to abort the transactional execution if these flags need to be updated. Further, a processor's page-table walk may generate accesses to its own transactionally written but uncommitted state. Some Intel TSX implementations may choose to abort the execution of a transactional region in such situations. Regardless, the architecture ensures that, if the transactional region aborts, then the transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs.

Executing self-modifying code transactionally may also cause transactional aborts. Programmers must continue to follow the Intel recommended guidelines for writing self-modifying and cross-modifying code even when employing Intel TSX.

While an Intel TSX implementation will typically provide sufficient resources for executing common transactional regions, implementation constraints and excessive sizes for transactional regions may cause a transactional execution to abort and transition to a non-transactional execution. The architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed.

Conflicting requests to a cache line accessed within a transactional region may prevent the transactional region from executing successfully. For example, if logical processor P0 reads line A in a transactional region and another logical processor P1 writes A (either inside or outside a transactional region) then logical processor P0 may abort if logical processor P1's write interferes with processor P0's ability to execute transactionally. Similarly, if P0 writes line A in a transactional region and P1 reads or writes A (either inside or outside a transactional region), then P0 may abort if P1's access to A interferes with P0's ability to execute transactionally. In addition, other coherence traffic may at times appear as conflicting requests and may cause aborts. While these false conflicts may happen, they are expected to be uncommon. The conflict resolution policy to determine whether P0 or P1 aborts in the above scenarios is implementation specific.
17.1 INTRODUCTION

Return-oriented programming (ROP), and similarly CALL/JMP-oriented programming (COP/JOP), have been the prevalent attack methodologies for stealth exploit writers targeting vulnerabilities in programs. These attack methodologies have the common elements:

- A code module with execution privilege and contain small snippets of code sequence with the characteristic: at least one instruction in the sequence being a control transfer instruction that depends on data either in the return stack or in a register for the target address.
- Diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target (via modification in the data stack or in the register).

Control-Flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/COP/JOP style control-flow subversion attacks:

- Shadow stack: Return address protection to defend against ROP.
- Indirect branch tracking: Free branch protection to defend against COP/JOP.

Both capabilities introduce new instruction set extensions, and are described in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

Control-Flow Enforcement Technology introduces a new exception (#CP) with interrupt vector 21.

17.1.1 Shadow Stack

A shadow stack is a second stack for the program that is used exclusively for control transfer operations. This stack is separate from the data stack and can be enabled for operation individually in user mode or supervisor mode. When shadow stacks are enabled, the CALL instruction pushes the return address on both the data and shadow stack. The RET instruction pops the return address from both stacks and compares them. If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP). Note that the shadow stack only holds the return addresses and not parameters passed to the call instruction.

The shadow stack is protected from tamper through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table protections are extended to support an additional attribute for pages to mark them as "Shadow Stack" pages. When shadow stacks are enabled, control transfer instructions/flows like near call, far call, call to interrupt/exception handlers, etc. store return addresses to the shadow stack and the access will fault if the underlying page is not marked as a "Shadow Stack" page. However stores from instructions like MOV, XSAVE, etc. will not be allowed. Likewise control transfer instructions like near RET, far RET, IRET, etc. when they attempt to read from the shadow stack the access will fault if the underlying page is not marked as a "Shadow Stack" page. This paging protection detects and prevents conditions that cause an overflow or underflow of the shadow stack when the shadow stack is delimited by non-shadow stack guard pages, or any malicious attempts to redirect the processor to consume data from addresses that are not shadow stack addresses.

17.1.2 Indirect Branch Tracking

The ENDBRANCH instruction is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. This instruction opcode is selected to be one that is a NOP on legacy machines such that programs compiled with ENDBRANCH new instruction continue to function on old machines without the CET enforcement. On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the processor pipeline to detect control flow violations. The CPU implements a state machine that tracks indirect JMP and CALL instructions. When one of these instructions is executed, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state. In WAIT_FOR_ENDBRANCH state the next instruction in the program
stream must be an ENDBRANCH. If the next instruction is not an ENDBRANCH, the processor causes a control protection exception (#CP); otherwise, the state machine moves back to IDLE state.

17.1.3 Speculative Behavior when CET is Enabled

Speculative execution of near indirect JMP/CALL/RET indirect branches may be able to create an active side channel vulnerability that reveals the contents of data.

There are two basic methods that an attacker may be able to use to control indirect branch speculation in order to speculatively execute code that causes a side channel:

1. Attacker controlled prediction.
2. Attacker controlled jump redirection.

With attacker controlled prediction, the attacker trains indirect branch predictors such that the desired victim indirect branch goes to the attacker desired location. Examples include Branch Target Injection (also called “Variant 2” and “Spectre”) and RSB wrap on underflow (also called “ret2spec”).

With attacker controlled jump redirection, the attacker controls a speculative-only value used as input to the indirect branch so that the branch mispredicts to the attacker desired location. Examples of this include Bound Check Bypass Store (where a speculative store containing an attacker controlled value may overwrite the indirect branch target before the load of the target) and Speculative Store Bypass (where a load of the indirect branch target may bypass the most recent store of the target value and thus speculatively read an older attacker controlled value at the same memory location).

In addition to the existing mitigation features like IBRS, STIBP, and IBPB, processors supporting CET will have a variety of additional features to constrain control flow speculation in order to mitigate such attacks. For details on these features, see Section 17.2.6, “Constraining Execution at Targets of RET,” and Section 17.3.8, “Constraining Speculation after Missing ENDBRANCH.”

17.2 SHADOW STACKS

A shadow stack is a second expand down stack used exclusively for control transfer operations. This stack is separate from the data stack. The shadow stack is not used to store data and hence is not explicitly writeable by software. Writes to the shadow stack are restricted to control transfer instructions and shadow stack management instructions. The shadow stack feature can be enabled separately in user mode (CPL == 3) or supervisor mode (CPL < 3).

Shadow stacks operate only in protected mode. Shadow stacks cannot be enabled in virtual 8086 mode.

It is recommended to not configure the shadow stack in the linear address range 0 to 64 KB or adjacent to the canonical address boundary.

17.2.1 Shadow Stack Pointer and its Operand and Address Size Attributes

When CET is enabled the processor supports a new architectural register, shadow stack pointer (SSP), when the processor supports the shadow stack feature. The SSP cannot be directly encoded as a source, destination or memory operand in instructions. The SSP points to the current top of the shadow stack.

The width of the shadow stack is 32-bit in 32-bit/compatibility mode and is 64-bit in 64-bit mode. The address-size attribute of the shadow stack is likewise 32-bit in 32-bit/compatibility mode and 64-bit in 64-bit mode.

17.2.2 Terminology

When shadow stacks are enabled, certain control transfer instructions/flows and shadow stack management instructions do loads and stores from and to the shadow stack. Such loads and stores from control transfer instructions and shadow stack management instructions are termed as shadow-stack loads and shadow-stack stores to distinguish them from a loads and stores performed by other instructions like MOV, XSAVES, etc.
The pseudocode for the instruction operations use the notation ShadowStackEnabled(CPL) as a test of whether shadow stacks are enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

ShadowStackEnabled(CPL):
  IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
    IF CPL = 3
      (* Obtain the shadow stack enable from IA32_U_CET MSR (MSR address 6A0H) used to enable feature for CPL = 3 *)
      SHADOW_STACK_ENABLED = IA32_U_CET.SH_STK_EN;
    ELSE
      (* Obtain the shadow stack enable from IA32_S_CET MSR (MSR address 6A2H) used to enable feature for CPL < 3 *)
      SHADOW_STACK_ENABLED = IA32_S_CET.SH_STK_EN;
    FI;
    IF SHADOW_STACK_ENABLED = 1
      THEN
        return TRUE;
      ELSE
        return FALSE;
    FI;
  ELSE
    (* Shadow stacks not enabled in real mode and virtual-8086 mode or if the master CET feature enable in CR4 is disabled *)
    return FALSE;
  ENDIF

Additionally, the following terms are used.

- **ShadowStackPush4B**: Decrements the shadow stack pointer (SSP) by 4 bytes and copies the 4 byte source operand to the top of the shadow stack.
- **ShadowStackPush8B**: Decrements the shadow stack pointer (SSP) by 8 bytes and copies the 8 byte source operand to the top of the shadow stack.
- **ShadowStackPop4B**: Copies 4 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 4 bytes to point to the new top of stack.
- **ShadowStackPop8B**: Copies 8 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 8 bytes to point to the new top of stack.
- **shadow_stack_lock_cmpxchg8B(address, new_value, expected_value)**: this function executes atomically and compares the expected_value to the 8 byte read from memory specified by the address operand using a locked shadow-stack load. If the two values are equal, the new_value is written to the address using an unlocking shadow-stack store. If the two values are not equal, then the value read by the shadow-stack load is written back, also using an unlocking shadow-stack store. The function returns the value read from the memory specified by the address operand.

### 17.2.3 Supervisor Shadow Stack Token

On an inter-privilege far CALL or when calling an interrupt/exception handler at a higher privilege level, a stack switch occurs; if shadow stacks are enabled at the new privilege level, then a shadow stack switch occurs. Shadow stacks that can be switched to by hardware as part of a privilege change are required to have a supervisor shadow stack token set up by the supervisor to provide the address of the new SSP register. The supervisor shadow stack tokens also serve the purpose of enforcing that a shadow stack can be made active on only one logical processor when switched to by the processor. The supervisor shadow stack token must be set up only on shadow stacks intended to be used on these transfers. The address of the supervisor shadow stack token is programmed into the
IA32_PLx_SSP MSR (where 0 ≤ x ≤ 2). The WRMSR and XRSTORS instructions require the address specified in the IA32_PLx_SSP MSR (where 0 ≤ x ≤ 2) to be 4 byte aligned; otherwise, the instruction causes a general protection exception (#GP(0)).

The supervisor shadow stack token is a 64-bit value formulated as follows.
- Bit 63:3: Bits 63:3 of the linear address of the supervisor shadow stack token.
- Bit 2: Reserved. Must be zero.
- Bit 1: Reserved. Must be zero.
- Bit 0: Busy bit. If 0, indicates this shadow stack is not active on any logical processor. If 1, indicates this shadow stack is currently active on one of the logical processors.

The following figure illustrates a supervisor shadow stack with a supervisor shadow stack token located at its base.

![Figure 17-1. Supervisor Shadow Stack with a Supervisor Shadow Stack Token](image)

If the far CALL or event delivery will push a 24-byte stack frame after the token is acquired, the 8-byte supervisor shadow stack token and the stack frame must be fully contained within a 32-byte region that is aligned to 32-bytes on the shadow stack. If they are not, a general-protection exception (#GP(0)) occurs.

The processor does the following checks prior to switching to a supervisor shadow stack programmed into the IA32_PLx_SSP MSR. These steps are performed atomically.

1. Load the supervisor shadow stack token from the address specified in the IA32_PLx_SSP MSR using a locked shadow-stack store.
2. Check if the busy bit in the token is 0; reserved bits must be 0.
3. Check if the address programmed in the MSR matches the address in the supervisor shadow stack token; reserved bits must be 0.
4. If checks 2 and 3 are successful, then set the busy bit in the token using an unlocking shadow-stack store and switching the SSP to the value specified in the IA32_PLx_SSP MSR.
5. If checks 2 or 3 fail, write back the value read at step 1 using an unlocking shadow-stack store (the busy bit is not set) and raise a #GP(0) exception.

If the far CALL or event delivery pushes a stack frame after the token is acquired and any of the pushes causes a fault or VM exit, the processor will revert to the old shadow stack and the busy bit in the new shadow stack's token remains set. The new shadow stack is said to be prematurely busy. Software should enable supervisor shadow stacks only if it is certain that this situation cannot occur. If CPUID.(EAX=07H,ECX=1H):EDX[bit 18] is enumerated as 1, it is sufficient for an operating system to ensure that none of the pushes can cause a page fault.

On a far RET to a lesser privilege level or on an IRET that switches shadow stack, the instruction clears the busy bit in the shadow stack token as follows. These steps are also performed atomically.

1. Load the supervisor shadow stack token from the SSP using a locked shadow-stack load.
2. Check if the busy bit in the token is 1; reserved bits must be 0.
3. Check if the address programmed in supervisor shadow stack token matches SSP; reserved bits must be 0.
4. If checks 2 and 3 are successful, then write back the token with an unlocking shadow-stack store, clearing the busy bit; otherwise, write back the value read at step 1 using an unlocking shadow-stack store and continue without modifying the contents of the shadow stack pointed to by SSP.

17.2.4 Shadow Stack Usage on Task Switch

A task switch (see Chapter 8, “Task Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A) may be invoked by:

- JMP or CALL instruction to a TSS descriptor in the GDT.
- JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.

With shadow stack enabled, the new task must be associated with a 32-bit TSS and must not be in virtual-8086 mode. The 32-bit SSP for the new task is located at offset 104 in the 32-bit TSS. Thus the TSS of the new task must be at least 108 bytes. This SSP is required to be 8 byte aligned, and required to point to a “supervisor shadow stack” token (though the task may be at CPL3).

On a task switch initiated by a CALL instruction, an interrupt, or exception, the SSP of the old task is pushed onto the shadow stack of the new task along with the CS and LIP of the old task. This is true even for a nested task switch initiated by a CALL instruction. Likewise, on a task switch initiated by IRET, the SSP of the new task is restored from the shadow stack of old task. The CS and LIP on the shadow stack of the old task are matched against the return address determined by the CS and EIP of the new task. If the match fails, a #CP(FAR-RET/IRET) exception is reported.

17.2.5 Switching Shadow Stacks

The architecture provides a mechanism to switch shadow stacks using a pair of instructions; RSTORSSP and SAVEPREVSSP. The RSTORSSP instruction verifies a shadow-stack-restore token located at the top of the new shadow stack and referenced by the memory operand of this instruction. After RSTORSSP determines the validity of the restore point on the new shadow stack, it switches the SSP to point to the token. The shadow-stack-restore token is a 64-bit value formatted as follows.

- Bit 63:2: Value of shadow stack pointer when this restore point was created.
- Bit 1: Reserved. Must be zero.
- Bit 0: Mode bit. If 0, the token is a compatibility/legacy mode shadow-stack-restore token. If 1, then this shadow stack restore token can be used with a RSTORSSP instruction in 64-bit mode.

The shadow-stack-restore token is created by the SAVEPREVSSP instruction. The operating system may also create a restore point on a shadow stack by creating a shadow-stack-restore token.

Once the shadow stack has been switched to a new shadow stack by the RSTORSSP instruction, software can create a restore point on the old shadow stack by executing the SAVEPREVSSP instruction. In order to allow the SAVEPREVSSP instruction to determine the address where to save the shadow-stack-restore token, the RSTORSSP instruction replaces the shadow-stack-restore token with a previous-ssp token that holds the value of the SSP at the time the RSTORSSP instruction was invoked. The previous-ssp token is formatted as follows.

- Bit 63:2: Shadow stack pointer when the RSTORSSP instruction was invoked, i.e., the SSP of the old shadow stack.
- Bit 1: Set to 1.
- Bit 0: Mode bit. If 0, then this previous-ssp token can be used with a SAVEPREVSSP instruction in compatibility/legacy mode. If 1, then this previous-ssp token can be used with a SAVEPREVSSP instruction in 64-bit mode.

The following figure illustrates the RSTORSSP instruction operation during a shadow stack switching sequence.
In this example, the initial SSP is 1000H and the shadow-stack-restore token is on a new shadow stack at address 3FF8H. The token at address 3FF8H holds the SSP when this restore point was created; in this example it is 4000H.

In order to switch to the new shadow stack, the RSTORSSP instruction is invoked with the memory operand pointing set to 3FF8H. When the RSTORSSP instruction completes, the SSP is set to 3FF8H and the shadow-stack-restore token at 3FF8H is replaced by a previous-ssp token that holds the address 1000H, i.e., the old SSP.

The following figure illustrates the SAVEPREVSSP instruction operation during a shadow stack switching sequence.

To allow switching back to this old shadow stack, a SAVEPREVSSP instruction is now invoked. The SAVEPREVSSP instruction does not take any memory operand and expects to find a previous-ssp token at the top of the shadow stack, i.e., at address 3FF8H. The SAVEPREVSSP instruction then saves a shadow-stack-restore token on the old shadow stack at address FF8H, and the token itself holds the address 1000H which is the address recorded in the previous-ssp token. The SAVEPREVSSP instruction also pops the previous-ssp token off the current shadow stack and thus the SSP following SAVEPREVSSP is 4000H.

Subsequently to switch back to the old shadow stack, a RSTORSSP instruction may be invoked with memory operand set to FF8H.

If, following a switch to a new shadow stack, it is not required to create a restore point on the old shadow stack, then the previous-ssp token created by the RSTORSSP instruction can be popped off the shadow stack by using the INCSSP instruction.

See the SAVEPREVSSP and RSTORSSP instruction operations for the detailed algorithm.
17.2.6 Constraining Execution at Targets of RET

Instructions at the target of a RET instruction will not execute, even speculatively, if the RET addresses (either from normal stack or shadow stack) are speculative-only or do not match, unless the target of the RET is also predicted (e.g., by a Return Stack Buffer prediction), when CET shadow stack is enabled. A RET address would be speculative-only if it was modified by an older speculative-only store, or was an older value than the most recent value stored to that address on the logical processor.

17.3 INDIRECT BRANCH TRACKING

When the indirect branch tracking feature is active, the indirect JMP/CALL instruction behavior changes as follows.

- **JMP**: If the next instruction retired after an indirect JMP is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, then a #CP fault is generated. Below JMP instructions are tracked to enforce an ENDBRANCH. Note that Jcc, RIP relative, and far direct JMP are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
  - JMP r/m16, JMP r/m32, JMP r/m64
  - JMP m16:16, JMP m16:32, JMP m16:64
- **CALL**: If the next instruction retired after an indirect CALL is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 in 64-bit mode, then a #CP fault is generated. Below CALL instructions are tracked to enforce an ENDBRANCH. Note that relative and zero displacement forms of CALL instructions are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
  - CALL r/m16, CALL r/m32, CALL r/m64
  - CALL m16:16, CALL m16:32, CALL m16:64

The ENDBR32 and ENDBR64 instructions will have the same effect as the NOP instruction on Intel 64 processors that do not support CET. On processors supporting CET, these instructions do not change register or flag state. This allows CET instrumented programs to execute on processors that do not support CET. Even when CET is supported and enabled, these NOP-like instructions do not affect the execution state of the program, do not cause any additional register pressure, and are minimally intrusive from power and performance perspectives.

The processor implements two dual-state machines to track indirect CALL/JMP for terminations. One state machine is maintained for user mode and one for supervisor mode. At reset the user and supervisor mode state machines are in IDLE state.

On instructions other than indirect CALL/JMP, the state machine stays in the IDLE state.

On an indirect CALL or JMP instruction, the state machine transitions to the WAIT_FOR_ENDBRANCH state.

In the WAIT_FOR_ENDBRANCH state, the indirect branch tracking state machine verifies the next instruction is an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, and either:

- Causes a #CP fault, or
- Allows the next instruction if legacy compatibility configuration allows (see Section 17.3.6).

The priority of the #CP(ENDBRANCH) exception relative to other events is as follows.
Higher priority faults/traps/events that occur at the end of an indirect CALL/JMP are delivered ahead of any #CP(ENDBRANCH) fault. The CET state machine at the privilege level where the higher priority fault/trap/event occurred retains its state when the control transfers to the fault/trap/event handler. The instruction pointer pushed on the stack for a #CP(ENDBRANCH) fault is the address of the instruction at the target of the indirect CALL/JMP that caused the fault.

### 17.3.1 No-track Prefix for Near Indirect CALL/JMP

CET allows software to designate certain indirect CALL and JMP instructions as "non-tracked indirect control transfer instructions". Software (e.g., compiler generated code for switch statements, jump tables, etc.) should use the no-track prefix only if they have generated code to validate the possible targets of this CALL/JMP to be legal targets. Software (e.g., compilers), when using the no-track prefix with CALL/JMP where an absolute offset is specified indirectly in a memory location, should ensure that such memory locations cannot be tampered. When enabled by setting the NO_TRACK_EN control in the IA32_U_CET/IA32_S_CET MSR, near indirect CALL and JMP instructions when prefixed with 3EH do not modify the CET indirect branch tracker. Far CALL and JMP instructions are always tracked and ignore the 3EH prefix. When this control is 0, near indirect CALL and JMP instructions are always tracked irrespective of the presence of the 3EH prefix.

In 64-bit mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix if there isn’t a 64H/65H prefix on the instruction.

In legacy/compatibility mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix when it is the last group 2 prefix on the instruction.
17.3.2 Terminology

The pseudocode for the instruction operations use a notation EndbranchEnabled(CPL) as a test of whether indirect branch tracking is enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

EndbranchEnabled(CPL):

```
IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
  IF CPL = 3
    THEN (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL = 3 *)
      ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
    ELSE (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL < 3 *)
      ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
    FI;
  IF ENDBR_ENABLED = 1
    THEN return TRUE;
    ELSE return FALSE;
  FI;
ELSE (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature enable in CR4 is disabled *)
  return FALSE;
ENDIF
```

Likewise the notation EndbranchEnabledAndNotSuppressed is defined as follows:

EndbranchEnabledAndNotSuppressed(CPL):

```
IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
  IF CPL = 3
    THEN (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL = 3 *)
      ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
    SUPPRESSED = IA32_U_CET.SUPPRESS;
    ELSE (* Obtain the ENDBRANCH enable from MSR used to enable feature for CPL < 3 *)
      ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
    SUPPRESSED = IA32_S_CET.SUPPRESS;
    FI;
  IF ENDBR_ENABLED = 1 AND SUPPRESSED = 0
    THEN return TRUE;
    ELSE return FALSE;
  FI;
ELSE (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature enable in CR4 is disabled *)
  return FALSE;
ENDIF
17.3.3 Indirect Branch Tracking

The hardware implements two CET indirect branch tracker state machines, one for user mode (CPL == 3) and one for supervisor mode (CPL < 3). At any time, which of the CET indirect branch trackers is in the active state depends on the CPL of the machine. When a user space program is executing, the CPL 3 CET indirect branch tracker is active. When supervisor mode software is executing, the CPL < 3 tracker is active. This section describes the various control transfer conditions and the tracker state on those transfers.

17.3.3.1 Control Transfers between CPL 3 and CPL < 3

Some events and instructions can cause control transfer to occur from CPL 3 to CPL < 3, and vice versa. As part of the CPL change the hardware also switches the active CET indirect branch tracker. For example, when an interrupt occurs during execution of a user mode (CPL == 3) program and it causes the CPL to switch to supervisor mode (CPL < 3) then, as part of the CPL change, the user mode CET indirect branch tracker becomes inactive and the supervisor mode CET indirect branch tracker becomes active. A subsequent IRET is used by the interrupt handler to return to the interrupted user mode program. This IRET causes the processor to switch the CPL to user mode (CPL == 3) and, as part of the CPL change, the supervisor mode CET indirect branch tracker becomes inactive and the user mode CET indirect branch tracker becomes active.

The CPL where the event or instruction that caused the control transfer occurs is termed the source CPL, and the CET indirect branch tracker state at that CPL is referred here as the source CET indirect branch tracker state. The CPL reached at the end of the control transfer is termed the destination CPL, and the CET indirect branch tracker state at that CPL is referred to as the destination CET indirect branch tracker state.

This section describes various cases of control transfers that occur between user mode (CPL 3) and supervisor mode (CPL < 3).

In all these cases the source CET indirect branch tracker state becomes not active and retains its state (IDLE, WAIT_FOR_ENDBRANCH), and the target CET indirect branch tracker state becomes active if there was no fault during the transfer.

- **Case 1: Far CALL/JMP, SYSCALL/SYSENTER**
  - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT_FOR_ENDBRANCH. This enforces that the subroutine invoked by a far CALL/JMP must begin with an ENDBRANCH.

- **Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks**
  - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT_FOR_ENDBRANCH.

- **Case 3: IRET/Far RET**
  - If indirect branch tracking enabled, the target indirect branch tracker becomes active and keeps its state. If the user mode was interrupted by a higher priority event, like an interrupt at the end of the indirect CALL/JMP, then when an IRET or Far RET is used to return to the interrupted user mode program, the user mode indirect branch tracker retains its state and a #CP fault will occur if the next instruction decoded is not an ENDBR32/64 according to mode of machine.

17.3.3.2 Control Transfers within CPL 3 or CPL < 3

Some events and instructions can cause control transfer to occur within CPL 3 or CPL < 3. For such transfers since the CPL class does not change, the same indirect branch tracker is used at the beginning and end of the control transfer.

- **Case 1: Far CALL/JMP, Near indirect CALL/JMPCALL/JMP**
  - Far CALL/JMP: If indirect branch tracking is enabled, active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
  - Near indirect CALL/JMPCALL/JMP: If indirect branch tracking is enabled and not suppressed, active indirect branch tracker goes to WAIT_FOR_ENDBRANCH.

- **Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks**
If indirect branch tracking is enabled, the active indirect branch tracker is unsuppressed and goes to WAIT FOR ENDBRANCH.

Case 3: IRET

If indirect branch tracking is enabled, the active indirect branch tracker keeps its state.

17.3.4 Indirect Branch Tracking State Machine

The state machine is described by Table 17-1.

Table 17-1. Indirect Branch Tracking State Machine

<table>
<thead>
<tr>
<th>Current State</th>
<th>Trigger</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1</td>
<td>Instructions other than indirect CALL/JMP or 3EH prefixed near indirect CALL/JMP and NO_TRACK_EN=1</td>
<td>TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1</td>
</tr>
<tr>
<td></td>
<td>Indirect CALL/JMP without 3EH prefix</td>
<td>TRACKER=WAIT FOR ENDBRANCH, SUPPRESS=0, ENDBR_EN=1</td>
</tr>
<tr>
<td></td>
<td>Indirect CALL/JMP with 3EH prefix and NO_TRACK_EN=0 Far CALL/JMP</td>
<td>TRACKER=WAIT FOR ENDBRANCH, SUPPRESS=0, ENDBR_EN=1</td>
</tr>
<tr>
<td>TRACKER= WAIT FOR ENDBRANCH, SUPPRESS=0, ENDBR_EN=1</td>
<td>INT3/INT1</td>
<td>TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1</td>
</tr>
<tr>
<td></td>
<td>ENDBRANCH instruction</td>
<td>TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1</td>
</tr>
<tr>
<td></td>
<td>Successful ENCLU[ERESUME]</td>
<td>TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1</td>
</tr>
</tbody>
</table>
| | Instructions other than ENDBRANCH, successful ENCLU[ERESUME] or INT3 or INT1 | If legacy compatibility treatment is not enabled or if not allowed by legacy code page bitmap:  
- No state change and deliver #CP (ENDBRANCH)  
- If legacy compatibility treatment is enabled and transfer allowed by legacy code page bitmap:  
  - TRACKER=IDLE, SUPPRESS=SUPPRESS_DIS, ENDBR_EN=1 | TRACKER=IDLE, SUPPRESS=SUPPRESS_DIS, ENDBR_EN=1 |
| TRACKER=x, SUPPRESS=x, ENDBR_EN=0 | All instructions | TRACKER=x, SUPPRESS=x, ENDBR_EN=0 |
| TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1 | Far CALL/JMP, INTn/INT3/INTO | TRACKER=WAIT FOR ENDBRANCH, SUPPRESS=0, ENDBR_EN=1 |
| | ENDBRANCH instruction Successful ENCLU[ERESUME] | TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1 |
| | All other instructions including indirect CALL/JMP | TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1 |
| TRACKER=1, SUPPRESS=1, ENDBR_EN=1 (This state cannot be reached by hardware and is disallowed as a valid state by WRMSR/XRSTORS/VM entry/VM exit) | NA | NA |
CONTROL-FLOW ENFORCEMENT TECHNOLOGY (CET)

17.3.5 INT3 Treatment

INT3 are treated special in the WAIT_FOR_ENDBRANCH state. Occurrence of INT3 do not move the tracker to IDLE but instead the #BP trap from the INT3 instructions respectively is delivered as a higher priority event than the #CP exception due to missing ENDBRANCH.

Inside an enclave, INT3 delivers a fault-class exception and thus does not require the CPL to be less than DPL in the IDT gate 3. Following opt-out entry, the instruction delivers #UD. Following opt-in entry, INT3 delivers #BP. The special treatment of INT3 in WAIT_FOR_ENDBRANCH state does not apply in enclave mode following opt-out entry.

17.3.6 Legacy Compatibility Treatment

ENDBRANCH legacy compatibility treatment allows a CET enabled program to be used with legacy software that was not compiled / instrumented with ENDBRANCH. A CET enabled program enters legacy compatibility treatment when all of the below conditions are met.

1. Legacy compatibility configuration is enabled in this CPL class by setting the LEG_IW_EN bit in IA32_U_CET/IA32_S_CET.
2. Control transfer is performed using an indirect CALL/JMP without no-track prefix to an instruction other than ENDBRANCH.
3. The legacy code page bitmap is setup to indicate that the target of the control transfer is a legacy code page.

The legacy code page bitmap is a data structure in program memory that is used by the hardware to determine if the code page to which a legacy transfer is being performed is allowed. The access rights for accessing the legacy code page bitmap is determined by the current privilege level (CPL). The legacy code page bitmap is expected to be setup as a read-only data structure.

When a matching ENDBRANCH instruction is not decoded at the target of an indirect CALL/JMP when required, the processor performs the below actions.

CET indirect branch tracking state machine violation event handler:

If LEG_IW_EN == 1

LA = LIP;

IF ENCLAVE_MODE == 1

LA = LA - SECS.BASEADDR;

ENDIF

(* Load byte from bitmap. Address-size attribute for this load is 64 bits if IA32_EFER.LMA is 1 and is 32 bits when IA32_EFER.LMA is 0 *)

IF (IA32_EFER.LMA & CS.L) == 0

BITMAP_BYTE = load 1 byte from address (BITMAP_BASE + LA[31:15])
ELSE IF (CR4.LA57 == 0)

BITMAP_BYTE = load 1 byte from address (BITMAP_BASE + LA[47:15])
ELSE

BITMAP_BYTE = load 1 byte from address (BITMAP_BASE + LA[56:15])
FI;

IF BITMAP_BYTE & (1 << LA[14:12]) == 0 then Deliver #CP(ENDBRANCH) fault

IF CPL = 3

IA32_U_CET.TRACKER = IDLE
IA32_U_CET.SUPPRESS = IA32_U_CET.SUPPRESS_DIS == 0 ? 1 : 0
ELSE

IA32_S_CET.TRACKER = IDLE
IA32_S_CET.SUPPRESS = IA32_S_CET.SUPPRESS_DIS == 0 ? 1 : 0
ENDIF

Restart the instruction (handle all arch. consistency around MOV SS state machines, STI etc.) without opening up interrupt/trap window.

ELSE

Deliver #CP(ENDBRANCH) Fault

17-12 Vol. 1
Faults/traps in pseudocode are delivered normally (e.g., #PF, EPT violation). On a fault, the active tracker holds the last value (WAIT_FOR_ENDBRANCH) and the address saved on the stack is the current IP (instruction that wasn’t the ENDBRANCH).

The CET indirect branch tracking state machine is suppressed in legacy compatibility mode if the SUPPRESS_DIS control bit is 0.

Once the CET indirect branch tracking state machine has been suppressed, subsequent indirect CALL/JMP are not tracked for termination instruction.

Once CET indirect branch tracking has been suppressed, subsequent execution of ENDBRANCH instructions will do the following (see the ENDBR32 and ENDBR64 instructions in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A for details).

IF EndbranchEnabled(CPL) == 0
    NOP
ELSE
    SUPPRESS = 0
    TRACKER = IDLE
ENDIF

17.3.6.1 Legacy Code Page Bitmap Format

The legacy code page bitmap is a flat bitmap whose linear address is pointed to by the EB_LEG_BITMAP_BASE. Each bit in the bitmap represents a 4K page in linear memory. If the bit is 1 it indicates that the corresponding code page is a legacy code page; else it is a CET-enabled code page.

The processor uses the linear address of the instruction to which legacy transfer was attempted to lookup the bitmap. Bits of the linear address used as index in the bitmap are as follows.

- In legacy and compatibility mode: Bits 31:12.
- In 64-bit mode (EFER.LMA=1 and CS.L=1): Bits 47:12.

17.3.7 Other Considerations

17.3.7.1 Intel® Transactional Synchronization Extensions (Intel® TSX) Interactions

The XBEGIN instruction encodes the relative offset to the abort handler and hence the fallback to the abort handler can be considered as a “direct” branch and the abort handler does not need to have an ENDBRANCH.

CET continues to enforce indirect CALL/JMP tracking within a transaction. Legacy compatibility treatment inside a transaction functions normally. If a transaction abort occurs then the processor sets the state of the indirect branch tracker to IDLE and not-suppressed.

17.3.7.2 #CP(ENDBRANCH) Priority w.r.t #NM and #UD

#NM, #UD and #CP(ENDBRANCH) are opcode based faults. However, #CP(ENDBRANCH) is in a higher priority class than #NM and #UD as CET architecturally requires an ENDBRANCH at target of indirect CALL/JMP.

17.3.7.3 #CP(ENDBRANCH) Priority w.r.t #BP and #DB

Debug Exceptions priority is as follows.

- Traps delivered before any #CP(ENDBRANCH) fault: Data breakpoint trap, IO breakpoint trap single step trap, task switch trap.
- Code Breakpoint fault detected before instruction decode and delivered before #CP(ENDBRANCH).
- General-detect (GD) exception condition fault: Lower priority than #CP(ENDBRANCH).
CONTROL-FLOW ENFORCEMENT TECHNOLOGY (CET)

- On IRET back from #DB/#BP, the source indirect branch tracker becomes active if enabled and not suppressed. INT3 does not cause #CP(ENDBRANCH) to support debugger usage of replacing bytes of ENDBRANCH with INT3 to set breakpoints. INT3 at target of a CALL-JMP(indirect) cause #BP(INT3) instead of #CP(ENDBRANCH), #CP(ENDBRANCH) fault is delayed. #BP caused by INT3 treated like other events that are higher priority than CET fault. On IRET back from #BP the source indirect tracker becomes active if enabled and not suppressed.

17.3.8 Constraining Speculation after Missing ENDBRANCH

When the CET tracker is in the WAIT_FOR_ENDBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH. This means that when indirect branch tracking is enabled and not suppressed, the instructions at the target of a near indirect JMP/CALL without the no-track prefix will only speculatively execute if there is an ENDBRANCH at the target. This can constrain both attacker controlled prediction as well as attacker controlled jump redirection attacks on near indirect JMPs/CALLs by reducing the gadgets available to an attacker using these techniques. Early implementations of CET may limit the speculative execution to a small number of instructions (less than 8, with no more than 5 loads) past a missing ENDBRANCH, while later implementations will completely block the speculative execution of instructions after a missing ENDBRANCH.

This mechanism also limits or blocks speculation of the next sequential instructions after an indirect JMP or CALL, presuming the JMP/CALL puts the CET tracker into the WAIT_FOR_ENDBRANCH state and the next sequential instruction is not an ENDBRANCH.

17.4 INTEL® TRUSTED EXECUTION TECHNOLOGY (INTEL® TXT) INTERACTIONS

GETSEC[ENTERACCS] and GETSEC[SENTER] clear CR4.CET, and it is not restored when these instructions complete.

GETSEC[EXITAC] will cause #GP(0) fault if CR4.CET is set.
18.1 INTRODUCTION

Intel® Advanced Matrix Extensions (Intel® AMX) is a new 64-bit programming paradigm consisting of two components: a set of 2-dimensional registers (tiles) representing sub-arrays from a larger 2-dimensional memory image, and an accelerator able to operate on tiles, the first implementation is called TMUL (tile matrix multiply unit).

An Intel AMX implementation enumerates to the programmer how the tiles can be programmed by providing a palette of options. Two palettes are supported; palette 0 represents the initialized state, and palette 1 consists of 8 KB of storage spread across 8 tile registers named TMM0..TMM7. Each tile has a maximum size of 16 rows x 64 bytes, (1 KB), however the programmer can configure each tile to smaller dimensions appropriate to their algorithm. The tile dimensions supplied by the programmer (rows and bytes_per_row, i.e., colsb) are metadata that drives the execution of tile and accelerator instructions. In this way, a single instruction can launch autonomous multi-cycle execution in the tile and accelerator hardware. The palette value (palette_id) and metadata are held internally in a tile related control register (TILECFG). The TILECFG contents will be commensurate with that reported in the palette_table (see “CPUID—CPU Identification” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A for a description of the available parameters).

Intel AMX is an extensible architecture. New accelerators can be added, or the TMUL accelerator may be enhanced to provide higher performance. In these cases, the state (TILEDATA) provided by tiles may need to be made larger, either in one of the metadata dimensions (more rows or colsb) and/or by supporting more tile registers (names). The extensibility is carried out by adding new palette entries describing the additional state. Since execution is driven through metadata, an existing Intel AMX binary could take advantage of larger storage sizes and higher performance TMUL units by selecting the most powerful palette indicated by CPUID and adjusting loop and pointer updates accordingly.

![Figure 18-1. Intel® AMX Architecture](image-url)
Figure 18-1 shows a conceptual diagram of the Intel AMX architecture. An Intel architecture host drives the algorithm, the memory blocking, loop indices and pointer arithmetic. Tile loads and stores and accelerator commands are sent to multi-cycle execution units. Status, if required, is reported back. Intel AMX instructions are synchronous in the Intel architecture instruction stream and the memory loaded and stored by the tile instructions is coherent with respect to the host’s memory accesses. There are no restrictions on interleaving of Intel architecture and Intel AMX code or restrictions on the resources the host can use in parallel with Intel AMX (e.g., Intel AVX-512). There is also no architectural requirement on the Intel architecture compute capability of the Intel architecture host other than it supports 64-bit mode.

Intel AMX instructions use new registers and inherit basic behavior from Intel architecture in the same manner that Intel SSE and Intel AVX did. Tile instructions include loads and stores using the traditional Intel architecture register set as pointers. The TMUL instruction set (defined to be CPUID bits AMX-BF16 and AMX-INT8) only supports reg-reg operations.

TILECFG is programmed using the LDTILECFG instruction. The selected palette defines the available storage and general configuration while the rest of the memory data specifies the number of rows and column bytes for each tile. Consistency checks are performed to ensure the TILECFG matches the restrictions of the palette. A General Protection fault (#GP) is reported if the LDTILECFG fails consistency checks. A successful load of TILECFG with a palette_id other than 0 is represented in this document with TILES_CONFIGURED = 1. When the TILECFG is initialized (palette_id = 0), it is represented in the document as TILES_CONFIGURED = 0. Nearly all Intel AMX instructions will generate a #UD exception if TILES_CONFIGURED is not equal to 1; the exceptions are those that do TILECFG maintenance: LDTILECFG, STTILECFG, and TILERELEASE.

If a tile is configured to contain M rows by N column bytes, LDTILECFG will ensure that the metadata values are appropriate to the palette (e.g., that $M \leq 16$ and $N \leq 64$ for palette 1). The four $M$ and $N$ values can all be different as long as they adhere to the restrictions of the palette. Further dynamic checks are done in the tile and the TMUL instruction set to deal with cases where a legally configured tile may be inappropriate for the instruction operation. Tile registers can be set to ‘invalid’ by configuring the rows and colsb to ‘0’.

Tile loads and stores are strided accesses from the application memory to packed rows of data. Algorithms are expressed assuming row major data layout. Column major users should translate the terms according to their orientation.

TILELOAD* and TILESTORE* instructions are restartable and can handle (up to) 2*rows page faults per instruction. Restartability is provided by a start_row parameter in the TILECFG register.

The TMUL unit is conceptually a grid of fused multiply-add units able to read and write tiles. The dimensions of the TMUL unit (tmul_maxk and tmul_maxn) are enumerated similar to the maximum dimensions of the tiles (see “CPUID—CPU Identification” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A for details).

The matrix multiplications in the TMUL instruction set compute $C[M][N] += A[M][K] * B[K][N]$. The $M$, $N$, and $K$ values will cause the TMUL instruction set to generate a #UD exception if the dimensions do not match for matrix multiply or do not match the palette.

In Figure 18-2, the number of rows in tile B matches the $K$ dimension in the matrix multiplication pseudocode. $K$ dimensions smaller than that enumerated in the TMUL grid are also possible and any additional computation the TMUL unit can support will not affect the result.

The number of elements specified by colsb of the B matrix is also less than or equal to tmul_maxn. Any remaining values beyond that specified by the metadata will be set to zero.
The XSAVE feature set supports context management of the new state defined for Intel AMX. This support is described in Section 18.2.

18.1.1 Tile Architecture Details

The supported parameters for the tile architecture are reported via CPUID; this includes information about how the number of tile registers (max_names) can be configured (the palette). Configuring the tile architecture is intended to be done once when entering a region of tile code using the LDTILECFG instruction specifying the selected palette and describing in detail the configuration for each tile. Incorrect assignments will result in a General Protection fault (#GP). Successful LDTILECFG initializes (zeroes) TILEDATA.

Exiting a tile region is done with the TILERELEASE instruction. It takes no parameters and invalidates all tiles (indicating that the data no longer needs any saving or restoring). Essentially, it is an optimization of LDTILECFG with an implicit palette of 0.

For applications that execute consecutive Intel AMX regions with differing configurations, TILERELEASE is not required between them since the second LDTILECFG will clear all the data while loading the new configuration. There is no instruction set support for automatic nesting of tile regions, though with sufficient effort software can accomplish this by saving and restoring TILEDATA and TILECFG either through the XSAVE architecture or the Intel AMX instructions.

The tile architecture boots in its INIT state, with TILECFG and TILEDATA set to zero. A successfully executing LDTILECFG instruction to a non-zero palette sets the TILES_CONFIGURED=1, indicating the TILECFG is not in the INIT state. The TILERELEASE instruction sets TILES_CONFIGURED = 0 and initializes (zeroes) TILEDATA.

To facilitate handling of tile configuration data, there is a STTILECFG instruction. If the tile configuration is in the INIT state (TILES_CONFIGURED == 0), then STTILECFG will write 64 bytes of zeros. Otherwise STTILECFG will store the TILECFG to memory in the format used by LDTILECFG.
18.1.2 TMUL Architecture Details

The supported parameters for the TMUL architecture are reported via CPUID; see “CPUID—CPU Identification” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for details. These parameters include a maximum height (tmul_maxk) and a maximum SIMD dimension (tmul_maxn). The metadata that accompanies the srcdest, src1, and src2 tiles to the TMUL unit will be dynamically checked to see that they match the TMUL unit support for the data type and match the requirements of a meaningful matrix multiplication.

Figure 18-3 shows an example of the inner loop of an algorithm of using the TMUL architecture to compute a matrix multiplication. In this example, we use two result tiles, tmm0 and tmm1, from matrix C to accumulate the intermediate results. One tile from the A matrix (tmm2) is re-used twice as we multiply it by two tiles from the B matrix. The algorithm then advances pointers to load a new A tile and two new B tiles from the directions indicated by the arrows. An outer loop, not shown, adjusts the pointers for the C tiles.

![Figure 18-3. Matrix Multiply C+= A*B](image)

```c
LDTILECFG [rax]
// assume some outer loops driving the cache tiling (not shown)
{
    TILELOADD tmm0, [rsi+rdi] // srcdst, RSI points to C, RDI is strided value
    TILELOADD tmm1, [rsi+rdi+N] // second tile of C, unrolling in SIMD dimension N
    MOV r14, 0
    LOOP:
        TILELOADD tmm2, [r8+r9] // src2 is strided load of A, reused for 2 TMUL instr.
        TILELOADD tmm3, [r10+r11] // src1 is strided load of B
        TDPBUSD tmm0, tmm2, tmm3 // update left tile of C
        TILELOADD tmm3, [r10+r11+N] // src1 loaded with B from next rightmost tile
        TDPBUSD tmm1, tmm2, tmm3 // update right tile of C
        ADD r8, K                  // update pointers by constants known outside of loop
        ADD r10, K*r11
        ADD r14, K
        CMP r14, LIMIT
        JNE LOOP
    TILESTORED [rsi+rdi], tmm0 // update the C matrix in memory
    TILESTORED [rsi+rdi+M], tmm1
} // end of outer loop
TILERLEASE // return tiles to INIT state
```

18.1.3 Handling of Tile Row and Column Limits

Intel AMX operations will zero any rows and any columns beyond the dimensions specified by TILECFG. Tile operations will zero the data beyond the configured number of column bytes as each row is written. For example, with 64-byte rows and a tile configured with 10 rows and 48 columns, an operation writing dword elements would write...
each of the first 10 rows with 48 bytes of output/result data and zero the remaining 16 bytes in each row. Tile operations also fully zero any rows after the first 10 configured rows. When using a 1 KByte tile with 64-byte rows, there would be 16 rows, so in this example, the last 6 rows would also be zeroed.

Intel AMX instructions will always obey the metadata on reads and the zeroing rules on writes, and so a subsequent XSAVE would see zeros in the appropriate locations. Tiles that are not written by Intel AMX instructions between XRSTOR and XSAVE will write back with the same image they were loaded with regardless of the value of TILECFG.

18.1.4 Exceptions and Interrupts

Tile instructions are restartable so that operations that access strided memory can restart after page faults. To support restarting instructions after these events, the instructions store information in the TILECFG.start_row register. TILECFG.start_row indicates the row that should be used for restart; i.e., it indicates next row after the rows that have already been successfully loaded (on a TILELOAD) or written to memory (on a TILESTORE) and prevents repeating work that was successfully done.

The TMUL instruction set is not sensitive to the TILECFG.start_row value; this is due to there not being TMUL instructions with memory operands or any restartable faults.

18.2 RECOMMENDATIONS FOR SYSTEM SOFTWARE

Intel AMX is an XSAVE-enabled feature, meaning that it requires use of the XSAVE feature set for their enabling. Specifically, Intel AMX instructions and state are available only if system software has set CR4.OSXSAVE and also set XCR0[18:17] to 11B. In addition, use of Intel AMX instructions is disabled if system software has used extended feature disable (XFD) and set either IA32_XFD[17] or IA32_XFD[18] to 1. See Chapter 13, "Managing State Using the XSAVE Feature Set," for more details.

NOTE
The first processors implementing Intel AMX will support setting IA32_XFD[18] but not IA32_XFD[17].

Once Intel AMX has been enabled, system software can disable it by clearing XCR0[18:17], by clearing CR4.OSXSAVE, or by setting either IA32_XFD[17] or IA32_XFD[18]. Before doing do, system software should first initialize AMX state (e.g., by executing TILERELEASE); maintaining AMX state in a non-initialized state may have negative power and performance implications. In addition, software should not rely on the state of the tile data after setting IA32_XFD[17] or IA32_XFD[18]; software should always reload or reinitialize the tile data after clearing IA32_XFD[17] and IA32_XFD[18].

System software should not use XFD to implement a "lazy restore" approach to management of the TILEDATA state component. This approach will not operate correctly for a variety of reasons. One is that the LDTILECFG and TILERELEASE instructions initialize TILEDATA and do not cause an #NM exception. Another is that an execution of XSAVE, XSAVEC, XSAVEXOPT, or XSAVES by a user thread will save TILEDATA as initialized instead of the data expected by the user thread.

18.3 IMPLEMENTATION PARAMETERS

The parameters are reported via CPUID leaf 1DH. Index 0 reports all zeros for all fields.
define palette_table[id]:
    uint16_t total_tile_bytes
    uint16_t bytes_per_tile
    uint16_t bytes_per_row
    uint16_t max_names
    uint16_t max_rows

The tile parameters are set by LDTILECFG or XRSTOR* of TILECFG:
define tile[tid]:
    byte rows
    word colsb // bytes_per_row
    bool valid

18.4 HELPER FUNCTIONS

The helper functions used in Intel AMX instructions are defined below.

define write_row_and_zero(treg, r, data, nbytes):
    for j in 0 ... nbytes-1:
        treg.row[r].byte[j] := data.byte[j]

    // zero the rest of the row
    for j in nbytes ... palette_table[tilecfg.palette_id].bytes_per_row-1:
        treg.row[r].byte[j] := 0

define zero_upper_rows(treg, r):
    for i in r ... palette_table[tilecfg.palette_id].max_rows-1:
        for j in 0 ... palette_table[tilecfg.palette_id].bytes_per_row-1:
            treg.row[i].byte[j] := 0

define zero_tilecfg_start():
    tilecfg.start_row := 0

define zero_all_tile_data():
    if XCR0[TILEDATA]:
        b := CPUID(0xD,TILEDATA).EAX // size of feature
        for j in 0 ... b:
            TILEDATA.byte[j] := 0
define xcr0 Supports_palette(palette_id):
    if palette_id == 0:
        return 1
    elif palette_id == 1:
        if xcr0[TILECFG] and xcr0[TILEDATA]:
            return 1
    return 0
In addition to transferring data to and from external memory, IA-32 processors can also transfer data to and from input/output ports (I/O ports). I/O ports are created in system hardware by circuitry that decodes the control, data, and address pins on the processor. These I/O ports are then configured to communicate with peripheral devices. An I/O port can be an input port, an output port, or a bidirectional port. Some I/O ports are used for transmitting data, such as to and from the transmit and receive registers, respectively, of a serial interface device. Other I/O ports are used to control peripheral devices, such as the control registers of a disk controller.

This chapter describes the processor’s I/O architecture. The topics discussed include:

- I/O port addressing.
- I/O instructions.
- I/O protection mechanism.

19.1 I/O PORT ADDRESSING

The processor permits applications to access I/O ports in either of two ways:

- Through a separate I/O address space.
- Through memory-mapped I/O.

Accessing I/O ports through the I/O address space is handled through a set of I/O instructions and a special I/O protection mechanism. Accessing I/O ports through memory-mapped I/O is handled with the processor’s general-purpose move and string instructions, with protection provided through segmentation or paging. I/O ports can be mapped so that they appear in the I/O address space or the physical-memory address space (memory mapped I/O) or both.

One benefit of using the I/O address space is that writes to I/O ports are guaranteed to be completed before the next instruction in the instruction stream is executed. Thus, I/O writes to control system hardware cause the hardware to be set to its new state before any other instructions are executed. See Section 19.6, “Ordering I/O,” for more information on serializing of I/O operations.

19.2 I/O PORT HARDWARE

From a hardware point of view, I/O addressing is handled through the processor’s address lines. For the P6 family, Pentium 4, and Intel Xeon processors, the request command lines signal whether the address lines are being driven with a memory address or an I/O address; for Pentium processors and earlier IA-32 processors, the M/IO# pin indicates a memory address (1) or an I/O address (0). When the separate I/O address space is selected, it is the responsibility of the hardware to decode the memory-I/O bus transaction to select I/O ports rather than memory. Data is transmitted between the processor and an I/O device through the data lines.

19.3 I/O ADDRESS SPACE

The processor’s I/O address space is separate and distinct from the physical-memory address space. The I/O address space consists of \(2^{16}\) (64K) individually addressable 8-bit I/O ports, numbered 0 through FFFFH. I/O port addresses 0F8H through 0FFH are reserved. Do not assign I/O ports to these addresses. The result of an attempt to address beyond the I/O address space limit of FFFFH is implementation-specific; see the Developer’s Manuals for specific processors for more details.

Any two consecutive 8-bit ports can be treated as a 16-bit port, and any four consecutive ports can be a 32-bit port. In this manner, the processor can transfer 8, 16, or 32 bits to or from a device in the I/O address space. Like words in memory, 16-bit ports should be aligned to even addresses (0, 2, 4, ...) so that all 16 bits can be transferred in a
single bus cycle. Likewise, 32-bit ports should be aligned to addresses that are multiples of four (0, 4, 8, ...). The processor supports data transfers to unaligned ports, but there is a performance penalty because one or more extra bus cycle must be used.

The exact order of bus cycles used to access unaligned ports is undefined and is not guaranteed to remain the same in future IA-32 processors. If hardware or software requires that I/O ports be written to in a particular order, that order must be specified explicitly. For example, to load a word-length I/O port at address 2H and then another word port at 4H, two word-length writes must be used, rather than a single doubleword write at 2H.

Note that the processor does not mask parity errors for bus cycles to the I/O address space. Accessing I/O ports through the I/O address space is thus a possible source of parity errors.

19.3.1 Memory-Mapped I/O

I/O devices that respond like memory components can be accessed through the processor’s physical-memory address space (see Figure 19-1). When using memory-mapped I/O, any of the processor’s instructions that reference memory can be used to access an I/O port located at a physical-memory address. For example, the MOV instruction can transfer data between any register and a memory-mapped I/O port. The AND, OR, and TEST instructions may be used to manipulate bits in the control and status registers of a memory-mapped peripheral device.

Certain instructions may take an exception or VM exit after completing a memory access (either a read or a write) to a memory-mapped I/O address. This exception or VM exit could be due to the instruction performing multiple memory accesses (e.g., MOVs, PUSH mem, POP mem, PUSHAD, etc.) or could be due to the ordering of exceptions or VM exits within the instruction (e.g., a DIV mem that takes a #DE or a CALL that causes a task switch VM exit). If software later re-executes that instruction (e.g., after an IRET or VMRESUME), the MMIO (memory-mapped I/O) access may occur again. If the memory-mapped I/O access has a side-effect, that side-effect may be executed each time the memory-mapped I/O access occurs. If that is problematic, software must ensure that exceptions or VM exits do not occur after accessing the MMIO.

When using memory-mapped I/O, caching of the address space mapped for I/O operations must be prevented. With the Pentium 4, Intel Xeon, and P6 family processors, caching of I/O accesses can be prevented by using memory type range registers (MTRRs) to map the address space used for the memory-mapped I/O as uncacheable (UC). See Chapter 12, “Memory Cache Control,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a complete discussion of the MTRRs.

The Pentium and Intel486 processors do not support MTRRs. Instead, they provide the KEN# pin, which when held inactive (high) prevents caching of all addresses sent out on the system bus. To use this pin, external address decoding logic is required to block caching in specific address spaces.
All the IA-32 processors that have on-chip caches also provide the PCD (page-level cache disable) flag in page table and page directory entries. This flag allows caching to be disabled on a page-by-page basis. See “Page-Directory and Page-Table Entries” in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

### 19.4 I/O INSTRUCTIONS

The processor’s I/O instructions provide access to I/O ports through the I/O address space. (These instructions cannot be used to access memory-mapped I/O ports.) There are two groups of I/O instructions:

- Those that transfer a single item (byte, word, or doubleword) between an I/O port and a general-purpose register.
- Those that transfer strings of items (strings of bytes, words, or doublewords) between an I/O port and memory.

The register I/O instructions `IN` (input from I/O port) and `OUT` (output to I/O port) move data between I/O ports and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The address of the I/O port can be given with an immediate value or a value in the DX register.

The string I/O instructions `INS` (input string from I/O port) and `OUTS` (output string to I/O port) move data between an I/O port and a memory location. The address of the I/O port being accessed is given in the DX register; the source or destination memory address is given in the DS:ESI or ES:EDI register, respectively.

When used with the repeat prefix REP, the INS and OUTS instructions perform string (or block) input or output operations. The repeat prefix REP modifies the INS and OUTS instructions to transfer blocks of data between an I/O port and memory. Here, the ESI or EDI register is incremented or decremented (according to the setting of the DF flag in the EFLAGS register) after each byte, word, or doubleword is transferred between the selected I/O port and memory.

See the references for IN, INS, OUT, and OUTS in Chapter 3 and Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, for more information on these instructions.

### 19.5 PROTECTED-MODE I/O

When the processor is running in protected mode, the following protection mechanisms regulate access to I/O ports:

- When accessing I/O ports through the I/O address space, two protection devices control access:
  - The I/O privilege level (IOPL) field in the EFLAGS register.
  - The I/O permission bit map of a task state segment (TSS).

- When accessing memory-mapped I/O ports, the normal segmentation and paging protection and the MTRRs (in processors that support them) also affect access to I/O ports. See Chapter 5, “Protection,” and Chapter 12, "Memory Cache Control," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a complete discussion of memory protection.

The following sections describe the protection mechanisms available when accessing I/O ports in the I/O address space with the I/O instructions.

#### 19.5.1 I/O Privilege Level

In systems where I/O protection is used, the IOPL field in the EFLAGS register controls access to the I/O address space by restricting use of selected instructions. This protection mechanism permits the operating system or executive to set the privilege level needed to perform I/O. In a typical protection ring model, access to the I/O address space is restricted to privilege levels 0 and 1. Here, the kernel and the device drivers are allowed to perform I/O, while less privileged device drivers and application programs are denied access to the I/O address space. Application programs must then make calls to the operating system to perform I/O.
The following instructions can be executed only if the current privilege level (CPL) of the program or task currently executing is less than or equal to the IOPL: IN, INS, OUT, OUTS, CLI (clear interrupt-enable flag), and STI (set interrupt-enable flag). These instructions are called **I/O sensitive** instructions, because they are sensitive to the IOPL field. Any attempt by a less privileged program or task to use an I/O sensitive instruction results in a general-protection exception (#GP) being signaled. Because each task has its own copy of the EFLAGS register, each task can have a different IOPL.

The I/O permission bit map in the TSS can be used to modify the effect of the IOPL on I/O sensitive instructions, allowing access to some I/O ports by less privileged programs or tasks (see Section 19.5.2, "I/O Permission Bit Map").

A program or task can change its IOPL only with the POPF and IRET instructions; however, such changes are privileged. No procedure may change the current IOPL unless it is running at privilege level 0. An attempt by a less privileged procedure to change the IOPL does not result in an exception; the IOPL simply remains unchanged.

The POPF instruction also may be used to change the state of the IF flag (as can the CLI and STI instructions); however, the POPF instruction in this case is also I/O sensitive. A procedure may use the POPF instruction to change the setting of the IF flag only if the CPL is less than or equal to the current IOPL. An attempt by a less privileged procedure to change the IF flag does not result in an exception; the IF flag simply remains unchanged.

### 19.5.2 I/O Permission Bit Map

The I/O permission bit map is a device for permitting limited access to I/O ports by less privileged programs or tasks and for tasks operating in virtual-8086 mode. The I/O permission bit map is located in the TSS (see Figure 19-2) for the currently running task or program. The address of the first byte of the I/O permission bit map is given in the I/O map base address field of the TSS. The size of the I/O permission bit map and its location in the TSS are variable.

![Figure 19-2. I/O Permission Bit Map](image)

Because each task has its own TSS, each task has its own I/O permission bit map. Access to individual I/O ports can thus be granted to individual tasks.

If in protected mode and the CPL is less than or equal to the current IOPL, the processor allows all I/O operations to proceed. If the CPL is greater than the IOPL or if the processor is operating in virtual-8086 mode, the processor checks the I/O permission bit map to determine if access to a particular I/O port is allowed. Each bit in the map corresponds to an I/O port byte address. For example, the control bit for I/O port address 29H in the I/O address space is found at bit position 1 of the sixth byte in the bit map. Before granting I/O access, the processor tests all the bits corresponding to the I/O port being addressed. For a doubleword access, for example, the processors tests the four bits corresponding to the four adjacent 8-bit port addresses. If any tested bit is set, a general-protection exception (#GP) is signaled. If all tested bits are clear, the I/O operation is allowed to proceed.
Because I/O port addresses are not necessarily aligned to word and doubleword boundaries, the processor reads two bytes from the I/O permission bit map for every access to an I/O port. To prevent exceptions from being generated when the ports with the highest addresses are accessed, an extra byte needs to be included in the TSS immediately after the table. This byte must have all of its bits set, and it must be within the segment limit.

It is not necessary for the I/O permission bit map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had set bits in the map. For example, if the TSS segment limit is 10 bytes past the bit-map base address, the map has 11 bytes and the first 80 I/O ports are mapped. Higher addresses in the I/O address space generate exceptions.

If the I/O bit map base address is greater than or equal to the TSS segment limit, there is no I/O permission map, and all I/O instructions generate exceptions when the CPL is greater than the current IOPL.

19.6 ORDERING I/O

When controlling I/O devices it is often important that memory and I/O operations be carried out in precisely the order programmed. For example, a program may write a command to an I/O port, then read the status of the I/O device from another I/O port. It is important that the status returned be the status of the device after it receives the command, not before.

When using memory-mapped I/O, caution should be taken to avoid situations in which the programmed order is not preserved by the processor. To optimize performance, the processor allows cacheable memory reads to be reordered ahead of buffered writes in most situations. Internally, processor reads (cache hits) can be reordered around buffered writes. When using memory-mapped I/O, therefore, it is possible that an I/O read might be performed before the memory write of a previous instruction. The recommended method of enforcing program ordering of memory-mapped I/O accesses with the Pentium 4, Intel Xeon, and P6 family processors is to use the MTRRs to make the memory mapped I/O address space uncacheable; for the Pentium and Intel486 processors, either the KEN# pin or the PCD flags can be used for this purpose (see Section 19.3.1, “Memory-Mapped I/O”).

When the target of a read or write is in an uncacheable region of memory, memory reordering does not occur externally at the processor’s pins (that is, reads and writes appear in-order). Designating a memory mapped I/O region of the address space as uncacheable ensures that reads and writes of I/O devices are carried out in program order. See Chapter 12, “Memory Cache Control,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on using MTRRs.

Another method of enforcing program order is to insert one of the serializing instructions, such as the CPUID instruction, between operations. See Chapter 9, “Multiple-Processor Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on serialization of instructions.

It should be noted that the chipset being used to support the processor (bus controller, memory controller, and/or I/O controller) may post writes to uncacheable memory which can lead to out-of-order execution of memory accesses. In situations where out-of-order processing of memory accesses by the chipset can potentially cause faulty memory-mapped I/O processing, code must be written to force synchronization and ordering of I/O operations. Serializing instructions can often be used for this purpose.

When the I/O address space is used instead of memory-mapped I/O, the situation is different in two respects:

- The processor never buffers I/O writes. Therefore, strict ordering of I/O operations is enforced by the processor. (As with memory-mapped I/O, it is possible for a chipset to post writes in certain I/O ranges.)
- The processor synchronizes I/O instruction execution with external bus activity (see Table 19-1).
### Table 19-1. I/O Instruction Serialization

<table>
<thead>
<tr>
<th>Instruction Being Executed</th>
<th>Processor Delays Execution of ...</th>
<th>Until Completion of ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Current Instruction?</td>
<td>Next Instruction?</td>
</tr>
<tr>
<td>IN</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INS</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>REP INS</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>OUTS</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>REP OUTS</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
When writing software intended to run on IA-32 processors, it is necessary to identify the type of processor present in a system and the processor features that are available to an application.

### 20.1 USING THE CPUID INSTRUCTION

Use the CPUID instruction for processor identification in the Pentium M processor family, Pentium 4 processor family, Intel Xeon processor family, P6 family, Pentium processor, and later Intel486 processors. This instruction returns the family, model, and (for some processors) a brand string for the processor that executes the instruction. It also indicates the features that are present in the processor and gives information about the processor's caches and TLB.

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. The CPUID instruction will cause the invalid opcode exception (#UD) if executed on a processor that does not support it.

To obtain processor identification information, a source operand value is placed in the EAX register to select the type of information to be returned. When the CPUID instruction is executed, selected information is returned in the EAX, EBX, ECX, and EDX registers. For a complete description of the CPUID instruction, tables indicating values returned, and example code, see CPUID—CPU Identification in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

#### 20.1.1 Notes on Where to Start

The following guidelines are among the most important, and should always be followed when using the CPUID instruction to determine available features:

- Always begin by testing for the "GenuineIntel," message in the EBX, EDX, and ECX registers when the CPUID instruction is executed with EAX equal to 0. If the processor is not genuine Intel, the feature identification flags may have different meanings than are described in Intel documentation.
- Test feature identification flags individually and do not make assumptions about undefined bits.

#### 20.1.2 Identification of Earlier IA-32 Processors

The CPUID instruction is not available in earlier IA-32 processors up through the earlier Intel486 processors. For these processors, several other architectural features can be exploited to identify the processor.

The settings of bits 12 and 13 (IOPL), 14 (NT), and 15 (reserved) in the EFLAGS register are different for Intel's 32-bit processors than for the Intel 8086 and Intel 286 processors. By examining the settings of these bits (with the PUSHF/PUSHFD and POPF/POPFD instructions), an application program can determine whether the processor is an 8086, Intel 286, or one of the Intel 32-bit processors:

- 8086 processor — Bits 12 through 15 of the EFLAGS register are always set.
- Intel 286 processor — Bits 12 through 15 are always clear in real-address mode.
- 32-bit processors — In real-address mode, bit 15 is always clear and bits 12 through 14 have the last value loaded into them. In protected mode, bit 15 is always clear, bit 14 has the last value loaded into it, and the IOPL bits depend on the current privilege level (CPL). The IOPL field can be changed only if the CPL is 0.

Other EFLAGS register bits that can be used to differentiate between the 32-bit processors:

- Bit 18 (AC) — Implemented only on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The inability to set or clear this bit distinguishes an Intel386 processor from the later IA-32 processors.
- Bit 21 (ID) — Determines if the processor is able to execute the CPUID instruction. The ability to set and clear this bit indicates that it is a Pentium 4, Intel Xeon, P6 family, Pentium, or later-version Intel486 processor.
To determine whether an x87 FPU or Numeric Processor Extension (NPX) is present in a system, applications can write to the x87 FPU status and control registers using the FNINIT instruction and then verify that the correct values are read back using the FNSTENV instruction.

After determining that an x87 FPU or NPX is present, its type can then be determined. In most cases, the processor type will determine the type of FPU or NPX; however, an Intel386 processor is compatible with either an Intel 287 or Intel 387 math coprocessor.

The method the coprocessor uses to represent $\infty$ (after the execution of the FINIT, FNINIT, or RESET instruction) indicates which coprocessor is present. The Intel 287 math coprocessor uses the same bit representation for $+\infty$ and $-\infty$; whereas, the Intel 387 math coprocessor uses different representations for $+\infty$ and $-\infty$. 
### A.1 EFLAGS AND INSTRUCTIONS

Table A-2 summarizes how the instructions affect the flags in the EFLAGS register. The following codes describe how the flags are affected.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
<th>TF</th>
<th>IF</th>
<th>DF</th>
<th>NT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td></td>
<td></td>
<td></td>
<td>TM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAD</td>
<td></td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAM</td>
<td></td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AAS</td>
<td></td>
<td></td>
<td></td>
<td>TM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>TM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>TM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>0</td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>BOUND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSF/BSR</td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSWAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT/BTS/BTR/BTC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>CALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>CLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CLTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>CMOVcc</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A-1. Codes Describing Flags

- **T**: Instruction tests flag.
- **M**: Instruction modifies flag (either sets or resets depending on operands).
- **0**: Instruction resets flag.
- **1**: Instruction sets flag.
- **—**: Instruction’s effect on flag is undefined.
- **R**: Instruction restores prior value of flag.
- **Blank**: Instruction does not affect flag.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>OF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
<th>TF</th>
<th>IF</th>
<th>DF</th>
<th>NT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPS</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>CMPXCHG</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPXCHG8B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMISD</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>0</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMISS</td>
<td>0</td>
<td>0</td>
<td>M</td>
<td>0</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cwd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMOVcc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOMI, FCOMIP, FUCOMI, FUCOMIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVLPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UCOMISD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UCOMISS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jcc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jcxz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lahf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDS/LES/LSS/LFS/LGS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lea</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGDT/LIDT/LLDT/LMSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A-2. EFLAGS Cross-Reference (Contd.)
Table A-2. EFLAGS Cross-Reference (Contd.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
<th>TF</th>
<th>IF</th>
<th>DF</th>
<th>NT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LODS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOPE/LOOPNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MONITOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MwAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV control, debug, test</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVsx/MOVzx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>O</td>
<td>M</td>
<td>M</td>
<td>—</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP/POPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>PUSH/PUSHA/PUSHF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL/RCR 1</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL/RCR count</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDMSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDPMC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDTSC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REP/REPE/REPNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL/ROR 1</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL/ROR count</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSM</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>SAHF</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>SAL/SAR/SHL/SHR 1</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>—</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>SAL/SAR/SHL/SHR count</td>
<td>—</td>
<td>M</td>
<td>M</td>
<td>—</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>SBB</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>SCAS</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>SETcc</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>SGDT/SIDT/SLDT/SMSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### EFLAGS CROSS-REFERENCE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OF</th>
<th>SF</th>
<th>ZF</th>
<th>AF</th>
<th>PF</th>
<th>CF</th>
<th>TF</th>
<th>IF</th>
<th>DF</th>
<th>NT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHLD/SHRD</td>
<td></td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST</td>
<td>0</td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERR/VERRw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WBINVD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRMSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XADD</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xchg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XLAT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>0</td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.1 CONDITION CODES

Table B-1 lists condition codes that can be queried using CMOVcc, FCMOVcc, Jcc, and SETcc. Condition codes refer to the setting of one or more status flags (CF, OF, SF, ZF, and PF) in the EFLAGS register. In the table below:

- The “Mnemonic” column provides the suffix (cc) added to the instruction to specify a test condition.
- “Condition Tested For” describes the targeted condition.
- “Instruction Subcode” provides the opcode suffix added to the main opcode to specify the test condition.
- “Status Flags Setting” describes the flag setting.

<table>
<thead>
<tr>
<th>Mnemonic (cc)</th>
<th>Condition Tested For</th>
<th>Instruction Subcode</th>
<th>Status Flags Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Overflow</td>
<td>0000</td>
<td>OF = 1</td>
</tr>
<tr>
<td>NO</td>
<td>No overflow</td>
<td>0001</td>
<td>OF = 0</td>
</tr>
<tr>
<td>B</td>
<td>Below</td>
<td>0010</td>
<td>CF = 1</td>
</tr>
<tr>
<td>C</td>
<td>Carry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAE</td>
<td>Neither above nor equal</td>
<td>0011</td>
<td>CF = 0</td>
</tr>
<tr>
<td>NB</td>
<td>Not below</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NC</td>
<td>Not carry</td>
<td>0100</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>AE</td>
<td>Above or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Equal</td>
<td>0101</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>Z</td>
<td>Zero</td>
<td>0110</td>
<td>(CF OR ZF) = 1</td>
</tr>
<tr>
<td>BE</td>
<td>Below or equal</td>
<td>0111</td>
<td>(CF OR ZF) = 0</td>
</tr>
<tr>
<td>NA</td>
<td>Not above</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Sign</td>
<td>1000</td>
<td>SF = 1</td>
</tr>
<tr>
<td>NS</td>
<td>No sign</td>
<td>1001</td>
<td>SF = 0</td>
</tr>
<tr>
<td>P</td>
<td>Parity</td>
<td>1010</td>
<td>PF = 1</td>
</tr>
<tr>
<td>PE</td>
<td>Parity even</td>
<td>1011</td>
<td>PF = 0</td>
</tr>
<tr>
<td>NP</td>
<td>No parity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td>Parity odd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Less</td>
<td>1100</td>
<td>(SF XOR OF) = 1</td>
</tr>
<tr>
<td>NGE</td>
<td>Neither greater nor equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NL</td>
<td>Not less</td>
<td>1101</td>
<td>(SF XOR OF) = 0</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal</td>
<td>1110</td>
<td>((SF XOR OF) OR ZF) = 1</td>
</tr>
<tr>
<td>NG</td>
<td>Not greater</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NLE</td>
<td>Neither less nor equal</td>
<td>1111</td>
<td>((SF XOR OF) OR ZF) = 0</td>
</tr>
</tbody>
</table>
Many of the test conditions are described in two different ways. For example, LE (less or equal) and NG (not greater) describe the same test condition. Alternate mnemonics are provided to make code more intelligible.

The terms “above” and “below” are associated with the CF flag and refer to the relation between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relation between two signed integer values.
C.1 OVERVIEW

This appendix shows which of the floating-point exceptions can be generated for:

- x87 FPU instructions — see Table C-2.
- Intel SSE instructions — see Table C-3.
- Intel SSE2 instructions — see Table C-4.
- Intel SSE3 instructions — see Table C-5.
- Intel SSE4 instructions — see Table C-6.

Table C-1 lists types of floating-point exceptions that potentially can be generated by the x87 FPU and by Intel SSE, SSE2, and SSE3 instructions.

The floating-point exceptions shown in Table C-1 (except for #D and #IS) are defined in IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. See Section 4.9.1, “Floating-Point Exception Conditions,” for a detailed discussion of floating-point exceptions.

C.2 X87 FPU INSTRUCTIONS

Table C-2 lists the x87 FPU instructions in alphabetical order. For each instruction, it summarizes the floating-point exceptions that the instruction can generate.

### Table C-2. Exceptions Generated with x87 FPU Floating-Point Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>#IS</th>
<th>#IA</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2XM1</td>
<td>Exponential</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>FABS</td>
<td>Absolute value</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FADD(P)</td>
<td>Add floating-point</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>FBLD</td>
<td>BCD load</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table C-2. Exceptions Generated with x87 FPU Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>#IS</th>
<th>#IA</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBSTP</td>
<td>BCD store and pop</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>FCHS</td>
<td>Change sign</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCLEX</td>
<td>Clear exceptions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMOVcc</td>
<td>Floating-point conditional move</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOM, FCOMP, FCOMPP</td>
<td>Compare floating-point</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOMI, FCOMIP, FCOMI, FUCOMIP</td>
<td>Compare floating-point and set EFLAGS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOS</td>
<td>Cosine</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>FDECSTP</td>
<td>Decrement stack pointer</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>FDIV(R)(P)</td>
<td>Divide floating-point</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FFREE</td>
<td>Free register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIADD</td>
<td>Integer add</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FICOM(P)</td>
<td>Integer compare</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIDIV</td>
<td>Integer divide</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIDIVR</td>
<td>Integer divide reversed</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FILD</td>
<td>Integer load</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIMUL</td>
<td>Integer multiply</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FINCSTP</td>
<td>Increment stack pointer</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FINIT</td>
<td>Initialize processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIST(P)</td>
<td>Integer store</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTTP</td>
<td>Truncate to integer (SSE3 instruction)</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>FISUB(R)</td>
<td>Integer subtract</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FLD extended or stack</td>
<td>Load floating-point</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD single or double</td>
<td>Load floating-point</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD1</td>
<td>Load + 1.0</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDcw</td>
<td>Load Control word</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FLDEnv</td>
<td>Load environment</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FLDL2E</td>
<td>Load log₂e</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDL2T</td>
<td>Load log₂10</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDLOG2</td>
<td>Load log₁₀2</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDLN2</td>
<td>Load logₑ2</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDPI</td>
<td>Load π</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDZ</td>
<td>Load + 0.0</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL(P)</td>
<td>Multiply floating-point</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FNOP</td>
<td>No operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPATAN</td>
<td>Partial arctangent</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPREM</td>
<td>Partial remainder</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPREM1</td>
<td>IEEE partial remainder</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table C-3. Exceptions Generated with Intel® SSE Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>#I</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS</td>
<td>Packed add.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ADDSS</td>
<td>Scalar add.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ANDNPS</td>
<td>Packed logical INVERT and AND.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ANDPS</td>
<td>Packed logical AND.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CMPPS</td>
<td>Packed compare.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CMPSS</td>
<td>Scalar compare.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
### Table C-3. Exceptions Generated with Intel® SSE Instructions (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>#I</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMISS</td>
<td>Scalar ordered compare lower SP FP numbers and set the status flags.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPi2PS</td>
<td>Convert two 32-bit signed integers from MM2/Mem to two SP FP.</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSP2PI</td>
<td>Convert lower two SP FP from XMM/Mem to two 32-bit signed integers in MM using rounding specified by MXCSR.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTI2SS</td>
<td>Convert one 32-bit signed integer from Integer Reg/Mem to one SP FP.</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SI</td>
<td>Convert one SP FP from XMM/Mem to one 32-bit signed integer using rounding mode specified by MXCSR, and move the result to an integer register.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTPS2PI</td>
<td>Convert two SP FP from XMM2/Mem to two 32-bit signed integers in MM1 using truncate.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTSS2SI</td>
<td>Convert lowest SP FP from XMM/Mem to one 32-bit signed integer using truncate, and move the result to an integer register.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS</td>
<td>Packed divide.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DIVSS</td>
<td>Scalar divide.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>LDMXCSR</td>
<td>Load control/status word.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXPS</td>
<td>Packed maximum.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXSS</td>
<td>Scalar maximum.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINPS</td>
<td>Packed minimum.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINSS</td>
<td>Scalar minimum.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVAPS</td>
<td>Move four packed SP values.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVHLPS</td>
<td>Move packed SP high to low.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVHPS</td>
<td>Move two packed SP values between memory and the high half of an XMM register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVLHPS</td>
<td>Move packed SP low to high.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVLPS</td>
<td>Move two packed SP values between memory and the low half of an XMM register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVMSKPS</td>
<td>Move sign mask to r32.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSS</td>
<td>Move scalar SP number between an XMM register and memory or a second XMM register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVUPS</td>
<td>Move unaligned packed data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPS</td>
<td>Packed multiply.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>MULSS</td>
<td>Scalar multiply.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ORPS</td>
<td>Packed OR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCPPS</td>
<td>Packed reciprocal.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCPSS</td>
<td>Scalar reciprocal.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSQRTPS</td>
<td>Packed reciprocal square root.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSQRTSS</td>
<td>Scalar reciprocal square root.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHUFPS</td>
<td>Shuffle.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPS</td>
<td>Square Root of the packed SP FP numbers.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSS</td>
<td>Scalar square root.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table C-4. Exceptions Generated with Intel® SSE2 Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>#I</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>STMXCSR</td>
<td>Store control/status word.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPS</td>
<td>Packed subtract.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SUBSS</td>
<td>Scalar subtract.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>UCOMISS</td>
<td>Unordered compare lower SP FP numbers and set the status flags.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNPCKHPS</td>
<td>Interleave SP FP numbers.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNPKLPS</td>
<td>Interleave SP FP numbers.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XORPS</td>
<td>Packed XOR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C.4 INTEL® SSE2 INSTRUCTIONS

Table C-4 lists the Intel SSE2 instructions with at least one of the following characteristics:
- Floating-point operands.
- Floating-point results.

For each instruction, the table summarizes the floating-point exceptions that the instruction can generate.

### Table C-4. Exceptions Generated with Intel® SSE Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>#I</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD</td>
<td>Add two packed DP FP numbers from XMM2/Mem to XMM1.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ADDSD</td>
<td>Add the lower DP FP number from XMM2/Mem to XMM1.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ANDNPD</td>
<td>Invert the 128 bits in XMM1 and then AND the result with 128 bits from XMM2/Mem.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ANDPD</td>
<td>Logical And of 128 bits from XMM2/Mem to XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPPD</td>
<td>Compare packed DP FP numbers from XMM2/Mem to packed DP FP numbers in XMM1 register using imm8 as predicate.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPSD</td>
<td>Compare lowest DP FP number from XMM2/Mem to lowest DP FP number in XMM1 register using imm8 as predicate.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMISD</td>
<td>Compare lower DP FP number in XMM1 register with lower DP FP number in XMM2/Mem and set the status flags accordingly.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTDQ2PS</td>
<td>Convert four 32-bit signed integers from XMM/Mem to four SP FP.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSP2DQ</td>
<td>Convert four SP FP from XMM/Mem to four 32-bit signed integers in XMM using rounding specified by MXCSR.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTPS2DQ</td>
<td>Convert four SP FP from XMM/Mem to four 32-bit signed integers in XMM using truncate.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTDQ2PD</td>
<td>Convert two 32-bit signed integers in XMM2/Mem to 2 DP FP in xmm1 using rounding specified by MXCSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPD2DQ</td>
<td>Convert two DP FP from XMM2/Mem to two 32-bit signed integers in xmm1 using rounding specified by MXCSR.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPD2PI</td>
<td>Convert lower two DP FP from XMM/Mem to two 32-bit signed integers in MM using rounding specified by MXCSR.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td>Convert two DP FP to two SP FP.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CVTPI2PD</td>
<td>Convert two 32-bit signed integers from MM2/Mem to two DP FP.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSP2PD</td>
<td>Convert two SP FP to two DP FP.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>#I</td>
<td>#D</td>
<td>#Z</td>
<td>#O</td>
<td>#U</td>
<td>#P</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>CVTSD2SI</td>
<td>Convert one DP FP from XMM/Mem to one 32 bit signed integer using rounding mode specified by MXCSR, and move the result to an integer register.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td>Convert scalar DP FP to scalar SP FP.</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>CVTSI2SD</td>
<td>Convert one 32-bit signed integer from Integer Reg/Mem to one DP FP.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td>Convert scalar SP FP to scalar DP FP.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTTPD2DQ</td>
<td>Convert two DP FP from XMM2/Mem to two 32-bit signed integers in XMM1 using truncate.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTTPD2PI</td>
<td>Convert two DP FP from XMM2/Mem to two 32-bit signed integers in MM1 using truncate.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTSD2SI</td>
<td>Convert lowest DP FP from XMM/Mem to one 32-bit signed integer using truncate, and move the result to an integer register.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPD</td>
<td>Divide packed DP FP numbers in XMM1 by XMM2/Mem.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DIVSD</td>
<td>Divide lower DP FP numbers in XMM1 by XMM2/Mem.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>MAXPD</td>
<td>Return the maximum DP FP numbers between XMM2/Mem and XMM1.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXSD</td>
<td>Return the maximum DP FP number between the lower DP FP numbers from XMM2/Mem and XMM1.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINPD</td>
<td>Return the minimum DP numbers between XMM2/Mem and XMM1.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINSD</td>
<td>Return the minimum DP FP number between the lowest DP FP numbers from XMM2/Mem and XMM1.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVAPD</td>
<td>Move 128 bits representing 2 packed DP data from XMM2/Mem to XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or Move 128 bits representing 2 packed DP from XMM1 register to XMM2/Mem.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVHPD</td>
<td>Move 64 bits representing one DP operand from Mem to upper field of XMM register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or move 64 bits representing one DP operand from upper field of XMM register to Mem.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVLPD</td>
<td>Move 64 bits representing one DP operand from Mem to lower field of XMM register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or move 64 bits representing one DP operand from lower field of XMM register to Mem.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVMSKPD</td>
<td>Move the sign mask to r32.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSD</td>
<td>Move 64 bits representing one scalar DP operand from XMM2/Mem to XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or move 64 bits representing one scalar DP operand from XMM1 register to XMM2/Mem.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVUPD</td>
<td>Move 128 bits representing 2 DP data from XMM2/Mem to XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or move 128 bits representing 2 DP data from XMM1 register to XMM2/Mem.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPD</td>
<td>Multiply packed DP FP numbers in XMM2/Mem to XMM1.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
FLOATING-POINT EXCEPTIONS SUMMARY

### C.5 INTEL® SSE3 INSTRUCTIONS

Table C-5 lists the Intel SSE3 instructions that have at least one of the following characteristics:

- Has floating-point operands.
- Generates floating-point results.

For each instruction, the table summarizes the floating-point exceptions that the instruction can generate.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>#I</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULSD</td>
<td>Multiply the lowest DP FP number in XMM2/Mem to XMM1.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ORPD</td>
<td>OR 128 bits from XMM2/Mem to XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHUFPD</td>
<td>Shuffle Double.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPD</td>
<td>Square Root Packed Double Precision</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>SQRTSD</td>
<td>Square Root Scaler Double Precision</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>SUBPD</td>
<td>Subtract Packed Double Precision.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SUBSD</td>
<td>Subtract Scaler Double Precision.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>UCOMISD</td>
<td>Compare lower DP FP number in XMM1 register with lower DP FP number in XMM2/Mem and set the status flags accordingly.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNPCKHPD</td>
<td>Interleaves DP FP numbers from the high halves of XMM1 and XMM2/Mem into XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNPCKLPD</td>
<td>Interleaves DP FP numbers from the low halves of XMM1 and XMM2/Mem into XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XORPD</td>
<td>XOR 128 bits from XMM2/Mem to XMM1 register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C.6 SSSE3 INSTRUCTIONS

SSSE3 instructions operate on integer data elements. They do not generate floating-point exceptions.

### C.7 INTEL® SSE4 INSTRUCTIONS

Table C-6 lists the Intel SSE4.1 instructions that generate floating-point results.

For each instruction, the table summarizes the floating-point exceptions that the instruction can generate.

*Other Intel SSE3 instructions do not generate floating-point exceptions.*
FLOATING-POINT EXCEPTIONS SUMMARY

Table C-6. Exceptions Generated with Intel® SSE4 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>#I</th>
<th>#D</th>
<th>#Z</th>
<th>#O</th>
<th>#U</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPPD</td>
<td>DP FP dot product.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DPPS</td>
<td>SP FP dot product.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ROUNDPD</td>
<td>Round packed DP FP values to integer FP values.</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUNDPS</td>
<td>Round packed SP FP values to integer FP values.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>ROUNDSD</td>
<td>Round scalar DP FP value to integer FP value.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>ROUNDSS</td>
<td>Round scalar SP FP value to integer FP value.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

NOTES:
1. If bit 3 of immediate operand is 0.

Other Intel SSE4.1 and SSE4.2 instructions do not generate floating-point exceptions.
See Section 11.5, “Intel® SSE, SSE2, and SSE3 Exceptions,” for a detailed discussion of SIMD floating-point exceptions.

This appendix considers only Intel SSE, SSE2, and SSE3 instructions that can generate numeric (SIMD floating-point) exceptions, and gives an overview of the necessary support for handling such exceptions. This appendix does not address instructions that do not generate floating-point exceptions (such as RSQRTSS, RSQRTPS, RCPSS, or RCPPS), any x87 instructions, or any unlisted instruction.

For detailed information on which instructions generate numeric exceptions, and a listing of those exceptions, refer to Appendix C, “Floating-Point Exceptions Summary.” Non-numeric exceptions are handled in a way similar to that for the standard IA-32 instructions.

D.1 TWO OPTIONS FOR HANDLING FLOATING-POINT EXCEPTIONS

Just as for x87 FPU floating-point exceptions, the processor takes one of two possible courses of action when an SSE/SSE2/SSE3 instruction raises a floating-point exception:

- If the exception being raised is masked (by setting the corresponding mask bit in the MXCSR to 1), then a default result is produced which is acceptable in most situations. No external indication of the exception is given, but the corresponding exception flags in the MXCSR are set and may be examined later. Note though that for packed operations, an exception flag that is set in the MXCSR will not tell which of the sub-operands caused the event to occur.

- If the exception being raised is not masked (by setting the corresponding mask bit in the MXCSR to 0), a software exception handler previously registered by the user with operating system support will be invoked through the SIMD floating-point exception (#XM, exception 19). This case is discussed below in Section D.2, “Software Exception Handling.”

D.2 SOFTWARE EXCEPTION HANDLING

The #XM handler is usually part of the system software (the operating system kernel). Note that an interrupt descriptor table (IDT) entry must have been previously set up for exception 19 (refer to Chapter 6, “Interrupt and Exception Handling,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). Some compilers use specific run-time libraries to assist in floating-point exception handling. If any x87 FPU floating-point operations are going to be performed that might raise floating-point exceptions, then the exception handling routine must either disable all floating-point exceptions (for example, loading a local control word with FLDCW), or it must be implemented as re-entrant. If this is not the case, the routine has to clear the status flags for x87 FPU exceptions or to mask all x87 FPU floating-point exceptions. For SIMD floating-point exceptions though, the exception flags in MXCSR do not have to be cleared, even if they remain unmasked (but they may still be cleared). Exceptions are in this case precise and occur immediately, and a SIMD floating-point exception status flag that is set when the corresponding exception is unmasked will not generate an exception.

Typical actions performed by this low-level exception handling routine are:

- Incrementing an exception counter for later display or printing.
- Printing or displaying diagnostic information (e.g., the MXCSR and XMM registers).
- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it.
- Storing information about the exception in a data structure that will be passed to a higher level user exception handler.
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

In most cases (and this applies also to the Intel SSE, SSE2, and SSE3 instructions), there will be three main components of a low-level floating-point exception handler: a prologue, a body, and an epilogue.

The prologue performs functions that must be protected from possible interruption by higher-priority sources - typically saving registers and transferring diagnostic information from the processor to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler (assuming that the interrupt handler was called through an interrupt gate, meaning that the processor cleared the interrupt enable (IF) flag in the EFLAGS register - refer to Section 6.5.1, "Call and Return Operation for Interrupt or Exception Handling Procedures").

The body of the exception handler examines the diagnostic information and makes a response that is application-dependent. It may range from halting execution, to displaying a message, to attempting to fix the problem and then proceeding with normal execution, to setting up a data structure, calling a higher-level user exception handler and continuing execution upon return from it. This latter case will be assumed in Section D.4, "SIMD Floating-Point Exceptions and the IEEE Standard 754," below.

Finally, the epilogue essentially reverses the actions of the prologue, restoring the processor state so that normal execution can be resumed.

The following example represents a typical exception handler. To link it with Example D-2 that will follow in Section D.4.3, "Example SIMD Floating-Point Emulation Implementation," assume that the body of the handler (not shown here in detail) passes the saved state to a routine that will examine in turn all the sub-operands of the excepting instruction, invoking a user floating-point exception handler if a particular set of sub-operands raises an unmasked (enabled) exception, or emulating the instruction otherwise.

Example D-1.  SIMD Floating-Point Exception Handler

SIMD_FP_EXC_HANDLER PROC

;PROLOGUE
;SAVE REGISTERS THAT MIGHT BE USED BY THE EXCEPTION HANDLER
PUSH EBP ;SAVE EBP
PUSH EAX ;SAVE EAX
...
MOV EBP, ESP ;SAVE ESP in EBP
SUB ESP, 512 ;ALLOCATE 512 BYTES
AND ESP, 0fffffff0h ;MAKE THE ADDRESS 16-BYTE ALIGNED
FXSAVE [ESP] ;SAVE FP, MMX, AND SIMD FP STATE
PUSH [EBP+EFLAGS_OFFSET] ;COPY OLD EFLAGS TO STACK TOP
POPF ;RESTORE THE INTERRUPT ENABLE FLAG IF
;TO VALUE BEFORE SIMD FP EXCEPTION

;BODY
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
LDMXCSR LOCAL_MXCSR ;LOAD LOCAL MXCSR VALUE IF NEEDED
...
...

;EPilogue
FXRSTOR [ESP] ;RESTORE MODIFIED STATE IMAGE
MOV ESP, EBP ;DE-ALLOCATE STACK SPACE
...
POP EAX ;RESTORE EAX
POP EBP ;RESTORE EBP
IRET ;RETURN TO INTERRUPTED CALCULATION
SIMD_FP_EXC_HANDLER ENDP
D.3 EXCEPTION SYNCHRONIZATION

An SSE/SSE2/SSE3 instruction can execute in parallel with other similar instructions, with integer instructions, and with floating-point or MMX instructions. Unlike for x87 instructions, special precaution for exception synchronization is not necessary in this case. This is because floating-point exceptions for SSE/SSE2/SSE3 instructions occur immediately and are not delayed until a subsequent floating-point instruction is executed. However, floating-point emulation may be necessary when unmasked floating-point exceptions are generated.

D.4 SIMD FLOATING-POINT EXCEPTIONS AND THE IEEE STANDARD 754

SSE/SSE2/SSE3 extensions are 100% compatible with the IEEE Standard 754 for Floating-Point Arithmetic, satisfying all of its mandatory requirements (when the flush-to-zero or denormals-are-zeros modes are not enabled). But a programming environment that includes SSE/SSE2/SSE3 instructions will comply with both the obligatory and the strongly recommended requirements of the IEEE Standard 754 regarding floating-point exception handling, only as a combination of hardware and software (which is acceptable). The standard states that a user should be able to request a trap on any of the five floating-point exceptions (note that the denormal exception is an IA-32 addition), and it also specifies the values (operands or result) to be delivered to the exception handler.

The main issue is that for SSE/SSE2/SSE3 instructions that raise post-computation exceptions (traps: overflow, underflow, or inexact), unlike for x87 FPU instructions, the processor does not provide the result recommended by IEEE Standard 754 to the user handler. If a user program needs the result of an instruction that generated a post-computation exception, it is the responsibility of the software to produce this result by emulating the faulting SSE/SSE2/SSE3 instruction. Another issue is that the standard does not specify explicitly how to handle multiple floating-point exceptions that occur simultaneously. For packed operations, a logical OR of the flags that would be set by each sub-operation is used to set the exception flags in the MXCSR. The following subsections present one possible way to solve these problems.

D.4.1 Floating-Point Emulation

Every operating system must provide a kernel level floating-point exception handler (a template was presented in Section D.2, "Software Exception Handling," above). In the following discussion, assume that a user mode floating-point exception filter is supplied for SIMD floating-point exceptions (for example as part of a library of C functions), that a user program can invoke in order to handle unmasked exceptions. The user mode floating-point exception filter (not shown here) has to be able to emulate the subset of Intel SSE, SSE2, and SSE3 instructions that can generate numeric exceptions, and has to be able to invoke a user provided floating-point exception handler for floating-point exceptions. When a floating-point exception that is not masked is raised by an Intel SSE, SSE2, and SSE3 instruction, the low-level floating-point exception handler will be called. This low-level handler may in turn call the user mode floating-point exception filter. The filter function receives the original operands of the excepting instruction as no results are provided by the hardware, whether a pre-computation or a post-computation exception has occurred. The filter will unpack the operands into up to four sets of sub-operands, and will submit them one set at a time to an emulation function (See Example D-2 in Section D.4.3, "Example SIMD Floating-Point Emulation Implementation."). The emulation function will examine the sub-operands, and will possibly redo the necessary calculation.

Two cases are possible:

- If an unmasked (enabled) exception would occur in this process, the emulation function will return to its caller (the filter function) with the appropriate information. The filter will invoke a (previously registered) user floating-point exception handler for this set of sub-operands, and will record the result upon return from the user handler (provided the user handler allows continuation of the execution).
- If no unmasked (enabled) exception would occur, the emulation function will determine and will return to its caller the result of the operation for the current set of sub-operands (it has to be IEEE Standard 754 compliant). The filter function will record the result (plus any new flag settings).

The user level filter function will then call the emulation function for the next set of sub-operands (if any). When done with all the operand sets, the partial results will be packed (if the excepting instruction has a packed floating-point result, which is true for most SSE/SSE2/SSE3 numeric instructions) and the filter will return to the low-level exception handler, which in turn will return from the interruption, allowing execution to continue. Note that the
instruction pointer (EIP) has to be altered to point to the instruction following the excepting instruction, in order to continue execution correctly.

If a user mode floating-point exception filter is not provided, then all the work for decoding the excepting instruction, reading its operands, emulating the instruction for the components of the result that do not correspond to unmasked floating-point exceptions, and providing the compounded result will have to be performed by the user-provided floating-point exception handler.

Actual emulation might have to take place for one operand or pair of operands for scalar operations, and for all sub-operands or pairs of sub-operands for packed operations. The steps to perform are the following:

- The excepting instruction has to be decoded and the operands have to be read from the saved context.
- The instruction has to be emulated for each (pair of) sub-operand(s); if no floating-point exception occurs, the partial result has to be saved; if a masked floating-point exception occurs, the masked result has to be produced through emulation and saved, and the appropriate status flags have to be set; if an unmasked floating-point exception occurs, the result has to be generated by the user provided floating-point exception handler, and the appropriate status flags have to be set.
- The partial results have to be combined and written to the context that will be restored upon application program resumption.

A diagram of the control flow in handling an unmasked floating-point exception is presented below.

![Figure D-1. Control Flow for Handling Unmasked Floating-Point Exceptions](image)

From the user-level floating-point filter, Example D-2 in Section D.4.3, “Example SIMD Floating-Point Emulation Implementation,” presents only the floating-point emulation part. In order to understand the actions involved, the expected response to exceptions has to be known for all Intel SSE, SSE2, and SSE3 numeric instructions in two situations: with exceptions enabled (unmasked result), and with exceptions disabled (masked result). The latter can be found in Section 6.5, “Interrupts and Exceptions.” The response to NaN operands that do not raise an exception is specified in Section 4.8.3.4, “NaNs.” Operations on NaNs are explained in the same source. This response is also discussed in more detail in the next subsection, along with the unmasked and masked responses to floating-point exceptions.

### D.4.2 Intel® SSE, SSE2, and SSE3 Response To Floating-Point Exceptions

This subsection specifies the unmasked response expected from the Intel SSE, SSE2, and SSE3 instructions that raise floating-point exceptions. The masked response is given in parallel, as it is necessary in the emulation process.
of the instructions that raise unmasked floating-point exceptions. The response to NaN operands is also included in more detail than in Section 4.8.3.4, "NaNs." For floating-point exception priority, refer to "Priority Among Simultaneous Exceptions and Interrupts" in Chapter 6, “Interrupt and Exception Handling,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

D.4.2.1  Numeric Exceptions

There are six classes of numeric (floating-point) exception conditions that can occur: Invalid operation (#I), Divide-by-Zero (#Z), Denormal Operand (#D), Numeric Overflow (#O), Numeric Underflow (#U), and Inexact Result (precision) (#P). #I, #Z, #D are pre-computation exceptions (floating-point faults), detected before the arithmetic operation. #O, #U, #P are post-computation exceptions (floating-point traps).

Users can control how the Intel SSE, SSE2, and SSE3 floating-point exceptions are handled by setting the mask/unmask bits in MXCSR. Masked exceptions are handled by the processor, or by software if they are combined with unmasked exceptions occurring in the same instruction. Unmasked exceptions are usually handled by the low-level exception handler, in conjunction with user-level software.

D.4.2.2  Results of Operations with NaN Operands or a NaN Result for Intel® SSE, SSE2, and SSE3 Numeric Instructions

The tables below (E-1 through E-10) specify the response of Intel SSE, SSE2, and SSE3 instructions to NaN inputs, or to other inputs that lead to NaN results.

These results will be referenced by subsequent tables (e.g., E-10). Most operations do not raise an invalid exception for quiet NaN operands, but even so, they will have higher precedence over raising floating-point exceptions other than invalid operation.

Note that the single precision QNaN Indefinite value is FFC00000H, the double precision QNaN Indefinite value is FFF8000000000000H, and the Integer Indefinite value is 80000000H (not a floating-point number, but it can be the result of a conversion instruction from floating-point to integer).

For an unmasked exception, no result will be provided by the hardware to the user handler. If a user registered floating-point exception handler is invoked, it may provide a result for the excepting instruction, that will be used if execution of the application code is continued after returning from the interruption.

In Tables D-1 through Table D-12, the specified operands cause an invalid exception, unless the unmasked result is marked with "not an exception". In this latter case, the unmasked and masked results are the same.

Table D-1.  ADDPS, ADDSS, SUBPS, SUBSS, MULPS, MULSS, DIVPS, DIVSS, ADDPD, ADDSD, SUBPD, SUBSD, MULPD, MULSD, DIVPD, DIVSD, ADDSUBPS, ADDSUBPD, HADDPS, HADDPD, HSUBPS, and HSUBPD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNaN1 op SNaN2</td>
<td>SNaN1</td>
<td>00400000H or SNaN1</td>
</tr>
<tr>
<td>SNaN1 op QNaN2</td>
<td>SNaN1</td>
<td>00400000H or SNaN1</td>
</tr>
<tr>
<td>QNaN1 op SNaN2</td>
<td>QNaN1</td>
<td>None</td>
</tr>
<tr>
<td>QNaN1 op QNaN2</td>
<td>QNaN1</td>
<td>None</td>
</tr>
<tr>
<td>SNaN op real value</td>
<td>SNaN</td>
<td>00400000H or SNaN</td>
</tr>
<tr>
<td>Real value op SNaN</td>
<td>SNaN</td>
<td>00400000H or SNaN</td>
</tr>
<tr>
<td>QNaN op real value</td>
<td>QNaN</td>
<td>QNaN (not an exception)</td>
</tr>
<tr>
<td>Real value op QNaN</td>
<td>QNaN</td>
<td>QNaN (not an exception)</td>
</tr>
</tbody>
</table>
Neither source operand is SNaN, but #I is signaled (e.g., for Inf - Inf, Inf \ast \infty, Inf / Inf, 0/0) 

NOTES:
1. For Tables E-1 to E-12: op denotes the operation to be performed.
2. SNaN | 00400000H is a quiet NaN in single precision format (if SNaN is in single precision) and SNaN | 0008000000000000H is a quiet NaN in double precision format (if SNaN is in double precision), obtained from the signaling NaN given as input.
3. Operations involving only quiet NaNs do not raise floating-point exceptions.

Table D-2. CMPPS.EQ, CMPSS.EQ, CMPPS.ORD, CMPSS.ORD, CMPPD.EQ, CMPSD.EQ, CMPPD.ORD, and CMPSD.ORD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN op Opd2 (any Opd2)</td>
<td>000000000H or 0000000000000000H¹</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFFH¹ (not an exception)</td>
</tr>
<tr>
<td>Opd1 op NaN (any Opd1)</td>
<td>000000000H or 0000000000000000H¹</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFFH¹ (not an exception)</td>
</tr>
</tbody>
</table>

NOTE:
1. 32-bit results are for single, and 64-bit results for double precision operations.

Table D-3. CMPPS.NEQ, CMPSS.NEQ, CMPPS.UNORD, CMPSS.UNORD, CMPPD.NEQ, CMPSD.NEQ, CMPPD.UNORD, and CMPSD.UNORD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN op Opd2 (any Opd2)</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFFH¹</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFFH¹ (not an exception)</td>
</tr>
<tr>
<td>Opd1 op NaN (any Opd1)</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFFH¹</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFFH¹ (not an exception)</td>
</tr>
</tbody>
</table>

NOTE:
1. 32-bit results are for single, and 64-bit results for double precision operations.

Table D-4. CMPPS.LT, CMPSS.LT, CMPPS.LE, CMPSS.LE, CMPPD.LT, CMPSD.LT, CMPPD.LE, and CMPSD.LE

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN op Opd2 (any Opd2)</td>
<td>000000000H or 0000000000000000H¹</td>
<td>None</td>
</tr>
<tr>
<td>Opd1 op NaN (any Opd1)</td>
<td>000000000H or 0000000000000000H¹</td>
<td>None</td>
</tr>
</tbody>
</table>

NOTE:
1. 32-bit results are for single, and 64-bit results for double precision operations.
### Table D-5. CMPPS.NLT, CMPSS.NLT, CMPPS.NLE, CMPSS.NLE, CMPPD.NLT, CMPSD.NLT, CMPPD.NLE, and CMPSD.NLE

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN op Opd2 (any Opd2)</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFH&lt;sup&gt;1&lt;/sup&gt;</td>
<td>None</td>
</tr>
<tr>
<td>Opd1 op NaN (any Opd1)</td>
<td>FFFFFFFFH or FFFFFFFFFFFFFFFFFH&lt;sup&gt;1&lt;/sup&gt;</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTE:**
1. 32-bit results are for single, and 64-bit results for double precision operations.

### Table D-6. COMISS and COMISD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNaN op Opd2 (any Opd2)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>None</td>
</tr>
<tr>
<td>Opd1 op SNaN (any Opd1)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>None</td>
</tr>
<tr>
<td>QNaN op Opd2 (any Opd2)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>None</td>
</tr>
<tr>
<td>Opd1 op QNaN (any Opd1)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>None</td>
</tr>
</tbody>
</table>

### Table D-7. UCOMISS and UCOMISD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNaN op Opd2 (any Opd2)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>None</td>
</tr>
<tr>
<td>Opd1 op SNaN (any Opd1)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>None</td>
</tr>
<tr>
<td>QNaN op Opd2 (any Opd2 ≠ SNaN)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111 (not an exception)</td>
</tr>
<tr>
<td>Opd1 op QNaN (any Opd1 ≠ SNaN)</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111</td>
<td>OF, SF, AF = 000 ZF, PF, CF = 111 (not an exception)</td>
</tr>
</tbody>
</table>

### Table D-8. CVTPS2PI, CVTSS2SI, CVTTPS2PI, CVTTSS2SI, CVTPD2PI, CVTSD2SI, CVTTPD2PI, CVTTSD2SI, CVTPS2DQ, CVTTPS2DQ, CVTPD2DQ, and CVTTPD2DQ

<table>
<thead>
<tr>
<th>Source Operand</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNaN</td>
<td>80000000H or 8000000000000000&lt;sup&gt;1&lt;/sup&gt; (Integer Indefinite)</td>
<td>None</td>
</tr>
<tr>
<td>QNaN</td>
<td>80000000H or 8000000000000000&lt;sup&gt;1&lt;/sup&gt; (Integer Indefinite)</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTE:**
1. 32-bit results are for single, and 64-bit results for double precision operations.
Table D-9. MAXPS, MAXSS, MINPS, MINSS, MAXPD, MAXSD, MINPD, and MINSD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opd1 op NaN2 (any Opd1)</td>
<td>NaN2</td>
<td>None</td>
</tr>
<tr>
<td>NaN1 op Opd2 (any Opd2)</td>
<td>Opd2</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTE:**
1. SNaN and QNaN operands raise an Invalid Operation fault.

Table D-10. SQRTPS, SQRTSS, SQRTPD, and SQRTSD

<table>
<thead>
<tr>
<th>Source Operand</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNaN</td>
<td>QNaN</td>
<td>QNaN (not an exception)</td>
</tr>
<tr>
<td>SNaN</td>
<td>SNaN</td>
<td>SNaN</td>
</tr>
<tr>
<td>Source operand is not SNaN; but #I is signaled (e.g., for sqrt (-1.0))</td>
<td>Single precision or double precision QNaN Indefinite</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTE:**
1. SNaN | 00400000H is a quiet NaN in single precision format (if SNaN is in single precision) and SNaN | 0008000000000000H is a quiet NaN in double precision format (if SNaN is in double precision), obtained from the signaling NaN given as input.

Table D-11. CVTPS2PD and CVTSS2SD

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNaN</td>
<td>QNaN1</td>
<td>QNaN1 (not an exception)</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN1</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The double precision output QNaN1 is created from the single precision input QNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 11-bit exponent 7FFH, and the 24-bit significand is extended to a 53-bit significand by appending 29 bits equal to 0.
2. The double precision output QNaN1 is created from the single precision input SNaN as follows: the sign bit is preserved, the 8-bit exponent FFH is replaced by the 11-bit exponent 7FFH, and the 24-bit significand is extended to a 53-bit significand by appending 29 bits equal to 0. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.

Table D-12. CVTPD2PS and CVTSD2SS

<table>
<thead>
<tr>
<th>Source Operands</th>
<th>Masked Result</th>
<th>Unmasked Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNaN</td>
<td>QNaN1</td>
<td>QNaN1 (not an exception)</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN1</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The single precision output QNaN1 is created from the double precision input QNaN as follows: the sign bit is preserved, the 11-bit exponent 7FFH is replaced by the 8-bit exponent FFH, and the 53-bit significand is truncated to a 24-bit significand by removing its 29 least significant bits.
2. The single precision output QNaN1 is created from the double precision input SNaN as follows: the sign bit is preserved, the 11-bit exponent 7FFH is replaced by the 8-bit exponent FFH, and the 53-bit significand is truncated to a 24-bit significand by removing its 29 least significant bits. The second most significant bit of the significand is changed from 0 to 1 to convert the signaling NaN into a quiet NaN.
D.4.2.3  Condition Codes, Exception Flags, and Response for Masked and Unmasked Numeric Exceptions

In the following, the masked response is what the processor provides when a masked exception is raised by an Intel SSE, SSE2, or SSE3 numeric instruction. The same response is provided by the floating-point emulator for Intel SSE, SSE2, and SSE3 numeric instructions, when certain components of the quadruple input operands generate exceptions that are masked (the emulator also generates the correct answer, as specified by IEEE Standard 754 wherever applicable, in the case when no floating-point exception occurs). The unmasked response is what the emulator provides to the user handler for those components of the packed operands of Intel SSE, SSE2, and SSE3 instructions that raise unmasked exceptions. Note that for pre-computation exceptions (floating-point faults), no result is provided to the user handler. For post-computation exceptions (floating-point traps), a result is provided to the user handler, as specified below.

In the following tables, the result is denoted by ‘res’, with the understanding that for the actual instruction, the destination coincides with the first source operand (except for COMISS, UCOMISS, COMISD, and UCOMISD, whose destination is the EFLAGS register).

Table D-13. #I - Invalid Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS</td>
<td>src1 or src2(^1) = SNaN</td>
<td>Refer to Table D-1 for NaN operands, #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>ADDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HADDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSUBPS (the addition component)</td>
<td>src1 = +Inf, src2 = -Inf or src1 = -Inf, src2 = +Inf</td>
<td>res(^1) = QNaN Indefinite, #IA = 1</td>
<td></td>
</tr>
<tr>
<td>ADDSUBPD (the addition component)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPS</td>
<td>src1 or src2 = SNaN</td>
<td>Refer to Table D-1 for NaN operands, #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>SUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSUBPS (the subtraction component)</td>
<td>src1 = +Inf, src2 = +Inf or src1 = -Inf, src2 = -Inf</td>
<td>res = QNaN Indefinite, #IA = 1</td>
<td></td>
</tr>
<tr>
<td>ADDSUBPD (the subtraction component)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPS</td>
<td>src1 or src2 = SNaN</td>
<td>Refer to Table D-1 for NaN operands, #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>MULPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSS</td>
<td>src1 = ±Inf, src2 = ±0 or src1 = ±0, src2 = ±Inf</td>
<td>res = QNaN Indefinite, #IA = 1</td>
<td></td>
</tr>
<tr>
<td>MULSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS</td>
<td>src1 or src2 = SNaN</td>
<td>Refer to Table D-1 for NaN operands, #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>DIVPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSS</td>
<td>src1 = ±Inf, src2 = ±Inf or src1 = ±0, src2 = ±0</td>
<td>res = QNaN Indefinite, #IA = 1</td>
<td></td>
</tr>
<tr>
<td>DIVSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPS</td>
<td>src = SNaN</td>
<td>Refer to Table D-10 for NaN operands, #IA = 1</td>
<td>src unchanged, #IA = 1</td>
</tr>
<tr>
<td>SQRTPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSD</td>
<td>src &lt; 0 (note that -0 &lt; 0 is false)</td>
<td>res = QNaN Indefinite, #IA = 1</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) In the case of ADDPS, ADDPD, ADDSS, ADDSD, HADDPD, ADDSUBPS, ADDSUBPD, SUBPS, SUBPD, SUBSS, SUBSD, HSUBPS, HSUBPD, SQRTPS, SQRTPD, SQRTSS, SQRTSD, the destination coincides with the first source operand.
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPS, MAXSS, MAXPD, MAXSD</td>
<td>src1 = NaN or src2 = NaN</td>
<td>res = src2, #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>MINPS, MINSS, MINPD, MINSD</td>
<td>src1 = NaN or src2 = NaN</td>
<td>res = src2, #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>CMPPS.LT, CMPPS.LE, CMPPS.NLT, CMPPS.NLE, CMPSS.LT, CMPSS.LE, CMPSS.NLT, CMPSS.NLE, CMPPD.LT, CMPPD.LE, CMPPD.NLT, CMPPD.NLE, CMPPSD.LT, CMPSDL.E, CMPSDL.NLT, CMPSDL.NLE</td>
<td>src1 = NaN or src2 = NaN</td>
<td>Refer to Table D-4 and Table D-5 for NaN operands; #IA = 1</td>
<td>src1, src2 unchanged; #IA = 1</td>
</tr>
<tr>
<td>COMISS, COMISD</td>
<td>src1 = NaN or src2 = NaN</td>
<td>Refer to Table D-6 for NaN operands</td>
<td>src1, src2, EFLAGS unchanged; #IA = 1</td>
</tr>
<tr>
<td>UCOMISS, UCOMISD</td>
<td>src1 = NaN or src2 = NaN</td>
<td>Refer to Table D-7 for NaN operands</td>
<td>src1, src2, EFLAGS unchanged; #IA = 1</td>
</tr>
<tr>
<td>CVTTPS2PI, CVTSS2SI, CVTPD2PI, CVTSD2SI, CVTPS2DQ, CVTPD2DQ</td>
<td>src = NaN, ±Inf, or</td>
<td>res = Integer Indefinite, #IA = 1</td>
<td>src unchanged, #IA = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Note 2 for information on rnd.</td>
<td></td>
</tr>
<tr>
<td>CVTTPS2PI, CVTTSS2SI, CVTTPD2PI, CVTTSD2SI, CVTTPS2DQ, CVTTPD2DQ</td>
<td>src = NaN, ±Inf, or</td>
<td>res = Integer Indefinite, #IA = 1</td>
<td>src unchanged, #IA = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Note 2 for information on rz.</td>
<td></td>
</tr>
</tbody>
</table>
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

Table D-11. CVTPS2PD, CVTSS2SD

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTPS2PD</td>
<td>src = SNAN</td>
<td>Refer to Table D-11 for NaN operands</td>
<td>src unchanged, #IA = 1</td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. For Tables E-13 to E-18:
   - src denotes the single source operand of a unary operation.
   - src1, src2 denote the first and second source operand of a binary operation.
   - res denotes the numerical result of an operation.
2. rnd signifies the user rounding mode from MXCSR, and rz signifies the rounding mode toward zero (truncate), when rounding a floating-point value to an integer. For more information, refer to Table 4-8.
3. For NAN encodings, see Table 4-3.

Table D-12. CVTDP2PS, CVTSD2SS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTDP2PS</td>
<td>src = SNAN</td>
<td>Refer to Table D-12 for NaN operands</td>
<td>src unchanged, #IA = 1</td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table D-13. #I - Invalid Operations (Contd.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVPS</td>
<td>src1 = finite non-zero (normal, or denormal) src2 = ±0</td>
<td>res = ±Inf, #ZE = 1</td>
<td>src1, src2 unchanged; #ZE = 1</td>
</tr>
<tr>
<td>DIVSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table D-14. #Z - Divide-by-Zero

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVPS</td>
<td>src1 = finite non-zero (normal, or denormal) src2 = ±0</td>
<td>res = ±Inf, #ZE = 1</td>
<td>src1, src2 unchanged; #ZE = 1</td>
</tr>
</tbody>
</table>
### Table D-15. #D - Denormal Operand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS</td>
<td>src1 = denormal(^1) or src2 = denormal (and the DAZ bit in MXCSR is 0)</td>
<td>res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked post-computation exception occurs; #DE = 1.</td>
<td>src1, src2 unchanged; #DE = 1</td>
</tr>
<tr>
<td>ADDPD</td>
<td></td>
<td></td>
<td>Note that SQRT, CVTPS2PD, CVTSS2SD, CVTPD2PS, CVTSD2SS have only 1 src.</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HADDPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HADDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINDSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPS2PD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPPS</td>
<td>src1 = denormal(^1) or src2 = denormal (and the DAZ bit in MXCSR is 0)</td>
<td>Comparison result, stored in the destination register; #DE = 1</td>
<td>src1, src2 unchanged; #DE = 1</td>
</tr>
<tr>
<td>CMPPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMISS</td>
<td>src1 = denormal(^1) or src2 = denormal (and the DAZ bit in MXCSR is 0)</td>
<td>Comparison result, stored in the EFLAGS register; #DE = 1</td>
<td>src1, src2 unchanged; #DE = 1</td>
</tr>
<tr>
<td>COMISD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UCOMISS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UCOMISD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**

1. For denormal encodings, see Section 4.8.3.2, “Normalized and Denormalized Finite Numbers.”
### Table D-16. #O - Numeric Overflow

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rounding</td>
<td>Sign</td>
<td>Result &amp; Status Flags</td>
</tr>
<tr>
<td>ADDPS</td>
<td>To nearest</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td></td>
<td></td>
<td>res = +∞</td>
</tr>
<tr>
<td>HSUBPS</td>
<td></td>
<td></td>
<td>res = -∞</td>
</tr>
<tr>
<td>MULPS</td>
<td>Toward --∞</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>DIVPS</td>
<td></td>
<td></td>
<td>res = 1.11...1 * 2(^{127})</td>
</tr>
<tr>
<td>ADDSS</td>
<td>Toward +∞</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>SUBSS</td>
<td></td>
<td></td>
<td>res = 1.11...1 * 2(^{127})</td>
</tr>
<tr>
<td>MULSS</td>
<td>Toward 0</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>DIVSS</td>
<td></td>
<td></td>
<td>res = 1.11...1 * 2(^{127})</td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td></td>
<td></td>
<td>res = (result calculated with unbounded exponent and rounded to the destination precision) / 2(^{152})</td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td></td>
<td></td>
<td>#OE = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#PE = 1 if the result is inexact</td>
</tr>
<tr>
<td>ADDPD</td>
<td>To nearest</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td></td>
<td></td>
<td>res = +∞</td>
</tr>
<tr>
<td>SUBPD</td>
<td>Toward --∞</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>MULPD</td>
<td></td>
<td></td>
<td>res = 1.11...1 * 2(^{1023})</td>
</tr>
<tr>
<td>DIVPD</td>
<td>Toward +∞</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>ADDSD</td>
<td></td>
<td></td>
<td>res = 1.11...1 * 2(^{1023})</td>
</tr>
<tr>
<td>SUBSD</td>
<td>Toward 0</td>
<td>+</td>
<td>#OE = 1, #PE = 1</td>
</tr>
<tr>
<td>MULSD</td>
<td></td>
<td></td>
<td>res = 1.11...1 * 2(^{1023})</td>
</tr>
<tr>
<td>DIVSD</td>
<td></td>
<td></td>
<td>res = (result calculated with unbounded exponent and rounded to the destination precision) / 2(^{156})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#OE = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#PE = 1 if the result is inexact</td>
</tr>
</tbody>
</table>
### Table D-17. #U - Numeric Underflow

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS</td>
<td>Result calculated with unbounded exponent and rounded to the destination precision &lt; smallest single precision finite normal value.</td>
<td>res = ±0, denormal, or normal #UE = 1 and #PE = 1, but only if the result is inexact</td>
<td>res = (result calculated with unbounded exponent and rounded to the destination precision) * $2^{192}$</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td></td>
<td></td>
<td><strong>#UE = 1</strong></td>
</tr>
<tr>
<td>HADDPSP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDPD</td>
<td>Result calculated with unbounded exponent and rounded to the destination precision &lt; smallest double precision finite normal value.</td>
<td>res = ±0, denormal or normal #UE = 1 and #PE = 1, but only if the result is inexact</td>
<td>res = (result calculated with unbounded exponent and rounded to the destination precision) * $2^{1536}$</td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td></td>
<td></td>
<td><strong>#UE = 1</strong></td>
</tr>
<tr>
<td>HADDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table D-18. #P - Inexact Result (Precision)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Masked Response</th>
<th>Unmasked Response and Exception Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS</td>
<td>The result is not exactly representable in the destination format.</td>
<td>res = Result rounded to the destination precision and using the bounded exponent, but only if no unmasked underflow or overflow conditions occur (this exception can occur in the presence of a masked underflow or overflow); #PE = 1.</td>
<td>Only if no underflow/overflow condition occurred, or if the corresponding exceptions are masked:  • Set #OE if masked overflow and set result as described above for masked overflow.  • Set #UE if masked underflow and set result as described above for masked underflow. If neither underflow nor overflow, res equals the result rounded to the destination precision and using the bounded exponent set #PE = 1.</td>
</tr>
<tr>
<td>ADDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HADDPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HADDPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSUBPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTPD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTQ2PS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTQ2PD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTP2PI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTP2PD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT2PS2Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT2PD2Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT2PS2PI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT2PD2PI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT2PS2PD2Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2DD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2DI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2DS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2DD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D.4.3 Example SIMD Floating-Point Emulation Implementation

The sample code listed below may be considered as being part of a user-level floating-point exception filter for the Intel SSE, SSE2, and SSE3 numeric instructions. It is assumed that the filter function is invoked by a low-level exception handler (invoked for exception 19 when an unmasked floating-point exception occurs), and that it operates as explained in Section D.4.1, “Floating-Point Emulation.” The sample code does the emulation only for the SSE instructions for addition, subtraction, multiplication, and division. For this, it uses C code and x87 FPU operations. Operations corresponding to other Intel SSE, SSE2, and SSE3 numeric instructions can be emulated similarly. The example assumes that the emulation function receives a pointer to a data structure specifying a number of input parameters: the operation that caused the exception, a set of sub-operands (unpacked, of type float), the
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

rounding mode (the precision is always single), exception masks (having the same relative bit positions as in the MXCSR but starting from bit 0 in an unsigned integer), and flush-to-zero and denormals-are-zeros indicators.

The output parameters are a floating-point result (of type float), the cause of the exception (identified by constants not explicitly defined below), and the exception status flags. The corresponding C definition is:

typedef struct {
    unsigned int operation; //SSE or SSE2 operation: ADDPS, ADDSS, ...
    unsigned int operand1_uint32; //first operand value
    unsigned int operand2_uint32; //second operand value (if any)
    float result_fval; // result value (if any)
    unsigned int rounding_mode; //rounding mode
    unsigned int exc_masks; //exception masks, in the order P,U,O,Z,D,I
    unsigned int exception_cause; //exception cause
    unsigned int status_flag_inexact; //inexact status flag
    unsigned int status_flag_underflow; //underflow status flag
    unsigned int status_flag_overflow; //overflow status flag
    unsigned int status_flag_divide_by_zero; //divide by zero status flag
    unsigned int status_flag_denormal_operand; //denormal operand status flag
    unsigned int status_flag_invalid_operation; //invalid operation status flag
    unsigned int ftz; // flush-to-zero flag
    unsigned int daz; // denormals-are-zeros flag
} EXC_ENV;

The arithmetic operations exemplified are emulated as follows:

1. If the denormals-are-zeros mode is enabled (the DAZ bit in MXCSR is set to 1), replace all the denormal inputs with zeroes of the same sign (the denormal flag is not affected by this change).

2. Perform the operation using x87 FPU instructions, with exceptions disabled, the original user rounding mode, and single precision. This reveals invalid, denormal, or divide-by-zero exceptions (if there are any) and stores the result in memory as a double precision value (whose exponent range is large enough to look like “unbounded” to the result of the single precision computation).

3. If no unmasked exceptions were detected, determine if the magnitude of the result is less than the smallest normal number that can be represented in single precision format, or greater than the largest normal number that can be represented in single precision format (huge). If an unmasked overflow or underflow occurs, calculate the scaled result that will be handed to the user exception handler, as specified by IEEE Standard 754.

4. If no exception was raised, calculate the result with a “bounded” exponent. If the result is tiny, it requires denormalization (shifting the significand right while incrementing the exponent to bring it into the admissible range of [-126,+127] for single precision floating-point numbers).

The result obtained in step 2 cannot be used because it might incur a double rounding error (it was rounded to 24 bits in step 2, and might have to be rounded again in the denormalization process). To overcome this is, calculate the result as a double precision value, and store it to memory in single precision format.

Rounding first to 53 bits in the significand, and then to 24 never causes a double rounding error (exact properties exist that state when double-rounding error occurs, but for the elementary arithmetic operations, the rule of thumb is that if an infinitely precise result is rounded to 2p+1 bits and then again to p bits, the result is the same as when rounding directly to p bits, which means that no double-rounding error occurs).

5. If the result is inexact and the inexact exceptions are unmasked, the calculated result will be delivered to the user floating-point exception handler.

6. The flush-to-zero case is dealt with if the result is tiny.
7. The emulation function returns RAISE_EXCEPTION to the filter function if an exception has to be raised (the exception_cause field indicates the cause). Otherwise, the emulation function returns DO_NOT_RAISE_EXCEPTION. In the first case, the result is provided by the user exception handler called by the filter function. In the second case, it is provided by the emulation function. The filter function has to collect all the partial results, and to assemble the scalar or packed result that is used if execution is to continue.

Example D-2. SIMD Floating-Point Emulation

// masks for individual status word bits
#define PRECISION_MASK 20H
#define UNDERFLOW_MASK 10H
#define OVERFLOW_MASK 08H
#define ZERODIVIDE_MASK 04H
#define DENORMAL_MASK 02H
#define INVALID_MASK 01H

// 32-bit constants
static unsigned ZEROF_ARRAY[] = {00000000H};
#define ZEROF *(float *) ZEROF_ARRAY
    // +0.0
static unsigned NZEROF_ARRAY[] = {80000000H};
#define NZEROF *(float *) NZEROF_ARRAY
    // -0.0
static unsigned POSINFF_ARRAY[] = {7f800000H};
#define POSINFF *(float *)POSINFF_ARRAY
    // +Inf
static unsigned NEGINFF_ARRAY[] = {ff800000H};
#define NEGINFF *(float *)NEGINFF_ARRAY
    // -Inf

// 64-bit constants
static unsigned MIN_SINGLE_NORMAL_ARRAY[] = {00000000H, 38100000H};
#define MIN_SINGLE_NORMAL *(double *)MIN_SINGLE_NORMAL_ARRAY
    // +1.0 * 2^-126
static unsigned MAX_SINGLE_NORMAL_ARRAY[] = {70000000H, 47efffffH};
#define MAX_SINGLE_NORMAL *(double *)MAX_SINGLE_NORMAL_ARRAY
    // +1.1...1*2^127
static unsigned TWO_TO_192_ARRAY[] = {00000000H, 4bf00000H};
#define TWO_TO_192 *(double *)TWO_TO_192_ARRAY
    // +1.0 * 2^192
static unsigned TWO_TO_M192_ARRAY[] = {00000000H, 33f00000H};
#define TWO_TO_M192 *(double *)TWO_TO_M192_ARRAY
    // +1.0 * 2^-192

// auxiliary functions
static int isnanf (unsigned int ); // returns 1 if f is a NaN, and 0 otherwise
static float quietf (unsigned int ); // converts a signaling NaN to a quiet
    // NaN, and leaves a quiet NaN unchanged
static unsigned int check_for_daz (unsigned int ); // converts denormals
    // to zeros of the same sign;
    // does not affect any status flags

// emulation of SSE and SSE2 instructions using
// C code and x87 FPU instructions

unsigned int
simd_fp_emulate (EXC_ENV *exc_env)
{
    int uiopd1; // first operand of the add, subtract, multiply, or divide
    int uiopd2; // second operand of the add, subtract, multiply, or divide
    float res; // result of the add, subtract, multiply, or divide
    double dbl_res24; // result with 24-bit significand, but "unbounded" exponent
// (needed to check tininess, to provide a scaled result to
// an underflow/overflow trap handler, and in flush-to-zero mode)
double dbl_res;  // result in double precision format (needed to avoid a
// double rounding error when denormalizing)
unsigned int result_tiny;
unsigned int result_huge;
unsigned short int sw; // 16 bits
unsigned short int cw; // 16 bits

// have to check first for faults (V, D, Z), and then for traps (O, U, I)

// initialize x87 FPU (floating-point exceptions are masked)
_asm {
    fninit;
}
result_tiny = 0;
result_huge = 0;

switch (exc_env->operation) {
    case ADDPS:
    case ADDSS:
    case SUBPS:
    case SUBSS:
    case MULPS:
    case MULSS:
    case DIVPS:
    case DIVSS:

        uiopd1 = exc_env->operand1_uint32; // copy as unsigned int
        // do not copy as float to avoid conversion
        // of SNaN to QNaN by compiled code
        uiopd2 = exc_env->operand2_uint32;
        // do not copy as float to avoid conversion of SNaN
        // to QNaN by compiled code
        uiopd1 = check_for_daz (uiopd1); // operand1 = +0.0 * operand1 if it is
        // denormal and DAZ=1
        uiopd2 = check_for_daz (uiopd2); // operand2 = +0.0 * operand2 if it is
        // denormal and DAZ=1

        // execute the operation and check whether the invalid, denormal, or
        // divide by zero flags are set and the respective exceptions enabled

        // set control word with rounding mode set to exc_env->rounding_mode,
        // single precision, and all exceptions disabled
        switch (exc_env->rounding_mode) {
            case ROUND_TO_NEAREST:
                cw = 003fH; // round to nearest, single precision, exceptions masked
                break;
            case ROUND_DOWN:
                cw = 043fH; // round down, single precision, exceptions masked
                break;
            case ROUND_UP:
                cw = 083fH; // round up, single precision, exceptions masked
                break;
            case ROUND_TO_ZERO:
                cw = 0c3fH; // round to zero, single precision, exceptions masked
                break;
            default:
                ;
        }
        _asm {
            fldcw WORD PTR cw;
        }

}
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

}  

// compute result and round to the destination precision, with  
// "unbounded" exponent (first IEEE rounding)  
switch (exc_env->operation) {

  case ADDPS:
  case ADDSS:
    // perform the addition
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      faddp st(1), st(0); // may set inexact or invalid status flags
      // store result
      fstp QWORD PTR dbl_res24; // exact
    }
    break;

  case SUBPS:
  case SUBSS:
    // perform the subtraction
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      fsubp st(1), st(0); // may set the inexact or invalid status flags
      // store result
      fstp QWORD PTR dbl_res24; // exact
    }
    break;

  case MULPS:
  case MULSS:
    // perform the multiplication
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      fmulp st(1), st(0); // may set inexact or invalid status flags
      // store result
      fstp QWORD PTR dbl_res24; // exact
    }
    break;

  case DIVPS:
  case DIVSS:
    // perform the division
    __asm {
      fnclex;
      // load input operands
      fld DWORD PTR uiopd1; // may set denormal or invalid status flags
      fld DWORD PTR uiopd2; // may set denormal or invalid status flags
      fdivp st(1), st(0); // may set the inexact, divide by zero, or
                        // invalid status flags
      // store result
      fstp QWORD PTR dbl_res24; // exact
    }
    break;
// read status word
__asm {
  fstsw WORD PTR sw;
}

if (sw & ZERODIVIDE_MASK) sw = sw & ~DENORMAL_MASK; // clear D flag for (denormal / 0)

// if invalid flag is set, and invalid exceptions are enabled, take trap
if (!(exc_env->exc_masks & INVALID_MASK) && (sw & INVALID_MASK)) {
  exc_env->status_flag_invalid_operation = 1;
  exc_env->exception_cause = INVALID_OPERATION;
  return (RAISE_EXCEPTION);
}

// checking for NaN operands has priority over denormal exceptions;
// also fix for the SSE and SSE2
// differences in treating two NaN inputs between the
// instructions and other IA-32 instructions
if (isnanf (uiopd1) || isnanf (uiopd2)) {
  if (isnanf (uiopd1) && isnanf (uiopd2))
    exc_env->result_fval = quietf (uiopd1);
  else
    exc_env->result_fval = (float)dbl_res24; // exact

  if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
  return (DO_NOT_RAISE_EXCEPTION);
}

// if denormal flag set, and denormal exceptions are enabled, take trap
if (!(exc_env->exc_masks & DENORMAL_MASK) && (sw & DENORMAL_MASK)) {
  exc_env->status_flag_denormal_operand = 1;
  exc_env->exception_cause = DENORMAL_OPERAND;
  return (RAISE_EXCEPTION);
}

// if divide by zero flag set, and divide by zero exceptions are
// enabled, take trap (for divide only)
if (!(exc_env->exc_masks & ZERODIVIDE_MASK) && (sw & ZERODIVIDE_MASK)) {
  exc_env->status_flag_divide_by_zero = 1;
  exc_env->exception_cause = DIVIDE_BY_ZERO;
  return (RAISE_EXCEPTION);
}

// done if the result is a NaN (QNaN Indefinite)
res = (float)dbl_res24;
if (isnanf (*(unsigned int *)&res)) {
  exc_env->result_fval = res; // exact
  exc_env->status_flag_invalid_operation = 1;
  return (DO_NOT_RAISE_EXCEPTION);
}

// dbl_res24 is not a NaN at this point
if (sw & DENORMAL_MASK) exc_env->status_flag_denormal_operand = 1;

// Note: (dbl_res24 == 0.0 && sw & PRECISION_MASK) cannot occur
if (-MIN_SINGLE_NORMAL < dbl_res24 && dbl_res24 < 0.0 ||
  0.0 < dbl_res24 && dbl_res24 < MIN_SINGLE_NORMAL) {
result_tiny = 1;
}

// check if the result is huge
if (NEGINFF < dbl_res24 && dbl_res24 < -MAX_SINGLE_NORMAL ||
    MAX_SINGLE_NORMAL < dbl_res24 && dbl_res24 < POSINFF) {
    result_huge = 1;
}

// at this point, there are no enabled I,D, or Z exceptions
// to take; the instr.
// might lead to an enabled underflow, enabled underflow and inexact,
// enabled overflow, enabled overflow and inexact, enabled inexact, or
// none of these; if there are no U or O enabled exceptions, re-execute
// the instruction using IA-32 double precision format, and the
// user's rounding mode; exceptions must have
// been disabled before calling
// this function; an inexact exception may be reported on the 53-bit
// fsbnp, fmulp, or on both the 53-bit and 24-bit conversions, while an
// overflow or underflow (with traps disabled) may be reported on the
// conversion from dbl_res to res

// check whether there is an underflow, overflow,
// or inexact trap to be taken
// if the underflow traps are enabled and the result is
// tiny, take underflow trap
if (!(exc_env->exc_masks & UNDERFLOW_MASK) && result_tiny) {
    dbl_res24 = TWO_TO_192 * dbl_res24; // exact
    exc_env->status_flag_underflow = 1;
    exc_env->exception_cause = UNDERFLOW;
    exc_env->result_fval = (float)dbl_res24; // exact
    if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
    return (RAISE_EXCEPTION);
}

// if overflow traps are enabled and the result is huge, take
// overflow trap
if (!(exc_env->exc_masks & OVERFLOW_MASK) && result_huge) {
    dbl_res24 = TWO_TO_M192 * dbl_res24; // exact
    exc_env->status_flag_overflow = 1;
    exc_env->exception_cause = OVERFLOW;
    exc_env->result_fval = (float)dbl_res24; // exact
    if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;
    return (RAISE_EXCEPTION);
}

// set control word with rounding mode set to exc_env->rounding_mode,
// double precision, and all exceptions disabled
cw = cw | 0200H; // set precision to double
__asm {
    fldcw WORD PTR cw;
}

switch (exc_env->operation) {
    case ADDPS:
    case ADDSS:
        // perform the addition
        __asm {
            // load input operands
            fld DWORD PTR uiopd1; // may set the denormal status flag
            fld DWORD PTR uiopd2; // may set the denormal status flag
            faddp st(1), st(0); // rounded to 53 bits, may set the inexact
            // status flag
/ store result
    fstp  QWORD PTR dbl_res; // exact, will not set any flag
} break;

case SUBPS:
    case SUBSS:
        // perform the subtraction
        __asm {
            // load input operands
            fld DWORD PTR uiopd1; // may set the denormal status flag
            fld DWORD PTR uiopd2; // may set the denormal status flag
            fsubp st(1), st(0); // rounded to 53 bits, may set the inexact
            // status flag
            // store result
            fstp  QWORD PTR dbl_res; // exact, will not set any flag
        } break;

case MULPS:
    case MULSS:
        // perform the multiplication
        __asm {
            // load input operands
            fld DWORD PTR uiopd1; // may set the denormal status flag
            fld DWORD PTR uiopd2; // may set the denormal status flag
            fmulp st(1), st(0); // rounded to 53 bits, exact
            // store result
            fstp  QWORD PTR dbl_res; // exact, will not set any flag
        } break;

case DIVPS:
    case DIVSS:
        // perform the division
        __asm {
            // load input operands
            fld DWORD PTR uiopd1; // may set the denormal status flag
            fld DWORD PTR uiopd2; // may set the denormal status flag
            fdivp st(1), st(0); // rounded to 53 bits, may set the inexact
            // status flag
            // store result
            fstp  QWORD PTR dbl_res; // exact, will not set any flag
        } break;

default:
    ; // will never occur

// calculate result for the case an inexact trap has to be taken, or
// when no trap occurs (second IEEE rounding)
res = (float)dbl_res;
    // may set P, U or O; may also involve denormalizing the result

// read status word
    __asm {
        fstsw WORD PTR sw;
    }

// if inexact traps are enabled and result is inexact, take inexact trap
if (!((exc_env->exc_masks & PRECISION_MASK) &&
       ((sw & PRECISION_MASK) || (exc_env->ftz && result_tiny)))) {
    exc_env->status_flag_inexact = 1;
}
GUIDELINES FOR WRITING SIMD FLOATING-POINT EXCEPTION HANDLERS

```c
exc_env->exception_cause = INEXACT;
if (result_tiny) {
    exc_env->status_flag_underflow = 1;

    // if ftz = 1 and result is tiny, result = 0.0
    // (no need to check for underflow traps disabled: result tiny and
    // underflow traps enabled would have caused taking an underflow
    // trap above)
    if (exc_env->ftz) {
        if (res > 0.0)
            res = ZEROF;
        else if (res < 0.0)
            res = NZEROF;
        // else leave res unchanged
    }
}
if (result_huge) exc_env->status_flag_overflow = 1;
exc_env->result_fval = res;
return (RAISE_EXCEPTION);
}
```

// if it got here, then there is no trap to be taken; the following must
// hold: ((the MXCSR U exceptions are disabled or
// // the MXCSR underflow exceptions are enabled and the underflow flag is
// // clear and (the inexact flag is set or the inexact flag is clear and
// // the 24-bit result with unbounded exponent is not tiny))
// and (the MXCSR overflow traps are disabled or the overflow flag is
// // clear) and (the MXCSR inexact traps are disabled or the inexact flag
// // is clear)
// // in this case, the result has to be delivered (the status flags are
// // sticky, so they are all set correctly already)

// read status word to see if result is inexact
__asm {
    fstsw WORD PTR sw;
}
if (sw & UNDERFLOW_MASK) exc_env->status_flag_underflow = 1;
if (sw & OVERFLOW_MASK) exc_env->status_flag_overflow = 1;
if (sw & PRECISION_MASK) exc_env->status_flag_inexact = 1;

// if ftz = 1, and result is tiny (underflow traps must be disabled),
// result = 0.0
if (exc_env->ftz && result_tiny) {
    if (res > 0.0)
        res = ZEROF;
    else if (res < 0.0)
        res = NZEROF;
    // else leave res unchanged
    exc_env->status_flag_inexact = 1;
    exc_env->status_flag_underflow = 1;
}
exc_env->result_fval = res;
if (sw & ZERODIVIDE_MASK) exc_env->status_flag_divide_by_zero = 1;
if (sw & DENORMAL_MASK) exc_env->status_flag_denormal= 1;
if (sw & INVALID_MASK) exc_env->status_flag_invalid_operation = 1;
return (DO_NOT_RAISE_EXCEPTION);
break;
```

**case CMPPS:**
case CMPSS:
    ...
    break;

case COMISS:
case UCOMISS:
    ...
    break;

case CVTPi2PS:
case CVTSi2SS:
    ...
    break;

case CVTPS2PI:
case CVTSS2SI:
case CVTTPS2PI:
case CVTTSS2SI:
    ...
    break;

case MAXPS:
case MAXSS:
case MINPS:
case MINSS:
    ...
    break;

case SQRTPS:
case SQRTSS:
    ...
    break;

...

case UNSPEC:
    ...
    break;

default:
    ...
}
E.1 INTEL® MEMORY PROTECTION EXTENSIONS (INTEL® MPX)

Intel® Memory Protection Extensions (Intel® MPX) is a new capability introduced into Intel Architecture. Intel MPX can increase the robustness of software when it is used in conjunction with compiler changes to check memory references, for those references whose compile-time normal intentions are usurped at runtime due to buffer overflow or underflow. Two of the most important goals of Intel MPX are to provide this capability at low performance overhead for newly compiled code, and to provide compatibility mechanisms with legacy software components. A direct benefit Intel MPX provides is hardening software against malicious attacks designed to cause or exploit buffer overruns. This chapter describes the software visible interfaces of this extension.

E.2 INTRODUCTION

Intel MPX is designed to allow a system (i.e., the logical processor(s) and the OS software) to run both Intel MPX enabled software and legacy software (written for processors without Intel MPX). When executing software containing a mixture of Intel MPX-unaware code (legacy code) and Intel MPX-enabled code, the legacy code does not benefit from Intel MPX, but it also does not experience any change in functionality or reduction in performance. The performance of Intel MPX-enabled code running on processors that do not support Intel MPX may be similar to the use of embedding NOPs in the instruction stream.

Intel MPX is designed such that an Intel MPX enabled application can link with, call into, or be called from legacy software (libraries, etc.) while maintaining existing application binary interfaces (ABIs). And in most cases, the benefit of Intel MPX requires minimal changes to the source code at the application programming interfaces (APIs) to legacy library/applications. As described later, Intel MPX associates bounds with pointers in a novel manner, and the Intel MPX hardware uses bounds to check that the pointer based accesses are suitably constrained. Intel MPX enabled software is not required to uniformly or universally utilize the new hardware capabilities over all memory references. Specifically, programmers can selectively use Intel MPX to protect a subset of pointers. The code enabled for Intel MPX benefits from memory protection against vulnerability such as buffer overrun. Therefore there is a heightened incentive for software vendors to adopt this technology. At the same time, the security benefit of Intel MPX-protection can be implemented according to the business priorities of software vendors. A software vendor can choose to adopt Intel MPX in some modules to realize partial benefit from Intel MPX quickly, and introduce Intel MPX in other modules in phases (e.g., some programmer intervention might be required at the interface to legacy calls). This adaptive property of Intel MPX is designed to give software vendors control on their schedule and modularity of adoption. It also allows a software vendor to secure defense for higher priority or more attack-prone software first; and allows the use of Intel MPX features in one phase of software engineering (e.g., testing) and not in another (e.g., general release) as dictated by business realities.

The initial goal of Intel MPX is twofold: (1) provide means to defend a system against attacks that originate external to some trust perimeter where the trust perimeter subsumes the system memory and integral data repositories, and (2) provide means to pinpoint accidental logic defects in pointer usage, by undergirding memory references with hardware based pointer validation.

As with any instruction set extensions, Intel MPX can be used by application developers beyond detecting buffer overflow, the processor does not limit the use of Intel MPX for buffer overflow detection.
INTEL® MEMORY PROTECTION EXTENSIONS

E.3 INTEL MPX PROGRAMMING ENVIRONMENT

Intel MPX introduces new **bounds registers** and new instructions that operate on bounds registers. Intel MPX allows an OS to support user mode software (operating at CPL=3) and supervisor mode software (CPL < 3) to add memory protection capability against buffer overrun. It provides controls to enable Intel MPX extensions for user mode and supervisor mode independently. Intel MPX extensions are designed to allow software to associate bounds with pointers, and allow software to check memory references against the bounds associated with the pointer to prevent out of bound memory access (thus preventing buffer overflow). The bounds registers hold lower bound and upper bound that can be checked when referencing memory. An out-of-bounds memory reference then causes a #BR exception. Intel MPX also introduces configuration facilities that the OS must manage to support enabling of user-mode (and/or supervisor-mode) software operations using bounds registers.

E.3.1 Detection and Enumeration of Intel MPX Interfaces

Detection of hardware support for processor extended state component is provided by the main CPUID leaf function 0DH with index ECX = 0. Specifically, the return value in EDX:EAX of CPUID.(EAX=0DH, ECX=0) provides a 64-bit wide bit vector of hardware support of processor state components. If CPUID.(EAX=07H,ECX=0H):EBX.MPX[bit 14] = 1 (the processor supports Intel MPX), CPUID.(EAX=0DH,ECX=0):EAX[bits 4:3] will enumerate the XSAVE state components associated with Intel MPX. These two component states of Intel MPX are the following:

- **BNDREGS**: CPUID.(EAX=0DH,ECX=0):EAX[3] indicates XCR0.BNDREGS[bit 3] is supported. This bit indicates bound register component of Intel MPX state, comprised of four bounds registers, BND0-BND3 (see Appendix E.3.2).
- **BNDCSR**: CPUID.(EAX=0DH,ECX=0):EAX[4] indicates XCR0.BNDCSR[bit 4] is supported. This bit indicates bounds configuration and status component of Intel MPX comprised of BNDCFGU and BNDSTATUS. OS must enable both BNDCSR and BNDREGS bits in XCR0 to ensure full Intel MPX support to applications.

The size of the processor state component, enabled by XCR0.BNDREGS, is enumerated by CPUID.(EAX=0DH,ECX=03H).EAX[31:0] and the byte offset of this component relative to the beginning of the XSAVE/XRSTOR area is reported by CPUID.(EAX=0DH, ECX=03H).EBX[31:0].

The size of the processor state component, enabled by XCR0.BNDCSR, is enumerated by CPUID.(EAX=0DH,ECX=04H).EAX[31:0] and the byte offset of this component relative to the beginning of the XSAVE/XRSTOR area is reported by CPUID.(EAX=0DH, ECX=04H).EBX[31:0].

On processors that support Intel MPX, CPUID.(EAX=0DH,ECX=0):EAX[3] and CPUID.(EAX=0DH,ECX=0):EAX[4] will both be 1. On processors that do not support Intel MPX, CPUID.(EAX=0DH,ECX=0):EAX[3] and CPUID.(EAX=0DH,ECX=0):EAX[4] will both be 0.

The layout of XCR0 for extended processor state components defined in Intel Architecture is shown in Figure 2-8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Enabling Intel MPX requires an OS to manage bits [4:3] of XCR0; see Section 13.5.

The BNDLDX and BNDSTX instructions (Appendix E.4.3) each take an operand whose bits are used to traverse data structures in memory. In 64-bit mode, these instructions operate only on the lower bits in the supplied 64-bit addresses. The number of bits used is 48 plus a value called the **MPX address-width adjust (MAWA)**. The MAWA value depends on CPL:

- If CPL < 3, the supervisor MAWA (MAWAS) is used. This value is 0.
- If CPL = 3, the user MAWA (MAWAU) is used. The value of MAWAU is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17].

(Outside of 64-bit mode, BNDLDX, and BNDSTX use the entire 32 bits of the supplied linear-address operands.)

E.3.2 Bounds Registers

Intel MPX Architecture defines four new registers, BND0-BND3, which Intel MPX instructions operate on. Each bounds register stores a pair of 64-bit values which are the lower bound (LB) and upper bound (UB) of a buffer, see Figure E-1.
The bounds are unsigned effective addresses, and are inclusive. The upper bounds are architecturally represented in 1’s complement form. Lower bound = 0, and upper bound = 0 (1’s complement of all 1s) will allow access to the entire address space. The bounds are considered as INIT when both lower and upper bounds are 0 (cover the entire address space). The two Intel MPX instructions which operate on the upper bound (BNDMK and BNDCU) account for the 1’s complement representation of the upper bounds.

The instruction set does not impose any conventions on the use of bounds registers. Software has full flexibility associating pointers to bounds registers including sharing them for multiple pointers. RESET or INIT# will initialize (write zero to) BND0–BND3.

### E.3.3 Configuration and Status Registers

Intel MPX defines two configuration registers and one status register. The two configuration registers are defined for user mode (CPL = 3) and supervisor mode (CPL < 3). The user-mode configuration register BNDCFGU is accessible only with the XSAVE feature set instructions.

The supervisor mode configuration register is an MSR, referred to as IA32_BNDCFGS (MSR 0D90H). Because both configuration registers share a common layout (see Figure E-2), when describing the common behavior, these configuration registers are often denoted as BNDCFGx, where x can be U or S, for user and supervisor mode respectively.

The Enable bit in BNDCFGU enables Intel MPX in user mode (CPL = 3), and the Enable bit in BNDCFGS enables Intel MPX in supervisor mode (CPL < 3). The BNDPRESERVE bit controls the initialization behavior of CALL/RET/JMP/Jcc instructions without the BND (F2H) prefix; see Appendix E.5.3.

WRMSR to BNDCFGS will #GP if any of the reserved bits of BNDCFGS is not zero or if the base address of the bound directory is not canonical. XRSTOR of BNDCFGU ignores the reserved bits and does not fault if any is non-zero; similarly, it ignores the upper bits of the base address of the bound directory and sign-extends the highest implemented bit of the linear address to guarantee the canonicality of this address.

Intel MPX also defines a status register (BNDSTATUS) primarily used to communicate status information for #BR exception. The layout of the status register is shown in Figure E-3.
The BNDSTATUS register provides two fields to communicate the status of Intel MPX operations:

- **EC (bits 1:0):** The error code field communicates status information of a bound range exception #BR or operation involving bound directory.
- **ABD: (bits 63:2):** The address field of a bound directory entry can provide information when operation on the bound directory caused a #BR.

The valid error codes are defined in Table E-1.

**Table E-1. Error Code Definition of BNDSTATUS**

<table>
<thead>
<tr>
<th>EC</th>
<th>Description</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>No Intel MPX exception</td>
<td>No exception caused by Intel MPX operations.</td>
</tr>
<tr>
<td>01b</td>
<td>Bounds violation</td>
<td>#BR caused by BNDCL, BNDCU or BNDCN instructions; ABD is 0.</td>
</tr>
<tr>
<td>10b</td>
<td>Invalid BD entry</td>
<td>#BR caused by BNDLDX or BNDSTX instructions, ABD will be set to the linear address of the invalid bound-directory entry</td>
</tr>
<tr>
<td>11b</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**NOTES:**

1. When legacy BOUND instruction cause a #BR with Intel MPX enabled (see Appendix E.5.4), EC is written with Zero.

RESET or INIT# will set BNDCFGx and BNDSTATUS registers to zero.

### E.3.4 Read and Write of IA32_BNDCFGS

The RDMSR and WRMSR instructions can be used to read and write the IA32_BNDCFGS MSR. (The XSAVE state does not include IA32_BNDCFGS, and instructions in the XSAVE feature set do not access that register). Attempts to write to IA32_BNDCFGS check for canonicality of the addresses being loaded into IA32_BNDCFGS (regardless of mode at the time of execution) and will #GP if the address is not canonical or if reserved bits would be set.

Software can use RDMSR and WRMSR to read and write IA32_BNDCFGS as long as the processor implements Intel MPX, i.e., CPUID.(EAX=07H, ECX=0H).EBX.MPX = 1. The states of CR4 and XCR0 have no impact on the ability to access IA32_BNDCFGS.

### E.4 INTEL MPX INSTRUCTION SUMMARY

When Intel MPX is not enabled or not present, all Intel MPX instructions behave as NOP. There are eight Intel MPX instructions, Table E-2 provides a summary.

A C/C++ compiler can implement intrinsic support for Intel MPX instructions to facilitate pointer operation with capability of checking for valid bounds on pointers. Typically, Intel MPX intrinsics are implemented by compiler via inline code generation where bounds register allocations are handled by the compiler without requiring the
programmer to directly manipulate any bounds registers. Therefore no new data type for a bounds register is needed in the syntax of Intel MPX intrinsics.

### E.4.1 Instruction Encoding

All Intel MPX instructions are NOP on processors that report CPUID.(EAX=07H, ECX=0H).EBX.MPX [bit 14] = 0, or if Intel MPX is not enabled by the operating system (see Section 13.5). Applications can selectively opt-in to use Intel MPX instructions.

All Intel MPX opcodes encoded to operate on BND0-BND3 are valid Intel MPX instructions. All Intel MPX opcodes encoded to operate on bound registers beyond BND3 will #UD if Intel MPX is enabled.

BNDLDX/BNDSTX opcodes require 66H as a mandatory prefix with its operand size tied to the address size attribute of the supported operating modes. Attempt to override operand size attribute with 66H or with REX.W in 64-bit mode is ignored.

### E.4.2 Usage and Examples

**BNDMK** is typically used after memory is allocated for a buffer, e.g., by functions such as malloc, calloc, or when the memory is allocated on the stack. However, many other usages are possible such as when accessing an array member of a structure.

**Example E-1. BNDMK Example Usage in Application and Library Code**

```c
int A[100]; // assume the array A is allocated on the stack at 'offset' from RBP.
// the instruction to store starting address of array will be:
LEA RAX, [RBP+offset]
// the instruction to create the bounds for array A will be:
BNDMK BND0, [RAX+399]
// Store RAX into BND0.LB, and ~(RAX+399) into BND0.UB.
```

// similarly, for a library implementation of dynamic allocated memory
```
int * k = malloc(100);
// assuming that malloc returns pointer k in RAX and holds (size - 1) in RCX
// the malloc implementation will execute the following instruction before returning:
BNDMK BND0, [RAX+RCX]
// BND0.LB stores RAX, and BND0.UB stores ~(RAX+RCX)
```

**BNDMOV** is typically used to copy bounds from one bound register to another when a pointer is copied from one general purpose register to another, or to spill/fill bounds into memory corresponding to a spill/fill of a pointer.

**Example E-2. BNDMOV Example**

Spilling or caller save of bound register would use BNDMOV [RBP+ offset], BNDx.

Assuming that the calling convention is that bound of first pointer is passed in BND0, and that bound happens to be in BND3 before the call, the software will add instruction BNDMOV BND0, BND3 prior to the call.

### Table E-2. Intel MPX Instruction Summary

<table>
<thead>
<tr>
<th>Intel MPX Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNDMK b, m</td>
<td>Create LowerBound (LB) and UpperBound (UB) in the bounds register b</td>
</tr>
<tr>
<td>BNDCL b, r/m</td>
<td>Checks the address of a memory reference or address in r against the lower bound</td>
</tr>
<tr>
<td>BNDUC b, r/m</td>
<td>Checks the address of a memory reference or address in r against the upper bound in 1’s complement form</td>
</tr>
<tr>
<td>BNDCN b, r/m</td>
<td>Checks the address of a memory reference or address in r against the upper bound not in 1’s complement form</td>
</tr>
<tr>
<td>BNDMOV b, b/m</td>
<td>Copy/load LB and UB bounds from memory or a bounds register</td>
</tr>
<tr>
<td>BNDMOV b/m, b</td>
<td>Store LB and UB bounds in a bounds register to memory or another register</td>
</tr>
<tr>
<td>BNDLDX b, mib</td>
<td>Load bounds using address translation using an sib-addressing expression mib</td>
</tr>
<tr>
<td>BNDSTX mib, b</td>
<td>Store bounds using address translation using an sib-addressing expression mib</td>
</tr>
</tbody>
</table>
INTEL® MEMORY PROTECTION EXTENSIONS

BNDCL/BNDCU/BNDCN are typically used before writing to a buffer but can be used in other instances as well. If there are no bounds violations as a result of bound check instruction, the processor will proceed to execute the next instruction. However, if the bound check fails, it will signal #BR exception (fault).

Typically, the pointer used to write to memory will be compared against lower bound. However, for upper bound check, the software must add the (operand size - 1) to the pointer before upper bound checking.

For example, the software intends to write a 32-bit integer in 64-bit mode into a buffer at address specified in RAX, and the bounds are in register BND0, the instruction sequence will be:

BNDCL BND0, [RAX]
BNDCU BND0, [RAX+3] ; operand size is 4
MOV Dword ptr [RAX], RBX ; RBX has the data to be written to the buffer.

Software may move one of the two bound checks out of a loop if it can determine that memory is accessed strictly in ascending or descending order. For string instructions of the form REP MOVS, the software may choose to do check lower bound against first access and upper bound against last access to memory. However, if software wants to also check for wrap around conditions as part of address computation, it should check for both upper and lower bound for first and last instructions (total of four bound checks).

BNDSTX is used to store the bounds associated with a buffer and the “pointer value” of the pointer to that buffer onto a bound table entry via address translation using a two-level structure, see Appendix E.4.3.

For example, the software has a buffer with bounds stored in BND0, the pointer to the buffer is in ESI, the following sequence will store the “pointer value” (the buffer) and the bounds into a configured bound table entry using address translation from the linear address associated with the base of a SIB-addressing form consisting of a base register and a index register:

MOV ECX, Dword ptr [ESI] ; store the pointer value in the index register ECX
MOV EAX, ESI ; store the pointer in the base register EAX
BNDSTX Dword ptr [EAX+ECX], BND0 ; perform address translation from the linear address of the base EAX and store bounds and pointer value ECX onto a bound table entry.

Similarly to retrieve a buffer and its associated bounds from a bound table entry:

MOV EAX, dword ptr [EBX] ;
BNDLDX BND0, dword ptr [EBX+EAX] ; perform address translation from the linear address of the base EBX, and loads bounds and pointer value from a bound table entry

E.4.3 Loading and Storing Bounds in Memory

Intel MPX defines two instructions to load and store of the linear address of a pointer to a buffer, along with the bounds of the buffer into a data structure of extended bounds. When storing these extended bounds, the processor parses the address of the pointer (where it is stored) to locate an entry in a bound table in which to store the extended bounds. Loading of an extended bounds performs the reverse sequence.

The memory representation of an extended bound is a 4-tuple consisting of lower bound, upper bound, pointer value and a reserved field (for use by future versions of Intel MPX; software must not use this field). Accesses to these extended bounds use 32-bit or 64-bit operands according to the current paging mode. Thus, a bound table entry is 4*64 bits (32 bytes) in 64-bit mode and 4*32 bits (16 bytes) outside 64-bit mode. The linear address of a bound table is stored in a bound-directory entry (BDE). The linear address of the bound directory is derived from either BNDCFGU (CPL = 3) or BNDCFGS (CPL < 3).

The bound directory and bound tables are stored in application memory and are allocated by the application (in case of kernel use, the structures will be in kernel memory). The bound directory and each bound table are in contiguous linear memory.

Software should take care to allocate sufficient memory for the bound directory and the bound tables. The amount of memory required depends on the current operating mode and, in some cases, on CPL:

• In 64-bit mode:
  — Each bound table comprises $2^{17}$ 32-byte entries thus, the size of a bound table in 64-bit mode is 4 MBytes.
The size of the bound directory depends on the value of MAWA. Specifically, the bound directory comprises $2^{28+\text{MAWA}}$ 64-bit entries; thus, the size of a bound directory in 64-bit mode is $2^{1+\text{MAWA}}$ GBytes. The value of MAWA depends on CPL:

- If CPL < 3, the supervisor MAWA (MAWAS) is used. This value is 0. Thus, when CPL < 3, a bound directory comprises $2^{28}$ 64-bit entries and the size of a bound directory is 2 GBytes.
- If CPL = 3, the user MAWA (MAWAU) is used. The value of MAWAU is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. When CPL = 3, a bound directory comprises $2^{28+\text{MAWAU}}$ 64-bit entries and the size of a bound directory is $2^{1+\text{MAWAU}}$ GBytes.

**NOTE**

Software operating with CPL = 3 in 64-bit mode should use CPUID to determine the proper amount of memory to allocate for the bound directory.

* Outside 64-bit mode:
  - Each bound table comprises $2^{10}$ 16-byte entries; thus, the size of a bound table outside 64-bit mode is 16 KBytes.
  - The bound directory comprises $2^{20}$ 32-bit entries; thus, the size of a bound directory outside 64-bit mode is 4 MBytes. This size is independent of MAWA and CPL.

Bounds in memory are associated with the memory address where the pointer is stored, i.e., Ap. A linear address LAp is computed by adding the appropriate segment base to Ap. (Note: for these instructions, the segment override applies only to the computation.) Appendix E.4.3.1 and Appendix E.4.3.2 describe how BNDLDX and BNDSTX parse LAp to locate a bound-directory entry (BDE), which contains the address of a bound table, and then a bound-table entry (BTE), which contains the extended bounds for the pointer.

### E.4.3.1 BNDLDX and BNDSTX in 64-Bit Mode

Figure E-4 shows the two-level structures for address translation of extended bounds in 64-bit mode.

---

**Figure E-4. Bound Paging Structure and Address Translation in 64-Bit Mode**
As noted earlier, the linear address of the bound directory is derived from either BNDCFGU (CPL = 3) or BNDCFGS (CPL < 3). In 64-bit mode, each bound-directory entry (BDE) is 8 bytes. The number of entries in the bound directory is determined by the MPX address-width adjust (MAWA; see Appendix E.3.1). Specifically, the number of entries is $2^{28+\text{MAWA}}$.

In 64-bit mode, the processor uses the two-level structures to access extended bounds as follows:

- A bound directory is located at the 4-KByte aligned linear address specified in bits 63:12 of BNDCFGx (see Figure E-2). A bound directory comprises $2^{28+\text{MAWA}}$ 64-bit entries (BDEs); thus, the size of a bound directory in 64-bit mode is $2^{1+\text{MAWA}}$ GBytes. A BDE is selected using the LAp (linear address of pointer to a buffer) to construct a 64-bit offset as follows:
  - bits 63:31+MAWA are 0;
  - bits 30+MAWA:3 are LAp[47+MAWA:20]; and
  - bits 2:0 are 0.
  The address of the BDE is the sum of the bound-directory base address (from BNDCFGx) plus this 64-bit offset.

- Bit 0 of a BDE is a valid bit. If this bit is 0, use of the BDE by BNDLDX or BNDSTX causes #BR, sets BNDSTATUS[1:0] to 10b (the error code), and loads BNDSTATUS[63:2] with bits 63:2 of the linear address of the BDE. Otherwise, the processor uses bits 63:3 of the BDE as the 8-byte aligned address of a bound table (BT); the processor ignores bits 2:1 of a BDE.

  A bound table comprises $2^{17}$ 32-byte entries (BTEs); thus, the size of a bound table in 64-bit mode is 4 MBytes. A BTE is selected using the LAp (linear address of pointer to a buffer) to construct an offset as follows:
  - bits 21:5 are LAp[19:3]; and
  - bits 4:0 are 0.

  The address of the BTE is the sum of the bound-table base address (from the BDE) plus this offset.

- Each BTE comprises the following:
  - a 64-bit lower bound (LB) field;
  - a 64-bit upper bound (UB) field;
  - a 64-bit pointer value; and
  - a 64-bit reserved field. This field is reserved for future Intel MPX; software must not use it.

E.4.3.2  BNDLDX and BNDSTX Outside 64-Bit Mode

Figure E-5 shows the two-level structures for address translation of extended bounds outside 64-bit mode.

As noted earlier, the linear address of the bound directory is derived from either BNDCFGU (CPL = 3) or BNDCFGS (CPL < 3). Outside 64-bit mode, each bound-directory entry (BDE) is 4 bytes. The number of entries in the bound directory is $2^{20}$.

Outside 64-bit mode, the processor uses the two-level structures to access extended bounds as follows:

- A bound directory is located at the 4-KByte aligned linear address specified in bits 31:12 of BNDCFGx (see Figure E-2). A bound directory comprises $2^{20}$ 32-bit entries (BDEs); thus, the size of a bound directory outside 64-bit mode is 4 MBytes. A BDE is selected using the LAp (linear address of pointer to a buffer) to construct an offset as follows:
  - bits 21:2 are LAp[31:12]; and
  - bits 1:0 are 0.

  The address of the BDE is the sum of the bound-directory base address (from BNDCFGx) plus this offset.

- Bit 0 of a BDE is a valid bit. If this bit is 0, use of the BDE by BNDLDX or BNDSTX causes #BR, sets BNDSTATUS[1:0] to 10b (the error code), and loads BNDSTATUS[31:2] with bits 31:2 of the linear address of the BDE. Otherwise, the processor uses bits 31:2 of the BDE as the 4-byte aligned address of a bound table (BT); the processor ignores bit 1 of a BDE.
A bound table comprises $2^{10}$ 16-byte entries (BTEs); thus, the size of a bound table outside 64-bit mode is 16 KBytes. A BTE is selected using the LAp (linear address of pointer to a buffer) to construct an offset as follows:
- bits 13:4 are LAp[11:2]; and
- bits 3:0 are 0.

The address of the BTE is the sum of the bound-table base address (from the BDE) plus this offset. This address is use as an offset into the DS segment to determine the linear address of the BTE.

- Each BTE comprises the following:
  - a 32-bit lower bound (LB) field;
  - a 32-bit upper bound (UB) field;
  - a 32-bit pointer value; and
  - a 32-bit reserved field. This field is reserved for future Intel MPX; software must not use it.

### E.5 INTERACTIONS WITH INTEL MPX

#### E.5.1 Intel MPX and Operating Modes

In 64-bit Mode, all Intel MPX instructions use 64-bit operands for bounds and 64 bit addressing, i.e., REX.W & 67H have no effect on data or address size.

XSAVE, XSAVEOPT, and XRSTOR load/store 64-bit values in all modes, as these state-management instructions are not Intel MPX instructions.
In compatibility and legacy modes (including 16-bit code segments, real and virtual 8086 modes) all Intel MPX instructions use 32-bit operands for bounds and 32-bit addressing. The upper 32-bits of destination bound register are cleared (consistent with behavior of integer registers).

In 32-bit and compatibility mode, the bounds are 32-bit, and are treated same as 32-bit integer registers. Therefore, when 32-bit bound is updated in a bound register, the upper 32-bits are undefined. When switching from 64-bit, the behavior of content of bounds register will be similar to that of general purpose registers.

Table E-3 describes the impact of 67H prefix on memory forms of Intel MPX instructions (register-only forms ignore 67H prefix) when Intel MPX is enabled:

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>67H Prefix</th>
<th>Effective Address Size used for Intel MPX instructions when Intel MPX is enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit Mode</td>
<td>Y</td>
<td>64 bit addressing used</td>
</tr>
<tr>
<td>64-bit Mode</td>
<td>N</td>
<td>64 bit addressing used</td>
</tr>
<tr>
<td>32-bit Mode</td>
<td>Y</td>
<td>#UD</td>
</tr>
<tr>
<td>32-bit Mode</td>
<td>N</td>
<td>32 bit addressing used</td>
</tr>
<tr>
<td>16-bit Mode</td>
<td>Y</td>
<td>32 bit addressing used</td>
</tr>
<tr>
<td>16-bit Mode</td>
<td>N</td>
<td>#UD</td>
</tr>
</tbody>
</table>

**E.5.2 Intel® MPX Support for Pointer Operations with Branching**

Intel MPX provides flexibility in supporting pointer operation across control flow changes. Intel MPX allows:

- compatibility with legacy code that may perform pointer operation across control flow changes and are unaware of Intel MPX, along with
- Intel MPX-aware code that adds bounds checking protection to pointer operation across control flow changes.

The interface to provide such flexibility consists of:

- Using a prefix, referred to as BND prefix, to relevant branch instructions: CALL, RET, JMP, and Jcc.
- BNDCFGU and BNDCFGS provides the bit field, BNDPRESERVE (bit 1).

The value of BNDPRESERVE in conjunction with the presence/absence the BND prefix with those branching instruction will determine whether the values in BND0-BND3 will be initialized or unchanged.

**E.5.3 CALL, RET, JMP, and All Jcc**

An application compiled to use Intel MPX will use the REPNE (F2H) prefix (denoted by BND) for all forms of near CALL, near RET, near JMP, short & near Jcc instructions (BND+CALL, BND+RET, BND+JMP, BND+Jcc). See Table E-4 for specific opcodes. All far CALL, RET, and JMP instructions plus short JMP (JMP rel 8, opcode EB) instructions will never cause bound registers to be initialized.

If BNDPRESERVE bit is one, above instructions will NOT INIT the bounds registers when BND prefix is not present for above instructions (legacy behavior). However, If BNDPRESERVE is zero, above instructions will INIT ALL bound registers (BND0-BND3) when BND prefix is not present for above instructions. If BND prefix is present for above instructions, the BND registers will NOT INIT any bound registers (BND0-BND3).

The legacy code will continue to use non-prefixed forms of these instructions, so if BNDPRESERVE is zero, all the bound registers will INIT by legacy code. This allows the legacy function to execute and return to callee with all bound registers initialized (legacy code by definition cannot make or load bounds in bound registers because it does not have Intel MPX instructions). This will eliminate compatibility concerns when legacy function might have changed the pointer in registers but did not update the value of the bounds registers associated with these pointers.

If BNDCFGx.BNDPRESERVE is clear then non-prefixed forms of these instructions will initialize all the bound registers. If this bit is set then non-prefixed and prefixed forms of these instructions will preserve the contents of bound registers as shown in Table E-4.
E.5.4 BOUND Instruction and Intel MPX

If Intel MPX in enabled (see Section 13.5) and a #BR was caused due to a BOUND instruction, then BOUND instruction will write zero to the BNDSTATUS register. In all other situations, BOUND instruction will not modify BNDSTATUS. Specifically, the operation of the BOUND instruction can be described as:

\[
\begin{align*}
\text{IF } & ((\text{BOUND instruction caused #BR}) \ \text{AND} \ \text{CR4.OXXSAVE} = 1 \ \text{AND} \ \text{XCR0.BNDREGS} = 1 \ \text{AND} \ \text{XCR0.BNDCSR} = 1) \ \text{AND} \\
& (\text{CPL} = 3 \ \text{AND} \ \text{BNDCFGU.ENABLE} = 1) \ \text{OR} \ (\text{CPL} < 3 \ \text{AND} \ \text{BNDCFGS.ENABLE} = 1)) \ \text{THEN} \ \\
& \text{BNDSTATUS} := 0; \\
\text{ELSE} \ \\
& \text{BNDSTATUS} \text{ is not modified;} \\
\text{FI;}
\end{align*}
\]

E.5.5 Programming Considerations

Intel MPX instruction set does not dictate any calling convention, but allows the calling convention extensions to be interoperable with legacy code by making use of the of the bound registers and the bound tables to convey arguments and return values.

E.5.6 Intel MPX and System Management Mode

Upon delivery of an SMI to a processor supporting Intel MPX, the contents of IA32_BNDCFGS is saved to SMM state save map (at offset 7ED0H) and the register is then cleared when entering into SMM. RSM restores IA32_BNDCFGS from the SMM state save map. The instruction forces the reserved bits (11:2) to 0 and sign-extends the highest implemented bit of the linear address to guarantee the canonicality of this address (regardless of what is in SMM state save map).

The content of IA32_BNDCFGS is cleared after entering into SMM. Thus, Intel MPX is disabled inside an SMM handler until SMM code enables it explicitly. This will prevent initialization of the bound registers by execution of CALL, RET, JMP, or Jcc in SMM code.

E.5.7 Support of Intel MPX in VMCS

A new guest-state field for IA32_BNDCFGS is added to the VMCS. In addition, two new controls are added:

- a VM-exit control called "clear BNDCFGS"
- a VM-entry control called "load BNDCFGS."

Table E-4. Bounds Register INIT Behavior Due to BND Prefix with Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Branch Instruction Opcodes</th>
<th>BNDPRESERVE = 0</th>
<th>BNDPRESERVE = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>E8, FF/2</td>
<td>Init BND0-BND3</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>BND + CALL</td>
<td>F2 E8, F2 FF/2</td>
<td>BND0-BND3 unchanged</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>RET</td>
<td>C2, C3</td>
<td>Init BND0-BND3</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>BND + RET</td>
<td>F2 C2, F2 C3</td>
<td>BND0-BND3 unchanged</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>JMP</td>
<td>E9, FF/4</td>
<td>Init BND0-BND3</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>BND + JMP</td>
<td>F2 E9, F2 FF/4</td>
<td>BND0-BND3 unchanged</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>Jcc</td>
<td>70 through 7F, OF 80 through 0F 8F</td>
<td>Init BND0-BND3</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>BND + Jcc</td>
<td>F2 70 through F2 7F, F2 0F 80 through F2 0F 8F</td>
<td>BND0-BND3 unchanged</td>
<td>BND0-BND3 unchanged</td>
</tr>
</tbody>
</table>
VM exits always save IA32_BNDCFGS into BNDCFGS field of VMCS; if “clear BNDCFGS” is 1, VM exits clear IA32_BNDCFGS. If “load BNDCFGS” is 1, VM entry loads IA32_BNDCFGS from VMCS. If loading IA32_BNDCFGS, VM entry should check the value of that register in the guest-state area of the VMCS and cause the VM entry to fail (late) if the value is one that would causes WRMSR to fault if executed in ring 0.

E.5.8 Support of Intel MPX in Intel TSX

For some processor implementations, the following Intel MPX instructions may always cause transactional aborts:

• An Intel TSX transaction abort will occur in case of legacy branch (that causes bounds registers INIT) when at least one bounds register was in a NON-INIT state.

• An Intel TSX transaction abort will occur in case of a BNDLDX & BNDSTX instruction on non-flat segment.

Intel MPX Instructions (including BND prefix + branch instructions) not enumerated above as causing transactional abort when used inside a transaction will typically not cause an Intel TSX transaction to abort.
NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: Basic Architecture, Order Number 253665; Instruction Set Reference, A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.
Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation. No product or component can be absolutely secure. Your costs and results may vary. You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein. All product plans and roadmaps are subject to change without notice. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade. Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document. © Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
## CONTENTS

### CHAPTER 1

#### ABOUT THIS MANUAL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2 OVERVIEW OF VOLUME 2A, 2B, 2C, AND 2D: INSTRUCTION SET REFERENCE</td>
<td>1-4</td>
</tr>
<tr>
<td>1.3 NOTATIONAL CONVENTIONS</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.1 Bit and Byte Order</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.2 Reserved Bits and Software Compatibility</td>
<td>1-6</td>
</tr>
<tr>
<td>1.3.3 Instruction Operands</td>
<td>1-6</td>
</tr>
<tr>
<td>1.3.4 Hexadecimal and Binary Numbers</td>
<td>1-7</td>
</tr>
<tr>
<td>1.3.5 Segmented Addressing</td>
<td>1-7</td>
</tr>
<tr>
<td>1.3.6 Exceptions</td>
<td>1-7</td>
</tr>
<tr>
<td>1.3.7 A New Syntax for CPUID, CR, and MSR Values</td>
<td>1-7</td>
</tr>
<tr>
<td>1.4 RELATED LITERATURE</td>
<td>1-8</td>
</tr>
</tbody>
</table>

#### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

- Intel® AVX Instructions and the Upper 128-bits of YMM registers: 2-19
- 3-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3] - ‘vvv’ the Source or Dest Register Specifier: 2-16
- 2-byte VEX Byte 1, bit[2], and 3-byte VEX Byte 2, bit [2] - “L” : 2-18
- 2-byte VEX Byte 1, bit[1:0], and 3-byte VEX Byte 2, bits [1:0] - “pp” : 2-19
- The Opcode Byte : 2-19
- The ModRM, SIB, and Displacement Bytes : 2-19
- The Third Source Operand (Immediate Byte) : 2-19
- Intel® AVX Instructions and the Upper 128-bits of YMM registers : 2-19
- Vector Length Transition and Programming Considerations : 2-19
- Intel® AVX Instruction Length : 2-20
- Vector SIB (VSIB) Memory Addressing : 2-20
- 64-bit Mode VSIB Memory Addressing : 2-21
- INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX) : 2-21
- INTEL® AVX AND INTEL® SSE INSTRUCTION EXCEPTION CLASSIFICATION : 2-22
- Exceptions Type 1 (Aligned Memory Reference) : 2-27
- Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned) : 2-28

### CHAPTER 2

#### INSTRUCTION FORMAT

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.1 Instruction Prefixes</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.2 OpCodes</td>
<td>2-3</td>
</tr>
<tr>
<td>2.1.3 Mod/R/M and SIB Bytes</td>
<td>2-3</td>
</tr>
<tr>
<td>2.1.4 Displacement and Immediate Bytes</td>
<td>2-3</td>
</tr>
<tr>
<td>2.1.5 Addressing-Mode Encoding of Mod/R/M and SIB Bytes</td>
<td>2-4</td>
</tr>
<tr>
<td>2.2 IA-32E MODE</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.1 REX Prefixes</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.1.1 Encoding</td>
<td>2-8</td>
</tr>
<tr>
<td>2.2.1.2 More on REX Prefix Fields</td>
<td>2-8</td>
</tr>
<tr>
<td>2.2.1.3 Displacement</td>
<td>2-11</td>
</tr>
<tr>
<td>2.2.1.4 Direct Memory-Offset MOVs</td>
<td>2-11</td>
</tr>
<tr>
<td>2.2.1.5 Immediats</td>
<td>2-11</td>
</tr>
<tr>
<td>2.2.1.6 RIP-Relative Addressing</td>
<td>2-12</td>
</tr>
<tr>
<td>2.2.1.7 Default 64-Bit Operand Size</td>
<td>2-12</td>
</tr>
<tr>
<td>2.2.2 Additional Encodings for Control and Debug Registers</td>
<td>2-12</td>
</tr>
<tr>
<td>2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)</td>
<td>2-13</td>
</tr>
<tr>
<td>2.3.1 Instruction Format</td>
<td>2-13</td>
</tr>
<tr>
<td>2.3.2 VEX and the LOCK prefix</td>
<td>2-13</td>
</tr>
<tr>
<td>2.3.3 VEX and the 66H, F2H, and F3H prefixes</td>
<td>2-13</td>
</tr>
<tr>
<td>2.3.4 VEX and the REX prefix</td>
<td>2-13</td>
</tr>
<tr>
<td>2.3.5 The VEX Prefix</td>
<td>2-14</td>
</tr>
<tr>
<td>2.3.5.1 VEX Byte 0, bits[7:0]</td>
<td>2-15</td>
</tr>
<tr>
<td>2.3.5.2 VEX Byte 1, bit [7] - ‘R’</td>
<td>2-15</td>
</tr>
<tr>
<td>2.3.5.3 3-byte VEX byte 1, bit[6] - ‘X’</td>
<td>2-16</td>
</tr>
<tr>
<td>2.3.5.4 3-byte VEX byte 1, bit[5] - ‘B’</td>
<td>2-16</td>
</tr>
<tr>
<td>2.3.5.5 3-byte VEX byte 2, bit [7] - ‘W’</td>
<td>2-16</td>
</tr>
<tr>
<td>2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3] - ‘vvv’ the Source or Dest Register Specifier</td>
<td>2-16</td>
</tr>
<tr>
<td>2.3.6 Instruction Operand Encoding and VEX.vvvv, Mod/R/M</td>
<td>2-17</td>
</tr>
<tr>
<td>2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmm”</td>
<td>2-18</td>
</tr>
<tr>
<td>2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2] - “L” : 2-18</td>
<td></td>
</tr>
<tr>
<td>2.3.6.3 2-byte VEX byte 1, bit[1:0], and 3-byte VEX Byte 2, bits [1:0] - “pp” : 2-19</td>
<td></td>
</tr>
<tr>
<td>2.3.7 The Opcode Byte</td>
<td>2-19</td>
</tr>
<tr>
<td>2.3.8 The Mod/R/M, SIB, and Displacement Bytes</td>
<td>2-19</td>
</tr>
<tr>
<td>2.3.9 The Third Source Operand (Immediate Byte)</td>
<td>2-19</td>
</tr>
<tr>
<td>2.3.10 Intel® AVX Instructions and the Upper 128-bits of YMM registers</td>
<td>2-19</td>
</tr>
<tr>
<td>2.3.10.1 Vector Length Transition and Programming Considerations</td>
<td>2-19</td>
</tr>
<tr>
<td>2.3.11 Intel® AVX Instruction Length</td>
<td>2-20</td>
</tr>
<tr>
<td>2.3.12 Vector SIB (VSIB) Memory Addressing</td>
<td>2-20</td>
</tr>
<tr>
<td>2.3.12.1 64-bit Mode VSIB Memory Addressing</td>
<td>2-21</td>
</tr>
<tr>
<td>2.4 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)</td>
<td>2-21</td>
</tr>
<tr>
<td>2.5 INTEL® AVX AND INTEL® SSE INSTRUCTION EXCEPTION CLASSIFICATION</td>
<td>2-22</td>
</tr>
<tr>
<td>2.5.1 Exceptions Type 1 (Aligned Memory Reference)</td>
<td>2-27</td>
</tr>
<tr>
<td>2.5.2 Exceptions Type 2 (&gt;=16 Byte Memory Reference, Unaligned)</td>
<td>2-28</td>
</tr>
</tbody>
</table>
CONTENTS

2.5.3 Exceptions Type 3 (<16 Byte Memory Argument) ................................................................. 2-29
2.5.4 Exceptions Type 4 (>=16 Byte Mem Arg, No Alignment, No Floating-point Exceptions) .... 2-30
2.5.5 Exceptions Type 5 (<16 Byte Mem Arg and No FP Exceptions) ........................................ 2-31
2.5.6 Exceptions Type 6 (VEX-Encoded Instructions without Legacy SSE Analogues) .............. 2-32
2.5.7 Exceptions Type 7 (No FP Exceptions, No Memory Arg) .................................................... 2-33
2.5.8 Exceptions Type 8 (AVX and No Memory Argument) ........................................................ 2-33
2.5.9 Exceptions Type 11 (VEX-only, Mem Arg, No AC, Floating-point Exceptions) ................. 2-34
2.5.10 Exceptions Type 12 (VEX-only, VSIB Mem Arg, No AC, No Floating-point Exceptions) .. 2-35
2.6 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS ....................................................... 2-35
2.6.1 Exceptions Type 13 (VEX-Encoded GPR Instructions) ...................................................... 2-36
2.7 INTEL® AVX-512 ENCODING .............................................................................................. 2-36
2.7.1 Instruction Format and EVEX ............................................................ 2-37
2.7.2 Register Specifier Encoding and EVEX ................................................................. 2-39
2.7.3 Opmask Register Encoding ............................................................................................. 2-40
2.7.4 Masking Support in EVEX ............................................................................................. 2-40
2.7.5 Compressed Displacement (disp8*N) Support in EVEX .................................................. 2-41
2.7.6 EVEX Encoding of Broadcast/Rounding/SAE Support ....................................................... 2-42
2.7.7 Embedded Broadcast Support in EVEX ............................................................................. 2-42
2.7.8 Static Rounding Support in EVEX .................................................................................... 2-42
2.7.9 SAE Support in EVEX ....................................................................................................... 2-42
2.7.10 Vector Length Orthogonality ........................................................................................... 2-42
2.7.11 #UD Equations for EVEX ................................................................................................. 2-43
2.7.11.1 State Dependent #UD ............................................................................................... 2-43
2.7.11.2 Opcode Independent #UD .......................................................................................... 2-43
2.7.11.3 Opcode Dependent #UD ............................................................................................ 2-44
2.7.12 Device Not Available ........................................................................................................ 2-45
2.7.13 Scalar Instructions ............................................................................................................. 2-45
2.8 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS ............... 2-45
2.8.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions ......................................... 2-49
2.8.2 Exceptions Type E2 of EVEX-Encoded Instructions ......................................................... 2-51
2.8.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions ......................................... 2-52
2.8.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions ......................................... 2-54
2.8.5 Exceptions Type E5 and E5NF ......................................................................................... 2-56
2.8.6 Exceptions Type E6 and E6NF .......................................................................................... 2-58
2.8.7 Exceptions Type E7NM ..................................................................................................... 2-60
2.8.8 Exceptions Type E9 and E9NF ........................................................................................... 2-61
2.8.9 Exceptions Type E10 and E10NF ...................................................................................... 2-63
2.8.10 Exceptions Type E11 (EVEX-only, Mem Arg, No AC, Floating-point Exceptions) ........ 2-65
2.8.11 Exceptions Type E12 and E12NP (VSIB Mem Arg, No AC, No Floating-point Exceptions) . 2-66
2.9 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS, TYPE K20 AND TYPE K21 .. 2-68
2.9.1 Exceptions Type K20 .......................................................................................................... 2-68
2.9.2 Exceptions Type K21 ......................................................................................................... 2-69
2.10 INTEL® AMX INSTRUCTION EXCEPTION CLASSES ....................................................... 2-70

CHAPTER 3
INSTRUCTION SET REFERENCE, A-L

3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES .................................................... 3-1
3.1.1 Instruction Format .............................................................................................................. 3-1
3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix) ... 3-2
3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix) ....... 3-3
3.1.1.3 Instruction Column in the Opcode Summary Table ....................................................... 3-5
3.1.1.4 Operand Encoding Column in the Instruction Summary Table .................................... 3-8
3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table ............................................ 3-8
3.1.1.6 CPUID Support Column in the Instruction Summary Table ........................................... 3-9
3.1.1.7 Description Column in the Instruction Summary Table ................................................. 3-9
3.1.1.8 Description Section ....................................................................................................... 3-9
3.1.1.9 Operation Section .......................................................................................................... 3-9
3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section .................................................... 3-12
3.1.1.11 Flags Affected Section ................................................................................................. 3-14
3.1.1.12 FPU Flags Affected Section .......................................................................................... 3-14
3.1.1.13 Protected Mode Exceptions Section ............................................................................ 3-14
3.1.1.14 Real-Address Mode Exceptions Section ....................................................................... 3-15
3.1.1.15 Virtual-8086 Mode Exceptions Section ........................................................................ 3-15
3.1.1.16 Floating-Point Exceptions Section .............................................................................. 3-16
3.2 INTEL® AMX CONSIDERATIONS

3.1.1.18 Compatibility Mode Exceptions Section

3.1.1.17 SIMD Floating-Point Exceptions Section

BNDMOV—Move Bounds

BNDMK—Make Bounds

BNDLDX—Load Extended Bounds Using Address Translation

BNDCL—Check Lower Bound

BLSMSK—Get Mask Up to Lowest Set Bit

BLSI—Extract Lowest Set Isolated Bit

BLENDVPS—Variable Blend Packed Single Precision Floating-Point Values

BLENDPS—Blend Packed Single Precision Floating-Point Values

ANDNPS—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values

ANDPD—Add Packed Double Precision Floating-Point Values

ADDPS—Add Packed Single Precision Floating-Point Values

ADDSS—Add Scalar Single Precision Floating-Point Values

ADDUSBPd—Packed Double Precision Floating-Point Add/Subtract

ADDUSBPSP—Packed Single Precision Floating-Point Add/Subtract

ADOX—Unsigned Integer Addition of Two Operands With Overflow Flag

AESDEC—Perform One Round of an AES Decryption Flow

AESDEC128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker Using 128-Bit Key

AESDEC256KL—Perform Fourteen Rounds of AES Decryption Flow With Key Locker Using 256-Bit Key

AESDECLAST—Perform Last Round of an AES Decryption Flow

AESDECLIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

AESDECLIDE256KL—Perform Fourteen Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

AESENC—Perform One Round of an AES Encryption Flow

AESENC128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker Using 128-Bit Key

AESENC256KL—Perform Fourteen Rounds of AES Encryption Flow With Key Locker Using 256-Bit Key

AESENCLAST—Perform Last Round of an AES Encryption Flow

AESDECLIDE128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

AESDECLIDE256KL—Perform Fourteen Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

AESIMC—Perform the AES InvMixColumn Transformation

AESKEYGENASSIST—AES Round Key Generation Assist

AND—Logical AND

ANDN—Logical AND NOT

ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values

ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values

ARPL—Adjust RPL Field of Segment Selector

BEXTR—Bit Field Extract

BLENDVPS—Variable Blend Packed Double Precision Floating-Point Values

BLENDPS—Blend Packed Single Precision Floating-Point Values

BLSI—Extract Lowest Set Isolated Bit

BLSMSK—Get Mask Up to Lowest Set Bit

BLSR—Reset Lowest Set Bit

BNDCHECK—Check Lower Bound

BNDCU/BNDNC—Check Upper Bound

BNLDX—Load Extended Bounds Using Address Translation

BNDMK—Make Bounds

BNDMOV—Move Bounds

BNDSTX—Store Extended Bounds Using Address Translation

BOUND—Check Array Index Against Bounds

BSF—Bit Scan Forward

BSR—Bit Scan Reverse

CONTENTS
CONTENTS

BSWAP—Byte Swap .......................................................... 3-129
BT—Bit Test ..................................................................... 3-130
BTC—Bit Test and Complement ........................................ 3-132
BTR—Bit Test and Reset ................................................... 3-134
BTS—Bit Test and Set ....................................................... 3-136
BZH—Zero High Bits Starting with Specified Bit Position ... 3-138
CALL—Call Procedure ..................................................... 3-139
CBW/CWD/CDQ—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword .......... 3-156
CLAC—Clear AC Flag in EFLAGS Register ....................... 3-157
CLC—Clear Carry Flag ..................................................... 3-158
CLD—Clear Direction Flag ............................................... 3-159
CLEMOTE—Cache Line Demote ......................................... 3-160
CLFLUSH—Flush Cache Line ............................................. 3-162
CLFLUSHOPT—Flush Cache Line Optimized ..................... 3-164
CL—Clear Interrupt Flag ................................................... 3-166
CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token ................................................................. 3-168
CLTS—Clear Task-Switched Flag in CR0 .............................. 3-170
CLUI—Clear User Interrupt Flag ........................................ 3-171
CLWB—Cache Line Write Back ........................................ 3-172
CMC—Complement Carry Flag .......................................... 3-174
CMOVcc—Conditional Move ............................................. 3-175
CMP—Compare Two Operands ......................................... 3-179
CMPPD—Compare Packed Double Precision Floating-Point Values .......................................................... 3-181
CMPS—Compare Packed Single Precision Floating-Point Values .......................................................... 3-188
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands .......................................................... 3-195
CMPSD—Compare Scalar Double Precision Floating-Point Value ......................................................... 3-199
CMPSQ—Compare Scalar Single Precision Floating-Point Value .......................................................... 3-203
CMPPD—Compare Packed Double Precision Floating-Point Values .......................................................... 3-208
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands .......................................................... 3-213
CMOVcc—Conditional Move ............................................. 3-215
CPUID—CPU Identification ............................................... 3-217
CRC32—Accumulate CRC32 Value ..................................... 3-263
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values ................. 3-266
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values .................. 3-270
CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers ................. 3-273
CVTPD2PD—Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values .......................................................... 3-277
CVTPD2PS—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers ................. 3-278
CVTPD2PS—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values .................. 3-282
CVTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers .................. 3-283
CVTPS2PS—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values .......................................................... 3-284
CVTPS2PS—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers .................. 3-287
CVTPS2PS—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers .................. 3-290
CVTS2PS—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer .................................. 3-291
CVTS/SD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value .......................................................... 3-293
CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value .................................. 3-295
CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value .................................. 3-297
CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value .......................................................... 3-299
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Integer .................................................. 3-301
CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers .......................................................... 3-303
CVTTPD2PD—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers .......................................................... 3-307
CVTTPS2DQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers .......................................................... 3-311
CVTSSD2SI—Convert with Truncation Scalar Double Precision Floating-Point Value to Signed Integer .................. 3-312
CVTSSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Integer ............................ 3-314
FPTAN—Partial Tangent

FMUL/FMULP/FIMUL—Multiply

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

FISTTP—Store Integer With Truncation

FINCSTP—Increment Stack-Top Pointer

FILD—Load Integer

FFREE—Free Floating-Point Register

FCOS—Cosine

FCOMI/FCOMIP/FCOMIP—Compare Floating-Point Values and Set EFLAGS

FCOM/FCOMP/FCOMPP—Compare Floating-Point Values

FCMOVcc—Floating-Point Conditional Move

FCHS—Change Sign

FBSTP—Store BCD Integer and Pop

EXTRACTPS—Extract Packed Floating-Point Values

ENQCMDS—Enqueue Command Supervisor

EMS—Empty HMX Technology State

ENCODKEY128—Encode 128-Bit Key With Key Locker

ENCODKEY256—Encode 256-Bit Key With Key Locker

ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode

ENDBR64—Terminate an Indirect Branch in 64-bit Mode

ENQ—Enqueue Command

DPPS—Dot Product of Packed Single Precision Floating-Point Values

DIVSS—Divide Scalar Single Precision Floating-Point Values

DIVSD—Divide Scalar Double Precision Floating-Point Value

DIVPS—Divide Packed Single Precision Floating-Point Values

DIVPD—Divide Packed Double Precision Floating-Point Values

DEC—Decrement by 1

DAS—Decimal Adjust AL After Addition

DAA—Decimal Adjust AL After Addition

CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

CWD—Convert Word To Doubleword

DIV—Unsigned Divide

DIVP—Divide Packed Double Precision Floating-Point Values

DIVPS—Divide Packed Single Precision Floating-Point Values

DIVSD—Divide Scalar Double Precision Floating-Point Value

DIVSS—Divide Scalar Single Precision Floating-Point Values

DPP—Dot Product of Packed Double Precision Floating-Point Values

DPPS—Dot Product of Packed Single Precision Floating-Point Values

FST/FSTP—Store Floating-Point Value

FSINCOS—Sine and Cosine

FSIN—Sine

FSCALE—Scale

FRSTOR—Restore x87 FPU State

FST/FISTP—Store Integer
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSTCW/FNSTCW</td>
<td>Store x87 FPU Control Word</td>
<td>3-439</td>
</tr>
<tr>
<td>FSTENV/FNSTENV</td>
<td>Store x87 FPU Environment</td>
<td>3-441</td>
</tr>
<tr>
<td>FSTSwh/FNSTSw</td>
<td>Store x87 FPU Status Word</td>
<td>3-443</td>
</tr>
<tr>
<td>FSUB/FSUBP/FISUB</td>
<td>Subtract</td>
<td>3-445</td>
</tr>
<tr>
<td>FSUBR/FSUBRP/FISUBR</td>
<td>Reverse Subtract</td>
<td>3-448</td>
</tr>
<tr>
<td>FSTT—TEST</td>
<td>Examine Floating-Point</td>
<td>3-451</td>
</tr>
<tr>
<td>FUCOM/FUCOMP/FUCOMPP</td>
<td>Unordered Compare Floating-Point Values</td>
<td>3-453</td>
</tr>
<tr>
<td>FXAM—Exchange</td>
<td>Exchange Register Contents</td>
<td>3-455</td>
</tr>
<tr>
<td>FXCH—Exchange</td>
<td>Exchange Register Contents</td>
<td>3-457</td>
</tr>
<tr>
<td>FXRSTOR—Restore</td>
<td>x87 FPU, MMX, XMM, and MXCSR State</td>
<td>3-459</td>
</tr>
<tr>
<td>FXSAVE—Save</td>
<td>x87 FPU, MMX Technology, and SSE State</td>
<td>3-462</td>
</tr>
<tr>
<td>FXTRACT—Extract</td>
<td>Extract Exponent and Significand</td>
<td>3-470</td>
</tr>
<tr>
<td>FLY2X—Compute</td>
<td>y * log2x</td>
<td>3-472</td>
</tr>
<tr>
<td>FLY2XP1—Compute</td>
<td>y * log2(x + 1)</td>
<td>3-474</td>
</tr>
<tr>
<td>GF2P8AFFINEINVQB</td>
<td>Galois Field Affine Transformation Inverse</td>
<td>3-476</td>
</tr>
<tr>
<td>GF2P8AFFINEQQB</td>
<td>Galois Field Affine Transformation</td>
<td>3-479</td>
</tr>
<tr>
<td>GF2P8MULB</td>
<td>Galois Field Multiply Bytes</td>
<td>3-481</td>
</tr>
<tr>
<td>HADDPP—Packed Double Precision Floating-Point Horizontal Add</td>
<td>3-483</td>
<td></td>
</tr>
<tr>
<td>HADDPS—Packed Single Precision Floating-Point Horizontal Add</td>
<td>3-486</td>
<td></td>
</tr>
<tr>
<td>HLT—Halt</td>
<td></td>
<td>3-489</td>
</tr>
<tr>
<td>HRESET—History</td>
<td>Reset</td>
<td>3-490</td>
</tr>
<tr>
<td>HSUBPD—Packed</td>
<td>Double Precision Floating-Point Horizontal Subtract</td>
<td>3-492</td>
</tr>
<tr>
<td>HSUBPS—Packed</td>
<td>Single Precision Floating-Point Horizontal Subtract</td>
<td>3-495</td>
</tr>
<tr>
<td>IDIV—Signed Divide</td>
<td></td>
<td>3-498</td>
</tr>
<tr>
<td>IMUL—Signed Multiply</td>
<td></td>
<td>3-501</td>
</tr>
<tr>
<td>IN—Input From Port</td>
<td></td>
<td>3-505</td>
</tr>
<tr>
<td>INC—Increment by 1</td>
<td></td>
<td>3-507</td>
</tr>
<tr>
<td>INCSSPD/INCSSPQ—Increment Shadow Stack Pointer</td>
<td></td>
<td>3-509</td>
</tr>
<tr>
<td>INS/INSW/INSD—Input from Port to String</td>
<td></td>
<td>3-511</td>
</tr>
<tr>
<td>INSERTPS—Insert Scalar Single Precision Floating-Point Value</td>
<td></td>
<td>3-514</td>
</tr>
<tr>
<td>INT n/INTO/INT3/INT1—Call to Interrupt Procedure</td>
<td></td>
<td>3-517</td>
</tr>
<tr>
<td>INVD—Invalid Internal Caches</td>
<td></td>
<td>3-532</td>
</tr>
<tr>
<td>INVLPG—Invalid TLB Entries</td>
<td></td>
<td>3-534</td>
</tr>
<tr>
<td>INVP CID—Invalid Process-Context Identifier</td>
<td></td>
<td>3-536</td>
</tr>
<tr>
<td>IRET/IRETD/IRETQ—Interrupt Return</td>
<td></td>
<td>3-539</td>
</tr>
<tr>
<td>Jcc—Jump if Condition Is Met</td>
<td></td>
<td>3-548</td>
</tr>
<tr>
<td>JMP—Jump</td>
<td></td>
<td>3-553</td>
</tr>
<tr>
<td>KADDw/KADDDB/KADDQ/KADD—ADD Two Masks</td>
<td></td>
<td>3-562</td>
</tr>
<tr>
<td>KANDw/KANDB/KANDQ/KAND—Bitwise Logical AND Masks</td>
<td></td>
<td>3-564</td>
</tr>
<tr>
<td>KANDw/KANDB/KANDQ/KAND—Bitwise Logical AND NOT Masks</td>
<td></td>
<td>3-565</td>
</tr>
<tr>
<td>KMOVW/KKMOVW/KKMOVQ/KKMOV—Move From and to Mask Registers</td>
<td></td>
<td>3-566</td>
</tr>
<tr>
<td>KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register</td>
<td></td>
<td>3-568</td>
</tr>
<tr>
<td>KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks</td>
<td></td>
<td>3-569</td>
</tr>
<tr>
<td>KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks and Set Flags.</td>
<td></td>
<td>3-570</td>
</tr>
<tr>
<td>KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers</td>
<td></td>
<td>3-572</td>
</tr>
<tr>
<td>KSHIFTRw/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers</td>
<td></td>
<td>3-574</td>
</tr>
<tr>
<td>KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags.</td>
<td></td>
<td>3-576</td>
</tr>
<tr>
<td>KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers</td>
<td></td>
<td>3-578</td>
</tr>
<tr>
<td>KXNORw/KXNORB/KXNORQ/KXNORD—Bitwise Logical XOR Masks</td>
<td></td>
<td>3-579</td>
</tr>
<tr>
<td>KXORw/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks</td>
<td></td>
<td>3-580</td>
</tr>
<tr>
<td>LAHF—Load Access Rights Byte</td>
<td></td>
<td>3-581</td>
</tr>
<tr>
<td>LADDQ—Load Unaligned Integer 128 Bits</td>
<td></td>
<td>3-582</td>
</tr>
<tr>
<td>LDMXCSR—Load MXCSR Register</td>
<td></td>
<td>3-585</td>
</tr>
<tr>
<td>LDS/LESLFS/LGS/LSS—Load Far Pointer</td>
<td></td>
<td>3-587</td>
</tr>
<tr>
<td>LDTILECFG—Load Tile Configuration</td>
<td></td>
<td>3-588</td>
</tr>
<tr>
<td>LEA—Load Effective Address</td>
<td></td>
<td>3-592</td>
</tr>
<tr>
<td>LEAVE—High Level Procedure Exit</td>
<td></td>
<td>3-595</td>
</tr>
<tr>
<td>LFENCE—Load Fence</td>
<td></td>
<td>3-597</td>
</tr>
</tbody>
</table>
CHAPTER 4
INSTRUCTION SET REFERENCE, M-U

4.1  IM8 CONTROL BYTE OPERATION FOR PCMPSTRI / PCMPSTRM / PCMPSTRI / PCMPISTRM  4-1
  4.1.1 General Description  4-1
  4.1.2 Source Data Format  4-2
  4.1.3 Aggregation Operation  4-2
  4.1.4 Polarity  4-3
  4.1.5 Output Selection  4-4
  4.1.6 Valid/Invalid Override of Comparisons  4-4
  4.1.7 Summary of Im8 Control byte  4-5
  4.1.8 Diagram Comparison and Aggregation Process  4-6

4.2  COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX  4-6

4.3  INSTRUCTIONS (M-U)  4-7
  4.3.1 MASKMOVDQU—Store Selected Bytes of Double Quadword  4-8
  4.3.2 MASKMOVDQ—Store Selected Bytes of Quadword  4-10
  4.3.3 MAXPD—Maximum of Packed Double Precision Floating-Point Values  4-12
  4.3.4 MAXPS—Maximum of Packed Single Precision Floating-Point Values  4-15
  4.3.5 MAXSD—Return Maximum Scalar Double Precision Floating-Point Value  4-18
  4.3.6 MAXSS—Return Maximum Scalar Single Precision Floating-Point Value  4-20
  4.3.7 MFENCE—Memory Fence  4-22
  4.3.8 MINPD—Minimum of Packed Double Precision Floating-Point Values  4-23
  4.3.9 MINPS—Minimum of Packed Single Precision Floating-Point Values  4-26
  4.3.10 MINS—Return Minimum Scalar Double Precision Floating-Point Value  4-29
  4.3.11 MINSS—Return Minimum Scalar Single Precision Floating-Point Value  4-31
  4.3.12 MONITOR—Set Up Monitor Address  4-33
  4.3.13 MOV—Move  4-35
  4.3.14 MOV—to/from Control Registers  4-39
  4.3.15 MOV—to/from Debug Registers  4-42
  4.3.16 MOVAPD—Move Aligned Packed Double Precision Floating-Point Values  4-44
  4.3.17 MOVAPS—Move Aligned Packed Single Precision Floating-Point Values  4-48
  4.3.18 MOVBE—Move Data After Swapping Bytes  4-52
  4.3.19 MOVD/MOVOQ—Move Doubleword/Move Quadword  4-55
  4.3.20 MOVDUP—Replicate Double Precision Floating-Point Values  4-58
  4.3.21 MOVDIR—Move Doubleword as Direct Store  4-61
  4.3.22 MOVDIR64B—Move 64 Bytes as Direct Store  4-63
  4.3.23 MOVDQ/VMOVQ/8/16/32/64—Move Unaligned Packed Integer Values  4-65
  4.3.24 MOVQDQ/MOVVLQ/8/16/32/64—Move Quadword from XMM to MMX Technology Register  4-70
  4.3.25 MOVLDQU—Move Packed Single Precision Floating-Point Values High to Low  4-77
  4.3.26 MOVB—Move High Packed Double Precision Floating-Point Value  4-78
  4.3.27 MOVKPS—Move High Packed Single Precision Floating-Point Values  4-80
  4.3.28 MOVHPQ—Move High Packed Double Precision Floating-Point Value  4-82
  4.3.29 MOVLPD—Move Low Packed Double Precision Floating-Point Value  4-84
  4.3.30 MOVLPQ—Move Low Packed Single Precision Floating-Point Values  4-86
  4.3.31 MOVSQPD—Extract Packed Double Precision Floating-Point Sign Mask  4-88
  4.3.32 MOVPS—Extract Packed Single Precision Floating-Point Sign Mask  4-90
  4.3.33 MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint  4-92
  4.3.34 MOVNTDQ—Store Packed Integers Using Non-Temporal Hint  4-94
  4.3.35 MOVNTI—Store Doubleword Using Non-Temporal Hint  4-98
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHADDSW—Packed Horizontal Add and Saturate</td>
<td>4-291</td>
</tr>
<tr>
<td>PHMINPOSSW—Packed Horizontal Word Minimum</td>
<td>4-293</td>
</tr>
<tr>
<td>PHSUBW/PHSUBD—Packed Horizontal Subtract</td>
<td>4-295</td>
</tr>
<tr>
<td>PHSUBSW—Packed Horizontal Subtract and Saturate</td>
<td>4-298</td>
</tr>
<tr>
<td>PINSRB/PINSRD/PINSQ—Insert Byte/Dword/Qword</td>
<td>4-300</td>
</tr>
<tr>
<td>PINSW—Insert Word</td>
<td>4-303</td>
</tr>
<tr>
<td>PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes</td>
<td>4-305</td>
</tr>
<tr>
<td>PMADDWD—Multiply and Add Packed Integers</td>
<td>4-308</td>
</tr>
<tr>
<td>PMAXSB/PMAXSW/PMAXD/PMAXQ—Maximum of Packed Signed Integers</td>
<td>4-311</td>
</tr>
<tr>
<td>PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers</td>
<td>4-318</td>
</tr>
<tr>
<td>PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers</td>
<td>4-323</td>
</tr>
<tr>
<td>PMINSB/PMINSW—Minimum of Packed Signed Integers</td>
<td>4-327</td>
</tr>
<tr>
<td>PMINSD/PMINSQ—Minimum of Packed Signed Integers</td>
<td>4-332</td>
</tr>
<tr>
<td>PMINUB/PMINUW—Minimum of Packed Unsigned Integers</td>
<td>4-336</td>
</tr>
<tr>
<td>PMINUD/PMINUQ—Minimum of Packed Unsigned Integers</td>
<td>4-341</td>
</tr>
<tr>
<td>PMOVMSKB—Move Byte Mask</td>
<td>4-345</td>
</tr>
<tr>
<td>PMOVQ—Packed Move With Sign Extend</td>
<td>4-347</td>
</tr>
<tr>
<td>PMOVZX—Packed Move With Zero Extend</td>
<td>4-356</td>
</tr>
<tr>
<td>PMULDQ—Multiply Packed Doubleword Integers</td>
<td>4-366</td>
</tr>
<tr>
<td>PMULHSW—Packed Multiply High With Round and Scale</td>
<td>4-369</td>
</tr>
<tr>
<td>PMULHWW—Multiply Packed Unsigned Integers and Store High Result</td>
<td>4-373</td>
</tr>
<tr>
<td>PMULHW—Multiply Packed Signed Integers and Store High Result</td>
<td>4-377</td>
</tr>
<tr>
<td>PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result</td>
<td>4-381</td>
</tr>
<tr>
<td>PMULLW—Multiply Packed Signed Integers and Store Low Result</td>
<td>4-385</td>
</tr>
<tr>
<td>PMULDQ—Multiply Packed Unsigned Doubleword Integers</td>
<td>4-389</td>
</tr>
<tr>
<td>POP—Pop a Value From the Stack</td>
<td>4-392</td>
</tr>
<tr>
<td>POPA/POPAD—Pop All General-Purpose Registers</td>
<td>4-397</td>
</tr>
<tr>
<td>POPCNT—Return the Count of Number of Bits Set to 1</td>
<td>4-399</td>
</tr>
<tr>
<td>POPF/POPD/POPFQ—Pop Stack Into EFLAGS Register</td>
<td>4-401</td>
</tr>
<tr>
<td>POR—Bitwise Logical OR</td>
<td>4-405</td>
</tr>
<tr>
<td>PREFETCHh—Prefetch Data Into Caches</td>
<td>4-408</td>
</tr>
<tr>
<td>PREFETCHhW—Prefetch Data Into Caches in Anticipation of a Write</td>
<td>4-410</td>
</tr>
<tr>
<td>PSADBw—Compute Sum of Absolute Differences</td>
<td>4-412</td>
</tr>
<tr>
<td>PSHUF—Packed Shuffle Bytes</td>
<td>4-416</td>
</tr>
<tr>
<td>PSHUFD—Shuffle Packed Doublewords</td>
<td>4-420</td>
</tr>
<tr>
<td>PSHUFW—Shuffle Packed High Words</td>
<td>4-424</td>
</tr>
<tr>
<td>PSHULW—Shuffle Packed Low Words</td>
<td>4-427</td>
</tr>
<tr>
<td>PSHUFW—Shuffle Packed Words</td>
<td>4-430</td>
</tr>
<tr>
<td>PSGN/B/PSGNW/PSGN—Packed SIGN</td>
<td>4-431</td>
</tr>
<tr>
<td>PSLLDQ—Shift Double Quadword Left Logical</td>
<td>4-435</td>
</tr>
<tr>
<td>PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical</td>
<td>4-437</td>
</tr>
<tr>
<td>PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic</td>
<td>4-449</td>
</tr>
<tr>
<td>PSRLDQ—Shift Double Quadword Right Logical</td>
<td>4-459</td>
</tr>
<tr>
<td>PSRLW/PSRDL/PSRLQ—Shift Packed Data Right Logical</td>
<td>4-461</td>
</tr>
<tr>
<td>PSUBB/PSUW/PSUBD—Subtract Packed Integers</td>
<td>4-473</td>
</tr>
<tr>
<td>PSUBQ—Subtract Packed Quadword Integers</td>
<td>4-481</td>
</tr>
<tr>
<td>PSUBSB/PSUBSW—Subtract Packed Signed Integers Wth Signed Saturation</td>
<td>4-484</td>
</tr>
<tr>
<td>PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers Wth Unsigned Saturation</td>
<td>4-488</td>
</tr>
<tr>
<td>PTTEST—Logical Compare</td>
<td>4-492</td>
</tr>
<tr>
<td>PTWRITE—Write Data to a Processor Trace Packet</td>
<td>4-494</td>
</tr>
<tr>
<td>PUNPCKLBW/PUNPCKHDW/PUNPCKHDQ/PUNPCKHDQ—Unpack High Data</td>
<td>4-496</td>
</tr>
<tr>
<td>PUNPCKLW/PUNPCKLDW/PUNPCKLDD/PUNPCKLDQ—Unpack Low Data</td>
<td>4-506</td>
</tr>
<tr>
<td>PUSH—Push Word, Doubleword, or Quadword Onto the Stack</td>
<td>4-516</td>
</tr>
<tr>
<td>PUSHA/PUSHAD—Push All General-Purpose Registers</td>
<td>4-519</td>
</tr>
<tr>
<td>PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register Onto the Stack</td>
<td>4-521</td>
</tr>
<tr>
<td>PXOR—Logical Exclusive OR</td>
<td>4-523</td>
</tr>
<tr>
<td>RCR/RCL/ROR/ROR—Rotate</td>
<td>4-526</td>
</tr>
<tr>
<td>RCPPS—Compute Reciprocals of Packed Single Precision Floating-Point Values</td>
<td>4-531</td>
</tr>
<tr>
<td>RCPSS—Compute Reciprocal of Scalar Single Precision Floating-Point Values</td>
<td>4-533</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDFSBASE/RDGSBASE</td>
<td>Read FS/GS Segment Base</td>
</tr>
<tr>
<td>RDMSR</td>
<td>Read From Model Specific Register</td>
</tr>
<tr>
<td>RDPID</td>
<td>Read Processor ID</td>
</tr>
<tr>
<td>RDPKRU</td>
<td>Read Protection Key Rights for User Pages</td>
</tr>
<tr>
<td>RDPMC</td>
<td>Read Performance-Monitoring Counters</td>
</tr>
<tr>
<td>RDRAND</td>
<td>Read Random Number</td>
</tr>
<tr>
<td>RDSEED</td>
<td>Read Random SEED</td>
</tr>
<tr>
<td>RDSSPD/RDSSPQ</td>
<td>Read Shadow Stack Pointer</td>
</tr>
<tr>
<td>RDTSC</td>
<td>Read Time-Stamp Counter</td>
</tr>
<tr>
<td>RDTSCP</td>
<td>Read Time-Stamp Counter and Processor ID</td>
</tr>
<tr>
<td>REP/REPZ/REPNE/REPNZ</td>
<td>Repeat String Operation Prefix</td>
</tr>
<tr>
<td>RET</td>
<td>Return From Procedure</td>
</tr>
<tr>
<td>RORX</td>
<td>Rotate Right Logical Without Affecting Flags</td>
</tr>
<tr>
<td>ROUNDPD</td>
<td>Round Packed Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>ROUNDPS</td>
<td>Round Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>ROUNDSD</td>
<td>Round Scalar Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>ROUNDSS</td>
<td>Round Scalar Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>RSM</td>
<td>Resume From System Management Mode</td>
</tr>
<tr>
<td>RSRTPS</td>
<td>Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>RSRTSS</td>
<td>Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value</td>
</tr>
<tr>
<td>RSTORSSP</td>
<td>Restore Saved Shadow Stack Pointer</td>
</tr>
<tr>
<td>SAHF</td>
<td>Store AH Into Flags</td>
</tr>
<tr>
<td>SAL/SAR/SHL/SHR</td>
<td>Shift</td>
</tr>
<tr>
<td>SARX/SHLX/SHRX</td>
<td>Shift Without Affecting Flags</td>
</tr>
<tr>
<td>SAVEPREVSSP</td>
<td>Save Previous Shadow Stack Pointer</td>
</tr>
<tr>
<td>SBB</td>
<td>Integer Subtraction With Borrow</td>
</tr>
<tr>
<td>SCAS/SCASB/SCASW/SCASD</td>
<td>Scan String</td>
</tr>
<tr>
<td>SENDUIPI</td>
<td>Send User Interprocessor Interrupt</td>
</tr>
<tr>
<td>SERIALIZE</td>
<td>Serialize Instruction Execution</td>
</tr>
<tr>
<td>SETcc</td>
<td>Set Byte on Condition</td>
</tr>
<tr>
<td>SETSSBSY</td>
<td>Mark Shadow Stack Busy</td>
</tr>
<tr>
<td>SFENCE</td>
<td>Store Fence</td>
</tr>
<tr>
<td>SGDTR</td>
<td>Store Global Descriptor Table Register</td>
</tr>
<tr>
<td>SHA1RND4</td>
<td>Perform Four Rounds of SHA1 Operation</td>
</tr>
<tr>
<td>SHA1NEXTE</td>
<td>Calculate SHA1 State Variable E After Four Rounds</td>
</tr>
<tr>
<td>SHA1MSG1</td>
<td>Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords</td>
</tr>
<tr>
<td>SHA1MSG2</td>
<td>Perform a Final Calculation for the Next Four SHA1 Message Dwords</td>
</tr>
<tr>
<td>SHA256RND2</td>
<td>Perform Two Rounds of SHA256 Operation</td>
</tr>
<tr>
<td>SHA256MSG1</td>
<td>Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords</td>
</tr>
<tr>
<td>SHA256MSG2</td>
<td>Perform a Final Calculation for the Next Four SHA256 Message Dwords</td>
</tr>
<tr>
<td>SHLD</td>
<td>Double Precision Shift Left</td>
</tr>
<tr>
<td>SHRD</td>
<td>Double Precision Shift Right</td>
</tr>
<tr>
<td>SHUFDP</td>
<td>Packed Interleave Shuffle of Pairs of Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>SHUFPDS</td>
<td>Packed Interleave Shuffle of Quadruplets of Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>SIDT</td>
<td>Store Interrupt Descriptor Table Register</td>
</tr>
<tr>
<td>SLDT</td>
<td>Store Local Descriptor Table Register</td>
</tr>
<tr>
<td>SMSW</td>
<td>Store Machine Status Word</td>
</tr>
<tr>
<td>SQRTPD</td>
<td>Square Root of Double Precision Floating-Point Values</td>
</tr>
<tr>
<td>SQRTPS</td>
<td>Square Root of Single Precision Floating-Point Values</td>
</tr>
<tr>
<td>SQRTSD</td>
<td>Compute Square Root of Scalar Double Precision Floating-Point Value</td>
</tr>
<tr>
<td>SQRTSS</td>
<td>Compute Square Root of Scalar Single Precision Value</td>
</tr>
<tr>
<td>STAC</td>
<td>Set AC Flag in EFLAGS Register</td>
</tr>
<tr>
<td>STC</td>
<td>Set Carry Flag</td>
</tr>
<tr>
<td>STD</td>
<td>Set Direction Flag</td>
</tr>
<tr>
<td>STI</td>
<td>Set Interrupt Flag</td>
</tr>
<tr>
<td>STMXCSR</td>
<td>Store MXCSR Register State</td>
</tr>
<tr>
<td>STOS/STOSB/STOSW/STOSD/STOSQ</td>
<td>Store String</td>
</tr>
<tr>
<td>STR</td>
<td>Store Task Register</td>
</tr>
<tr>
<td>STTILECFG</td>
<td>Store Tile Configuration</td>
</tr>
</tbody>
</table>
CHAPTER 5
INSTRUCTION SET REFERENCE, V

5.1 TERINARY BIT VECTOR LOGIC TABLE. .......................................................... 5-1

5.2 INSTRUCTIONS (V) .......................................................... 5-4

VADDPH—Add Packed FP16 Values .......................................................... 5-5
VADDH—Add Scalar FP16 Values .......................................................... 5-7
VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors .......................................................... 5-8
VBLEN2MPP/VBLEN2MPM—Blend Float64/Float32 Vectors Using an OpMask Control .......................................................... 5-11
VBRUNCAST—Load with Broadcast Floating-Point Data .......................................................... 5-13
VCMPH—Compare Packed FP16 Values .......................................................... 5-21
VCMPSH—Compare Scalar FP16 Values .......................................................... 5-23
VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS .......................................................... 5-25
VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory .......................................................... 5-27
VCOMPRESSPS—Store Sparse Packed Single Precision Floating-Point Values Into Dense Memory .......................................................... 5-29
VCVTDOQPH—Convert Packed Signed Doubleword Integers to Packed FP16 Values .......................................................... 5-31
VCVTN2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data .......................................................... 5-33
VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data .......................................................... 5-35
VCVTD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values .......................................................... 5-37
VCVTDOQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers .......................................................... 5-39
VCVTD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers .......................................................... 5-41
VCVTD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers .......................................................... 5-43
VCVTH2DOQ—Convert Packed FP16 Values to Signed Doubleword Integers .......................................................... 5-45
VCVTH2PD—Convert Packed FP16 Values to FP64 Values .......................................................... 5-47
VCVT2PS/VCVTPH2PSX—Convert Packed FP16 Values to Single Precision Floating-Point Values .......................................................... 5-49
VCVTH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values .......................................................... 5-53
VCVTH2UDDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers .......................................................... 5-55
VCVTH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers .......................................................... 5-57
VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers ............................................................... 5-59
VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers ................................................................. 5-61
VCVTPS2PH—Convert Single-Precision FP Value to 16-bit FP Value ............................................................... 5-63
VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values ............... 5-67
VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values ................................................................. 5-69
VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values .................................................................................. 5-71
VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values .......................................................................................... 5-74
VCVTQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values .......... 5-76
VCVTQ2PH—Convert Packed Quadword Integers to Packed FP16 Values ..................................................... 5-78
VCVTQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values .... 5-80
VCVTSD2SH—Convert Low FP64 Value to an FP16 Value ............................................................................. 5-82
VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Doubleword Integer 5-83
VCVTSH2SD—Convert Low FP16 Value to an FP64 Value ............................................................................. 5-85
VCVTSH2SI—Convert Low FP16 Value to Signed Integer .............................................................................. 5-86
VCVTSH2SS—Convert Low FP16 Value to FP32 Value ................................................................................... 5-87
VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer ...................................................................... 5-88
VCVTSH2USI—Convert Low FP16 Value to Signed Integer .............................................................................. 5-88
VCVTSS2SH—Convert Low FP16 Value to an FP16 Value ............................................................................. 5-90
VCVTSS2USI—Convert Scalar Single Precision Floating-Point Value to Unsigned Doubleword Integer 5-92
VCVTTPD2QQ—Convert With Truncation Packed Quadword Integers ............................................................. 5-94
VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers ................................................................. 5-96
VCVTTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers ................................................................. 5-96
VCVTTPH2DDQ—Convert With Truncation Packed FP16 Values to Signed Doubleword Internegers .......... 5-100
VCVTTPH2FDQ—Convert With Truncation Packed FP16 Values to Signed Quadword Integers ............... 5-104
VCVTTPH2UDQ—Convert With Truncation Packed FP16 Values to Unsigned Doubleword Integers .......... 5-106
VCVTTPH2UQQ—Convert With Truncation Packed FP16 Values to Packed Signed Quadword Integers .... 5-108
VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers .................................................. 5-108
VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers ........................................................... 5-110
VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Quadword Integer Values ................................................................. 5-112
VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integers ................................................................. 5-114
VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values ................................................................. 5-114
VCVTTS2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer 5-116
VCVTTS2USI—Convert With Truncation Low FP16 Value to a Signed Integer .............................................. 5-119
VCVTTS2USI—Convert With Truncation Low FP16 Value to an Unsigned Integer ........................................... 5-120
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values ................................................................. 5-122
VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values ....................... 5-124
VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values ................................................................. 5-126
VCVTUQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double Precision Floating-Point Values ................................................................. 5-128
VCVTUQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values ............................... 5-130
VCVTUQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values ................................................................. 5-132
VCVTSI2SH—Convert Unsigned Doubleword Integer to an FP16 Value ...................................................... 5-132
VCVTSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value ....................... 5-134
VCVTSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value .......................... 5-136
VCVTU2PH—Convert Packed Unsiigned Word Integers to FP16 Values ....................................................... 5-140
VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values ............................................................ 5-142
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes ................. 5-144
VDIVPH—Divide Packed FP16 Values ............................................................................................................. 5-147
VDIVSH—Divide Scalar FP16 Values ............................................................................................................. 5-149
VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision ............................ 5-150
VERR/VERw—Verify a Segment for Reading or Writing .............................................................................. 5-152
CONTENTS

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values ................................................................. 5-323
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values ...................................................... 5-326
VPBROADCASTM—Broadcast Mask to Vector Register ........................................................................... 5-426
VPBROADCAST—Load Integer and Broadcast ........................................................................................... 5-323
VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register ................................................................. 5-423
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask .................................................................................. 5-410
VMULPH—Multiply Packed FP16 Values ....................................................................................................... 5-411
VMOVW—Move Word ................................................................................................................................... 5-410
VMOVSH—Move Scalar FP16 Value ............................................................................................................. 5-408
VMINPH—Return Minimum of Packed FP16 Values .................................................................................... 5-404
VPBLENDD—Blend Packed Dwords ............................................................................................................ 5-416
VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register .................. 5-449
VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register ................ 5-442
VPCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register ........................................................................................................... 5-440
VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask ................................................................ 5-437
VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask .............................................................. 5-440
VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask .................................................................... 5-437
VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask .............................................................. 5-440
VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask .............................................................. 5-443
VPCMPUW/VPCMPUW—Compare Packed Word Values Into Mask ............................................................... 5-446
VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Value to Single Precision Floating-Point Value ................................................................. 5-370
VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector ............................................ 5-380
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector ............................................. 5-382
VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values ................................................................. 5-356
VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values .......................................................... 5-359
VGETEXPSH—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value ................................................................. 5-362
VGETEXPS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value ................................................................. 5-368
VGETMANTP—Extract Float64 Vector of Normalized Mantissas From Float64 Vector ..................................................... 5-372
VGETMANTPH—Extract Float64 of Normalized Mantissa From Float64 Scalar ............................................................ 5-383
VGETMANTPD—Extract Float64 of Normalized Mantissa From Float64 Scalar ............................................................ 5-385
VGETMANTPS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar ................................................. 5-387
VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values ................................................................. 5-356
VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values .......................................................... 5-359
VGETEXPSH—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value ................................................................. 5-362
VGETEXPS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value ................................................................. 5-368
VGETMANTP—Extract Float64 Vector of Normalized Mantissas From Float64 Vector ..................................................... 5-372
VGETMANTPH—Extract Float64 of Normalized Mantissa From Float64 Scalar ............................................................ 5-383
VGETMANTPD—Extract Float64 of Normalized Mantissa From Float64 Scalar ............................................................ 5-385
VGETMANTPS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar ................................................. 5-387
VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values ................................................................. 5-356
VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values .......................................................... 5-359
VGETEXPSH—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value ................................................................. 5-362
VGETEXPS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value ................................................................. 5-368
VGETMANTP—Extract Float64 Vector of Normalized Mantissas From Float64 Vector ..................................................... 5-372
VGETMANTPH—Extract Float64 of Normalized Mantissa From Float64 Scalar ............................................................ 5-383
VGETMANTPD—Extract Float64 of Normalized Mantissa From Float64 Scalar ............................................................ 5-385
VGETMANTPS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar ................................................. 5-387
VINCERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values ................................................................. 5-389
VMASKMOV—Conditional SIMD Packed Loads and Stores .............................................................................. 5-393
VMAXPH—Return Maximum of Packed FP16 Values ...................................................................................... 5-400
VMAXSH—Return Maximum of Scalar FP16 Values ..................................................................................... 5-402
VMINPH—Return Minimum of Packed FP16 Values ..................................................................................... 5-404
VMINSH—Return Minimum of Scalar FP16 Value .......................................................................................... 5-406
VMOVSH—Move Scalar FP16 Value ............................................................................................................. 5-408
VMOVW—Move Word ................................................................................................................................... 5-410
VMULPH—Multiply Packed FP16 Values ........................................................................................................ 5-411
VMULSH—Multiply Scalar FP16 Values ........................................................................................................ 5-413
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers ................................................................................. 5-414
VPBLENDD—Blend Packed Dwords ............................................................................................................. 5-416
VPBLENDMB/VPBLENDMw—Blend Byte/Word Vectors Using an Opmask Control ...................................................... 5-418
VPBLENDMB/VPBLENMw—Blend int32/int64 Vectors Using an Opmask Control ...................................................... 5-420
VPBROADCASTBw/D/Q—Load With Broadcast Integer Data From General Purpose Register ......................... 5-423
VPBROADCAST—Load Integer and Broadcast ............................................................................................. 5-426
VPBROADCASTM—Broadcast Mask to Vector Register ..................................................................................... 5-435
VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask ........................................................................ 5-437
VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask ........................................................................ 5-440
VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask ........................................................................ 5-443
VPCMPUW/VPCMPUW—Compare Packed Word Values Into Mask ...................................................................... 5-446
VCOMPRESSB/VCOMPRESSW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register ........................................................................................................... 5-449
VCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register ......................... 5-442
VCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register ......................... 5-454
VPDPBUSD—Multiply and Add Unsigned and Signed Bytes
VPDPBUSDSS—Multiply and Add Unsigned and Signed Bytes With Saturation
VPDPwSSD—Multiply and Add Signed Word Integers
VPDPwSSDS—Multiply and Add Signed Word Integers With Saturation
VPERM2F128—Permute Floating-Point Values
VPERM128—Permute Integer Values
VPERMB—Permute Packed Bytes Elements
VPERMD/VPERMW—Permute Packed Doubleword/Word Elements
VPERM128B—Full Permute of Bytes From Two Tables Overwriting the Index
VPERM128W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index
VPERMLPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values
VPERMLPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values
VPERMPPD—Permute Double Precision Floating-Point Elements
VPERMP5—Permute Single Precision Floating-Point Elements
VPERMQ—Qwords Element Permutation
VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table
VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table
VPEXPSD/VPEXPSDw—Expand Byte/Word Values
VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register
VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register
VGATHERDD/VGATHERQD—Gather Packed Dword Values Using Signed Dword/Qword Indices
VGATHERDD/VGATHERQQ—Gather Packed Dword, Packed Quadword with Signed Qword Indices
VGATHERDQ/VGATHERQQ—Gather Packed Qword Values Using Signed Dword/Qword Indices
VGATHERQD/VGATHERQQ—Gather Packed Qword Values Using Signed Dword/Qword Indices
VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Quadword Values
VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators
VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add the Low 52-Bit Products to Qword Accumulators
VPMASKMOV—Conditional SIMD Integer Packed Loads and Stores
VPMOV8B2M/VPMOV8w2M/VPMOV1D2M/VPMOVQ2M—Convert a Vector Register to a Mask
VPMOVDB/VPMOVUSDB/VPMOVUSw—Down Convert Dword to Byte
VPMOVqW/VPMOVUSqW/VPMOVUSq DB—Down Conver Dword to Word
VPMOVMB/VPMOVMS2w/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register
VPMOVQDB/VPMOVUSQDB—Down Convert Qword to Dword
VPMOVQD/VPMOVUSQD—Down Convert Qword to Quadword
VPMOVQW/VPMOVUSQW—Down Convert Qword to Quadword
VPMOVW/VPMOVUSW/VPMOVWDB—Down Convert Word to Dword
VPMULSHFTQ—Select Packed Unaligned Bytes From Quadword Sources
VPOPQNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD
VPROCOLD/VPROCOLDW/VPROCOLDQ/VPROCOLDQ—Bit Rotate Left
VPSCATTERDD/VPSCATTERDQ/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices
VPSEHCD—Concatenate and Shift Packed Data Left Logical
VPSEHCDV—Concatenate and Variable Shift Packed Data Left Logical
VPSEHRD—Concatenate and Shift Packed Data Right Logical
VPSEHRDV—Concatenate and Variable Shift Packed Data Right Logical
VPSEHUFITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask
VPSEVVLW/VPSLLVLW/VPSLVLw—Variable Bit Shift Left Logical
VPSEVW/VPSRAVw/VPSRAVd/VPSRAVq—Variable Bit Shift Right Arithmetic
VPSEVWLw/VPSRLVLW/VPSRLVlq—Variable Bit Shift Right Logical
VPSEVNLW/VPTERNLOGQ—Bitwise Ternary Logic
VPPTESTMB/VPPTESTMw/VPPTESTMD/VPPTESTMQ—Logical AND and Set Mask
VPPTESTNMw/D/Q—Logical NAND and Set
CONTENTS

VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values ........................................... 5-635
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values ...................................................... 5-638
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value ......................................................... 5-640
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values .................................................... 5-642
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value ......................................................... 5-644
VRCPPH—Compute Reciprocals of Packed FP16 Values .................................................................................. 5-646
VRCPSPH—Compute Reciprocal of Scalar FP16 Value ...................................................................................... 5-648
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values .............................................. 5-649
VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values ................................................. 5-652
VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values .............................................. 5-655
VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value ........................................ 5-657
VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value ................................................... 5-659
VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value ........................................... 5-661
VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits ......................... 5-663
VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits ......................... 5-666
VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits ...................... 5-669
VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits ......................... 5-672
VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits .............................. 5-674
VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits .......................... 5-676
VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values ................. 5-678
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value ......................... 5-680
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values .................. 5-682
VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value ....................... 5-684
VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values ............................................... 5-686
VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value ............................. 5-688
VSACLEFPPD—Scale Packed Float64 Values With Float64 Values ............................................................... 5-689
VSACLEFPD—Scale Packed FP16 Values with FP16 Values ........................................................................... 5-692
VSACLEFPFS—Scale Packed Float32 Values with Float32 Values .............................................................. 5-694
VSACLEFPS—Scale Scalar Float64 Values With Float64 Values .................................................................. 5-697
VSACLEFSPD—Scale Scalar FP16 Values With FP16 Values ....................................................................... 5-699
VSACLEFPS—Scale Scalar Float32 Value With Float32 Value ...................................................................... 5-701
VSATTERDPS/VSCATTERPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices .......................................................................................... 5-703
VSHUFF32x4/VSHUFF64x2/VSHUFF32x4/VSHUFF64x2—Shuffle Packed Values at 128-Bit Granularity .......... 5-707
VSDRT—Compute Square Root of Packed FP16 Values ................................................................................. 5-712
VSDRTS—Compute Square Root of Scalar FP16 Value ................................................................................ 5-714
VSUBPH—Subtract Packed FP16 Values ........................................................................................................ 5-715
VSUBSH—Subtract Scalar FP16 Value ............................................................................................................ 5-717
VTESTPD/VTESTPS—Packed Bit Test ........................................................................................................... 5-718
VCOMISO—Unordered Compare Scalar FP16 Values and Set EFLAGS ....................................................... 5-721
VZEROALL—Zero XMM, YMM, and ZMM Registers ..................................................................................... 5-722
VZEROUPPER—Zero Upper Bits of YMM and ZMM Registers ....................................................................... 5-723

CHAPTER 6
INSTRUCTION SET REFERENCE, W-Z

6.1 INSTRUCTIONS (W-Z) ........................................................................................................... 6-1
WAIT/FWAIT—Wait ......................................................................................................................... 6-1
WBINVD—Write Back and Invalidate Cache ......................................................................................... 6-3
WBNPOINVD—Write Back and Do Not Invalidate Cache .................................................................... 6-5
WFSDBASE/WGRSBASE—Write FS/GS Segment Base ................................................................. 6-7
WRMSR—Write to Model Specific Register ....................................................................................... 6-9
WRPKRU—Write Data to User Page Key Register .............................................................................. 6-11
WRSSD/WRSSQ—Write to Shadow Stack ........................................................................................... 6-13
WRUSD/WRUSSQ—Write to User Shadow Stack ................................................................................ 6-15
XABORT—Transactional Abort ......................................................................................................... 6-17
XACQUIRE/XRELEASE—Hardware Lock Elision Prefix Hints ...................................................... 6-19
XADD—Exchange and Add .............................................................................................................. 6-23
XBEGINT—Transactional Begin ...................................................................................................... 6-25
XCHG—Exchange Register/Memory With Register ........................................................................... 6-28
CHAPTER 7
SAFER MODE EXTENSIONS REFERENCE

7.1 OVERVIEW ................................................................. 7-1
7.2 SMX FUNCTIONALITY .................................................... 7-1
7.2.1 Detecting and Enabling SMX ...................................... 7-1
7.2.2 SMX Instruction Summary ........................................... 7-2
7.2.2.1 GETSEC[CAPABILITIES] ....................................... 7-3
7.2.2.2 GETSEC[ENTERACCS] ........................................... 7-3
7.2.2.3 GETSEC[EXITAC] ................................................. 7-3
7.2.2.4 GETSEC[SENTER] ................................................. 7-4
7.2.2.5 GETSEC[SEXIT] .................................................. 7-4
7.2.2.6 GETSEC[PARAMETERS] ......................................... 7-4
7.2.2.7 GETSEC[PARAMETERS] ......................................... 7-4
7.2.2.8 GETSEC[WAKEUP] ............................................... 7-4
7.2.3 Measured Environment and SMX ................................ 7-5
7.3 GETSEC LEAF FUNCTIONS ............................................. 7-5
GETSEC[CAPABILITIES]—Report the SMX Capabilities .......... 7-7
GETSEC[ENTERACCS]—Execute Authenticated Chipset Code .... 7-10
GETSEC[EXITAC]—Exit Authenticated Code Execution Mode .... 7-18
GETSEC[SENTER]—Enter a Measured Environment .............. 7-21
GETSEC[SEXIT]—Exit Measured Environment ..................... 7-30
GETSEC[PARAMETERS]—Report the SMX Parameters ........... 7-33
GETSEC[SMCTRL]—SMX Mode Control ............................... 7-37
GETSEC[WAKEUP]—Wake Up Sleeping Processors in Measured Environment .......................... 7-40

CHAPTER 8
INSTRUCTION SET REFERENCE UNIQUE TO INTEL® XEON PHI™ PROCESSORS

PREFETCHWT1—Prefetch Vector Data Into Caches With Intent to Write and T1 Hint ..................... 8-2
V4FMADDPVS/V4FNMADDPVS—Packed Single Precision Floating-Point Fused Multiply-Add (4-Iterations) 8-4
V4FMADDS/V4FNMADDS—Scalar Single Precision Floating-Point Fused Multiply-Add (4-Iterations) ...... 8-6
VEXP2PD—Approximation to the Exponential 2^x of Packed Double Precision Floating-Point Values With Less Than 2^-23 Relative Error .......................................................... 8-8
VEXP2PS—Approximation to the Exponential 2^x of Packed Single Precision Floating-Point Values With Less Than 2^-23 Relative Error .......................................................... 8-10
VGATHERFVQPS/VGATHERFQPD/VGATHERFQOPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T0 Hint ............................................. 8-12
VGATHERP1QPS/VGATHERF1QDP/VGATHERF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint ............................................. 8-14
VP4DPWSDD—Dot Product of Signed Words With Dword Accumulation and Saturation (4-Iterations) ...... 8-16
VP4DPWSDD—Dot Product of Signed Words With Dword Accumulation (4-Iterations) ..................... 8-18
VRCP28PD—Approximation to the Reciprocal of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error .......................................................... 8-20
VRCP28SD—Approximation to the Reciprocal of Scalar Double Precision Floating-Point Value With Less Than
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODE</td>
<td></td>
</tr>
<tr>
<td>B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode</td>
<td></td>
</tr>
<tr>
<td>B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS</td>
<td></td>
</tr>
<tr>
<td>B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS</td>
<td></td>
</tr>
<tr>
<td>B.5 MMX INSTRUCTION FORMATS AND ENCODINGS</td>
<td></td>
</tr>
<tr>
<td>B.5.1 Granularity Field (gg)</td>
<td></td>
</tr>
<tr>
<td>B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg)</td>
<td></td>
</tr>
<tr>
<td>B.5.3 MMX Instruction Formats and Encodings Table</td>
<td></td>
</tr>
<tr>
<td>B.6 PROCESSOR EXTENDED STATE INSTRUCTION FORMATS AND ENCODINGS</td>
<td></td>
</tr>
<tr>
<td>B.7 SSE INSTRUCTION FORMATS AND ENCODINGS</td>
<td></td>
</tr>
<tr>
<td>B.8 SSE2 INSTRUCTION FORMATS AND ENCODINGS</td>
<td></td>
</tr>
<tr>
<td>B.9 SSE3 FORMATS AND ENCODING TABLE</td>
<td></td>
</tr>
<tr>
<td>B.10 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS</td>
<td></td>
</tr>
<tr>
<td>B.11 SPECIAL ENCODINGS FOR 64-BIT MODE</td>
<td></td>
</tr>
<tr>
<td>B.12 Floating-point Instruction Formats and Encodings</td>
<td></td>
</tr>
<tr>
<td>C.1 SIMPLE INTRINSICS</td>
<td></td>
</tr>
<tr>
<td>C.2 COMPOSITE INTRINSICS</td>
<td></td>
</tr>
</tbody>
</table>
CONTENTS

FIGURES

Figure 4-10. 256-bit VPHADDD Instruction Operation ............................................ 4-288
Figure 4-6. Operation of the PACKSSDW Instruction Using 64-Bit Operands ................ 4-189
Figure 4-5. 256-bit VMPSADBW Operation .......................................................... 4-139
Figure 4-3. MOVSHDUP Operation ................................................................. 4-117
Figure 4-1. Operation of PCMPSTRx and PCMPESTRx ....................................... 4-6
Figure 3-25. INVPCID Descriptor ................................................................. 3-536
Figure 3-24. VHSUBPS Operation .................................................................... 3-496
Figure 3-23. HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract ....... 3-492
Figure 3-22. VHSUBPD operation ..................................................................... 3-493
Figure 3-21. HSUBPD—Packed Double Precision Floating-Point Subtract ............... 3-492
Figure 3-20. VHADDPS Operation ..................................................................... 3-487
Figure 3-19. HADDPD—Packed Double Precision Floating-Point Horizontal Add ......... 3-484
Figure 3-18. VHADDPD Operation ..................................................................... 3-483
Figure 3-17. HADDPD—Packed Double Precision Floating-Point Horizontal Add ......... 3-483
Figure 3-16. 64-Byte Data Written to Enqueue Registers ................................... 3-348
Figure 3-15. VCVTTPD2DQ (VEX.256 encoded version) ...................................... 3-304
Figure 3-14. CVTPS2PD (VEX.256 encoded version) ............................................. 3-288
Figure 3-13. VCVTPD2Q (VEX.256 encoded version) ........................................... 3-279
Figure 3-12. VCVTPD2Q (VEX.256 encoded version). ............................................ 3-274
Figure 3-11. CVDQ2PD (VEX.256 encoded version). ............................................. 3-267
Figure 3-10. Algorithm for Extracting Processor Frequency .................................. 3-256
Figure 3-9. Determination of Support for the Processor Brand String ..................... 3-255
Figure 3-8. Feature Information Returned in the EDX Register ............................... 3-245
Figure 3-7. Feature Information Returned in the ECX Register ............................... 3-243
Figure 3-6. Version Information Returned by CPUID in EAX ............................... 3-241
Figure 3-5. Memory Layout of BNDMOV to/from Memory .................................... 3-117
Figure 3-4. ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract ........ 3-45
Figure 3-3. ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract ........ 3-45
Figure 2-10. Intel® AVX-512 Instruction Format and the EVEX Prefix ..................... 2-37
Figure 2-9. VEX bit fields ................................................................................. 2-15
Figure 2-8. Instruction Encoding Format with VEX Prefix ..................................... 2-13
Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used ......... 2-10
Figure 2-6. Memory Addressing With a SIB Byte ............................................... 2-9
Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used ........ 2-9
Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used .................. 2-9
Figure 2-3. Prefix Ordering in 64-bit Mode ....................................................... 2-8
Figure 2-2. Table Interpretation of ModR/M Byte (CBH) ........................................ 2-4
Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format .............................. 1-1
Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation ........................... 1-8
Figure 1-1. Bit and Byte Order .......................................................................... 1-6

Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction ............ 4-573
Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation .................................... 4-508
Figure 4-22. VPUNPCKLBW Instruction Operation Using 64-bit Operands ............... 4-508
Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation .................................... 4-498
Figure 4-20. PUNPCKHBDW Instruction Operation Using 64-bit Operands .............. 4-498
Figure 4-19. PSRLW, PSRLL, and PSRLQ Instruction Operation Using 64-bit Operand .... 4-439
Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand ........ 4-451
Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand .... 4-439
Figure 4-16. 256-bit VPUSHUF Instruction Operation .......................................... 4-418
Figure 4-15. PSHUF with 64-Bit Operands ....................................................... 4-418
Figure 4-14. PMULHW and PMULHW Instruction Operation Using 64-bit Operands .... 4-374
Figure 4-13. PMULL Instruction Operation Using 64-bit Operands ......................... 4-386
Figure 4-12. PMADDWD Execution Model Using 64-bit Operands .......................... 4-309
Figure 4-11. PMADDWD Execution Model Using 64-bit Operands .......................... 4-309
Figure 4-10. 256-bit VPADDADD Instruction Operation ...................................... 4-288
Figure 4-9. PEXT Example .............................................................................. 4-277
Figure 4-8. PDEP Example .............................................................................. 4-277
Figure 4-7. 256-bit VPALIGN Instruction Operation .......................................... 4-222
Figure 4-6. Operation of the PACKSSDW Instruction Using 64-Bit Operands ............ 4-189
Figure 4-5. 256-bit VMPSADBW Operation ...................................................... 4-413
Figure 4-4. MOVSDUP Operation ..................................................................... 4-117
Figure 4-3. MOVSHDUP Operation ................................................................... 4-117
Figure 4-2. VMOVDDUP Operation .................................................................. 4-59
Figure 4-1. Operation of PCMPSTRx and PCMPESx ........................................... 4-6

Page 2-8 of Vol. 2A
CONTENTS

TABLES

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte ................................. 2-5
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte ................................. 2-6
Table 2-3. 32-Bit Addressing Forms with the SIB Byte ........................................ 2-7
Table 2-4. REX Prefix Fields [BITS: 0100WRXB] .................................................. 2-9
Table 2-5. Direct Memory Offset Form of MOV .................................................... 2-11
Table 2-6. Special Cases of REX Encodings ....................................................... 2-11
Table 2-7. RIP-Relative Addressing ................................................................. 2-12
Table 2-8. VEX.vvvv to register name mapping ................................................. 2-17
Table 2-9. Instructions with a VEX.vvvv destination ......................................... 2-17
Table 2-10. VEX.m-mmm interpretation ......................................................... 2-18
Table 2-11. VEXL interpretation ................................................................. 2-19
Table 2-12. VEX.pp interpretation ............................................................. 2-19
Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte .............................. 2-21
Table 2-14. Exception Class Description .......................................................... 2-23
Table 2-15. Instructions in each Exception Class ............................................. 2-24
Table 2-16. #UD Exception and VEX.W=1 Encoding ........................................... 2-25
Table 2-17. #UD Exception and VEX.L Field Encoding ..................................... 2-26
Table 2-18. Type 1 Class Exception Conditions ............................................... 2-27
Table 2-19. Type 2 Class Exception Conditions ............................................... 2-28
Table 2-20. Type 3 Class Exception Conditions ............................................... 2-29
Table 2-21. Type 4 Class Exception Conditions ............................................... 2-30
Table 2-22. Type 5 Class Exception Conditions ............................................... 2-31
Table 2-23. Type 6 Class Exception Conditions ............................................... 2-32
Table 2-24. Type 7 Class Exception Conditions ............................................... 2-33
Table 2-25. Type 8 Class Exception Conditions ............................................... 2-33
Table 2-26. Type 11 Class Exception Conditions ............................................. 2-34
Table 2-27. Type 12 Class Exception Conditions ............................................. 2-35
Table 2-28. VEX-Encoded GPR Instructions .................................................... 2-36
Table 2-29. Type 13 Class Exception Conditions ............................................. 2-36
Table 2-30. EVEX Prefix Bit Field Functional Grouping ..................................... 2-38
Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits ....................................................................................................... 2-39
Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode ............................ 2-39
Table 2-33. Opmask Register Specifier Encoding ............................................. 2-40
Table 2-34. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast ................................................................................................... 2-41
Table 2-35. EVEX DISPB*N for Instructions Not Affected by Embedded Broadcast ....................................................................................................... 2-41
Table 2-36. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions ......................................................................................... 2-43
Table 2-37. OS XSAVE Enabling Requirements of Instruction Categories .......... 2-43
Table 2-38. Opcode Independent, State Dependent EVEX Bit Fields .................. 2-43
Table 2-39. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields ........ 2-44
Table 2-40. #UD Conditions of Opmask Related Encoding Field ...................... 2-44
Table 2-41. #UD Conditions Dependent on EVEX.b Context ............................. 2-45
Table 2-42. EVEX-Encoded Instruction Exception Class Summary .................... 2-45
Table 2-43. EVEX Instructions in Each Exception Class ..................................... 2-46
Table 2-44. Type E1 Class Exception Conditions ............................................... 2-49
Table 2-45. Type E1NF Class Exception Conditions ........................................ 2-50
Table 2-46. Type E2 Class Exception Conditions ............................................... 2-51
Table 2-47. Type E3 Class Exception Conditions ............................................... 2-52
Table 2-48. Type E3NF Class Exception Conditions ........................................ 2-53
Table 2-49. Type E4 Class Exception Conditions ............................................... 2-54
Table 2-50. Type E4NF Class Exception Conditions ........................................ 2-55
Table 2-51. Type E5 Class Exception Conditions ............................................... 2-56
Table 2-52. Type E5NF Class Exception Conditions ........................................ 2-57
Table 2-53. Type E6 Class Exception Conditions ............................................... 2-58
Table 2-54. Type E6NF Class Exception Conditions ........................................ 2-59
Table 2-55. Type E7NM Class Exception Conditions .......................................... 2-60
Table 2-56. Type E9 Class Exception Conditions ............................................... 2-61
Table 2-57. Type E9NF Class Exception Conditions ........................................ 2-62
Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions ........................................... 3-182
Table 3-2. Pseudo-Op and CMPPD Implementation ................................................................. 3-183
Table 3-3. Pseudo-Op and VCMPPD Implementation ........................................................... 3-184
Table 3-4. Pseudo-Op and CMPPS Implementation ............................................................... 3-189
Table 3-5. Pseudo-Op and VCMPPS Implementation .......................................................... 3-190
Table 3-6. Pseudo-Op and CMPSD Implementation ............................................................ 3-200
Table 3-7. Pseudo-Op and VCMPSD Implementation ......................................................... 3-200
Table 3-8. Pseudo-Op and CMPPS Implementation ............................................................. 3-204
Table 3-9. Pseudo-Op and VCMPPS Implementation .......................................................... 3-204
Table 3-10. Feature Information Returned by CPUID Instruction ....................................... 3-218
Table 3-11. Processor Type Field .......................................................................................... 3-241
Table 3-12. Results Obtained from F2XM1 ............................................................................ 3-359
Table 3-13. Type E10 Class Exception Conditions ................................................................. 2-63
Table 3-14. Type E1OF Class Exception Conditions ............................................................... 2-64
Table 3-15. Type E11 Class Exception Conditions ................................................................. 2-65
Table 3-16. Type E12 Class Exception Conditions ................................................................. 2-66
Table 3-17. Type E12NP Class Exception Conditions .............................................................. 2-67
Table 3-18. Type K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg) ............................................................... 2-68
Table 3-19. Type K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory) ............................................................... 2-69
Table 3-20. Intel® AMX Exception Classes ............................................................................. 2-70
Table 3-21. Register Codes Associated with +rb, +rw, +rd, +ro .................................................. 3-2
Table 3-22. Range of Bit Positions Specified by Bit Offset Operands ........................................ 3-12
Table 3-23. Standard and Non-Standard Data Types .............................................................. 3-14
Table 3-24. Intel 64 and IA-32 General Exceptions ............................................................... 3-15
Table 3-25. SIMD Floating-Point Exceptions ......................................................................... 3-16
Table 3-26. Standard and Non-Standard Data Types .............................................................. 3-14
Table 3-27. Processor Brand String Returned with Pentium 4 Processor ................................. 3-255
Table 3-28. Type E10NF Class Exception Conditions ............................................................. 2-64
Table 3-29. Intel 64 and IA-32 General Exceptions ............................................................... 3-15
Table 3-30. FPATAN Results .................................................................................................... 3-415
Table 3-31. FPREM Results ..................................................................................................... 3-417
Table 3-32. FCOS Results ........................................................................................................ 3-381
Table 3-33. Feature Information Returned in the ECX Register .............................................. 3-243
Table 3-34. Results Obtained from FABSD ............................................................................ 3-361
Table 3-35. FSIN Results ......................................................................................................... 3-431
Table 3-36. Results Obtained from F2XM1 ............................................................................ 3-359
Table 3-37. FADD/FADDP/FIADD Results ........................................................................... 3-363
Table 3-38. FSTTTP Results ................................................................................................... 3-401
Table 3-39. Results Obtained from FABS ............................................................................. 3-361
Table 3-40. FTST Results ........................................................................................................ 3-451
Table 3-41. FSUBR/FSUBRP/FISUBR Results ....................................................................... 3-449
Table 3-42. FXAM Results ...................................................................................................... 3-455
Table 3-43. Table 2-59. Type E10 Class Exception Conditions ................................................. 2-63
Table 3-44. Table 2-59. Type E10NR Class Exception Conditions ............................................. 2-64
Table 3-45. Table 2-60. Type E11 Class Exception Conditions .................................................. 2-65
Table 3-46. Table 2-61. Type E12 Class Exception Conditions ................................................ 2-66
Table 3-47. Table 2-62. Type E12NP Class Exception Conditions .......................................... 2-67
Table 3-48. Table 2-63. Type K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg) ............................................................... 2-68
Table 3-49. Table 2-64. Type K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory) ............................................................... 2-69
Table 3-50. Table 2-65. Intel® AMX Exception Classes ............................................................. 2-70
Table 3-51. Table 3-1. Register Codes Associated with +rb, +rw, +rd, +ro .................................................. 3-2
Table 3-52. Table 3-2. Range of Bit Positions Specified by Bit Offset Operands .......................... 3-12
Table 3-53. Table 3-3. Standard and Non-Standard Data Types ............................................. 3-14
Table 3-54. Table 3-4. Intel 64 and IA-32 General Exceptions ................................................. 3-15
Table 3-55. Table 3-5. SIMD Floating-Point Exceptions ........................................................... 3-16
Table 3-56. Table 3-6. Standard and Non-Standard Data Types ............................................. 3-14
Table 3-57. Table 3-7. Processor Type Field .............................................................................. 3-241
Table 3-58. Table 3-8. Results Obtained from F2XM1 ............................................................... 3-359
Table 3-59. Table 3-9. Feature Information Returned in the ECX Register .............................. 3-243
Table 3-60. Table 3-10. More on Feature Information Returned in the EDX Register ................ 3-246
Table 3-61. Table 3-11. Encoding of CPUID Leaf 2 Descriptors ............................................. 3-248
Table 3-62. Table 3-12. Processor Brand String Returned with Pentium 4 Processor ................ 3-255
Table 3-63. Table 3-13. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings .............................................................................. 3-257
Table 3-64. Table 3-14. DIV Action ......................................................................................... 3-323
Table 3-65. Table 3-15. FADD/FADDP/FIADD Results ............................................................. 3-363
Table 3-66. Table 3-16. Results Obtained from F2XM1 ............................................................. 3-359
Table 3-67. Table 3-17. Results Obtained from FABSD ........................................................... 3-361
Table 3-68. Table 3-18. FSTTTP Results ................................................................................ 3-401
Table 3-69. Table 3-19. FADD/FADDP/FIADD Results ........................................................... 3-363
Table 3-70. Table 3-20. FCHS Results ..................................................................................... 3-367
Table 3-71. Table 3-21. FCOM/FCOMP/FCOMPP Results .................................................... 3-375
Table 3-72. Table 3-22. FCOM/FCOM/FCOMP/FUCOM/FUCOM/IP Results ....................... 3-378
Table 3-73. Table 3-23. FCOS Results ...................................................................................... 3-381
Table 3-74. Table 3-24. FDIV/FDIVP/FDIV Results ................................................................. 3-385
Table 3-75. Table 3-25. FDIVR/FDIVRP/FDIVR Results ......................................................... 3-388
Table 3-76. Table 3-26. FICOM/FICOMP Results ................................................................. 3-391
Table 3-77. Table 3-27. FIST/FISTP Results .......................................................................... 3-398
Table 3-78. Table 3-28. FISTTTP Results ................................................................................ 3-401
Table 3-79. Table 3-29. FMUL/FMULP/FIMUL Results ............................................................. 3-412
Table 3-80. Table 3-30. FPATAN Results ............................................................................... 3-415
Table 3-81. Table 3-31. FPREM Results ................................................................................ 3-417
Table 3-82. Table 3-32. FPREM1 Results ............................................................................. 3-419
Table 3-83. Table 3-33. FPTAN Results .................................................................................. 3-421
Table 3-84. Table 3-34. FSCALE Results .............................................................................. 3-429
Table 3-85. Table 3-35. FSIN Results ...................................................................................... 3-431
Table 3-86. Table 3-36. FSINCOS Results ............................................................................ 3-433
Table 3-87. Table 3-37. FSQRT Results ............................................................................... 3-435
Table 3-88. Table 3-38. FSUB/FSUBP/FISUB Results ............................................................. 3-446
Table 3-89. Table 3-39. FSUB/FSUBP/FISUB Results ............................................................. 3-449
Table 3-90. Table 3-40. FTST Results ..................................................................................... 3-451
Table 3-91. Table 3-41. FUCOM/FUCOMP/FUCOMPP Results ............................................. 3-453
Table 3-92. Table 3-42. FXAM Results ................................................................................... 3-455
Table 5-14. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS, and MAX, MAX_ABS.

Table 5-13. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2].

Table 5-10. GetMant() Special Float Values Behavior.

Table 5-11. VGETEXPPS/SS Special Cases.

Table 5-5. VGETEXPPD/SD Special Cases.
Table 5-15. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, \(|a| = |b|, a > 0, b < 0|\) ................................. 5-626
Table 5-16. VRCP14PD/VRCP14SD Special Cases ............................................................................ 5-638
Table 5-17. VRCP14PS/VRCP14SS Special Cases ............................................................................ 5-642
Table 5-18. VCPH/VRCPSH Special Cases ....................................................................................... 5-646
Table 5-19. VREDUCEP/SD/PS/SS Special Cases ............................................................................. 5-650
Table 5-20. VREDUCEPH/VREDUCEPSH Special Cases ................................................................. 5-653
Table 5-21. VRNDSCALEP/SD/PS/SS Special Cases ..................................................................... 5-664
Table 5-22. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH .................................................... 5-667
Table 5-23. VRNDSCALEPH/VRNDSCALESH Special Cases ....................................................... 5-667
Table 5-24. VRSQRT14PD Special Cases ....................................................................................... 5-679
Table 5-25. VRSQRT14SD Special Cases ....................................................................................... 5-681
Table 5-26. VRSQRT14PS Special Cases ........................................................................................ 5-683
Table 5-27. VRSQRT14SS Special Cases ....................................................................................... 5-685
Table 5-28. VRSQRTPH/VRSQRTSH Special Cases ...................................................................... 5-686
Table 5-29. VSCALEFPD/SD/PS/SS Special Cases .................................................................... 5-689
Table 5-30. Additional VSCALEFPD/SD Special Cases ............................................................... 5-690
Table 5-31. VSCALEFPH/VSCALEFSPH Special Cases .............................................................. 5-692
Table 5-32. Additional VSCALEFPH/VSCALEFSPH Special Cases .................................... 5-692
Table 5-33. Additional VSCALEFPS/SS Special Cases ............................................................. 5-694
Table 7-1. Layout of IA32_FEATURE_CONTROL ................................................................. 7-2
Table 7-2. GETSEC Leaf Functions ................................................................................................. 7-3
Table 7-3. GETSEC Capability Result Encoding (EBX = 0) ........................................................... 7-7
Table 7-4. Register State Initialization After GETSEC[ENTERACCS] ........................................... 7-12
Table 7-5. IA32_MISC_ENABLE MSR Initialization by ENTERACCS and ENTER ..................... 7-13
Table 7-6. Register State Initialization After GETSEC[SENTER] and GETSEC[WAKEUP] .......... 7-24
Table 7-7. SMX Reporting Parameters Format. ........................................................................... 7-33
Table 7-8. TXT Feature Extensions Flags ...................................................................................... 7-34
Table 7-9. External Memory Types Using Parameter 3 ............................................................... 7-35
Table 7-10. Default Parameter Values .......................................................................................... 7-35
Table 7-11. Supported Actions for GETSEC[SMCTRL(0)] .......................................................... 7-37
Table 7-12. RLP MM JOIN Data Structure .................................................................................... 7-40
Table 8-1. Special Values Behavior. ............................................................................................. 8-9
Table 8-2. Special Values Behavior. ............................................................................................. 8-11
Table 8-3. VRCP28PD Special Cases ........................................................................................... 8-21
Table 8-4. VRCP28SD Special Cases ........................................................................................... 8-23
Table 8-5. VRCP28PS Special Cases ........................................................................................... 8-25
Table 8-6. VRCP28SS Special Cases ........................................................................................... 8-27
Table 8-7. VRSQRT28PD Special Cases ....................................................................................... 8-29
Table 8-8. VRSQRT28SD Special Cases ....................................................................................... 8-31
Table 8-9. VRSQRT28PS Special Cases ....................................................................................... 8-33
Table 8-10. VRSQRT28SS Special Cases ...................................................................................... 8-35
Table A-1. Superscripts Utilized in Opcode Tables ...................................................................... A-6
Table A-2. One-byte Opcode Map: (0OH – F7H) * .................................................................. A-7
Table A-3. Two-byte Opcode Map: 0OH – 77H (First Byte is OFH) * .................................. A-9
Table A-4. Three-byte Opcode Map: 0OH – F7H (First Two Bytes are OF 38H) * .......... A-13
Table A-5. Three-byte Opcode Map: 0OH – F7H (First two bytes are OF 3AH) * ........ A-15
Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * ........ A-18
Table A-7. D8 Opcode Map when ModR/M Byte is Within 00H to BFH * .......................... A-20
Table A-8. D8 Opcode Map when ModR/M Byte is Outside 00H to BFH * ......................... A-21
Table A-9. D9 Opcode Map when ModR/M Byte is Within 00H to BFH * .......................... A-21
Table A-10. D9 Opcode Map when ModR/M Byte is Outside 00H to BFH * ..................... A-22
Table A-11. DA Opcode Map when ModR/M Byte is Within 00H to BFH * ....................... A-22
Table A-12. DA Opcode Map when ModR/M Byte is Outside 00H to BFH * ................... A-23
Table A-13. DB Opcode Map when ModR/M Byte is Within 00H to BFH * ....................... A-23
Table A-14. DB Opcode Map when ModR/M Byte is Outside 00H to BFH * ................... A-24
Table A-15. DC Opcode Map when ModR/M Byte is Within 00H to BFH * .................... A-24
Table A-16. DC Opcode Map when ModR/M Byte is Outside 00H to BFH * .................... A-25
Table A-17. DD Opcode Map when ModR/M Byte is Within 00H to BFH * .................... A-25
Table A-18. DD Opcode Map when ModR/M Byte is Outside 00H to BFH * .................... A-26
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-19</td>
<td>DE Opcode Map When ModR/M Byte is Within 00H to BFH *</td>
<td>A-26</td>
</tr>
<tr>
<td>A-20</td>
<td>DE Opcode Map When ModR/M Byte is Outside 00H to BFH *</td>
<td>A-27</td>
</tr>
<tr>
<td>A-21</td>
<td>DF Opcode Map When ModR/M Byte is Within 00H to BFH *</td>
<td>A-27</td>
</tr>
<tr>
<td>A-22</td>
<td>DF Opcode Map When ModR/M Byte is Outside 00H to BFH *</td>
<td>A-28</td>
</tr>
<tr>
<td>B-1</td>
<td>Special Fields Within Instruction Encodings</td>
<td>B-2</td>
</tr>
<tr>
<td>B-2</td>
<td>Encoding of reg Field When w Field is Not Present in Instruction</td>
<td>B-3</td>
</tr>
<tr>
<td>B-3</td>
<td>Encoding of reg Field When w Field is Present in Instruction</td>
<td>B-3</td>
</tr>
<tr>
<td>B-4</td>
<td>Encoding of reg Field When w Field is Not Present in Instruction</td>
<td>B-4</td>
</tr>
<tr>
<td>B-5</td>
<td>Encoding of reg Field When w Field is Present in Instruction</td>
<td>B-4</td>
</tr>
<tr>
<td>B-6</td>
<td>Encoding of Operand Size (w) Bit</td>
<td>B-4</td>
</tr>
<tr>
<td>B-7</td>
<td>Encoding of Sign-Extend (s) Bit</td>
<td>B-5</td>
</tr>
<tr>
<td>B-8</td>
<td>Encoding of the Segment Register (sreg) Field</td>
<td>B-5</td>
</tr>
<tr>
<td>B-9</td>
<td>Encoding of Special-Purpose Register (eee) Field</td>
<td>B-5</td>
</tr>
<tr>
<td>B-10</td>
<td>Encoding of Conditional Test (tttn) Field</td>
<td>B-6</td>
</tr>
<tr>
<td>B-11</td>
<td>Encoding of Operation Direction (d) Bit</td>
<td>B-6</td>
</tr>
<tr>
<td>B-12</td>
<td>General Purpose Instruction Formats and Encodings for Non-64-Bit Modes</td>
<td>B-7</td>
</tr>
<tr>
<td>B-13</td>
<td>Special Symbols</td>
<td>B-18</td>
</tr>
<tr>
<td>B-14</td>
<td>General Purpose Instruction Formats and Encodings for 64-Bit Mode</td>
<td>B-18</td>
</tr>
<tr>
<td>B-15</td>
<td>Pentium® Processor Family Instruction Formats and Encodings, Non-64-Bit Modes</td>
<td>B-37</td>
</tr>
<tr>
<td>B-16</td>
<td>Pentium® Processor Family Instruction Formats and Encodings, 64-Bit Mode</td>
<td>B-37</td>
</tr>
<tr>
<td>B-17</td>
<td>Encoding of Granularity of Data Field (gg)</td>
<td>B-38</td>
</tr>
<tr>
<td>B-18</td>
<td>MMX Instruction Formats and Encodings</td>
<td>B-38</td>
</tr>
<tr>
<td>B-19</td>
<td>Formats and Encodings of XSAVE/XRSTOR/XGETBV/XSETBV Instructions</td>
<td>B-41</td>
</tr>
<tr>
<td>B-20</td>
<td>Formats and Encodings of P6 Family Instructions</td>
<td>B-41</td>
</tr>
<tr>
<td>B-21</td>
<td>Formats and Encodings of SSE Floating-Point Instructions</td>
<td>B-42</td>
</tr>
<tr>
<td>B-22</td>
<td>Formats and Encodings of SSE Integer Instructions</td>
<td>B-46</td>
</tr>
<tr>
<td>B-23</td>
<td>Encoding of Granularity of Data Field (gg)</td>
<td>B-47</td>
</tr>
<tr>
<td>B-24</td>
<td>Format and Encoding of SSE Cacheability &amp; Memory Ordering Instructions</td>
<td>B-47</td>
</tr>
<tr>
<td>B-25</td>
<td>Formats and Encodings of SSE2 Floating-Point Instructions</td>
<td>B-48</td>
</tr>
<tr>
<td>B-26</td>
<td>Formats and Encodings of SSE2 Integer Instructions</td>
<td>B-52</td>
</tr>
<tr>
<td>B-27</td>
<td>Format and Encoding of SSE2 Cacheability Instructions</td>
<td>B-52</td>
</tr>
<tr>
<td>B-28</td>
<td>Formats and Encodings of SSE3 Floating-Point Instructions</td>
<td>B-56</td>
</tr>
<tr>
<td>B-29</td>
<td>Formats and Encodings for SSE3 Event Management Instructions</td>
<td>B-57</td>
</tr>
<tr>
<td>B-30</td>
<td>Formats and Encodings for SSE3 Integer and Move Instructions</td>
<td>B-57</td>
</tr>
<tr>
<td>B-31</td>
<td>Formats and Encodings for SSSE3 Instructions</td>
<td>B-58</td>
</tr>
<tr>
<td>B-32</td>
<td>Formats and Encodings of AESNI and PCLMULQDQ Instructions</td>
<td>B-61</td>
</tr>
<tr>
<td>B-33</td>
<td>Special Case Instructions Promoted Using REX.W</td>
<td>B-61</td>
</tr>
<tr>
<td>B-34</td>
<td>Encodings of SSE4.1 Instructions</td>
<td>B-64</td>
</tr>
<tr>
<td>B-35</td>
<td>Encodings of SSE4.2 Instructions</td>
<td>B-69</td>
</tr>
<tr>
<td>B-36</td>
<td>Encodings of AVX Instructions</td>
<td>B-71</td>
</tr>
<tr>
<td>B-37</td>
<td>General Floating-Point Instruction Formats</td>
<td>B-108</td>
</tr>
<tr>
<td>B-38</td>
<td>Floating-Point Instruction Formats and Encodings</td>
<td>B-108</td>
</tr>
<tr>
<td>B-39</td>
<td>Encodings for VMX Instructions</td>
<td>B-112</td>
</tr>
<tr>
<td>B-40</td>
<td>Encodings for SMX Instructions</td>
<td>B-113</td>
</tr>
<tr>
<td>C-1</td>
<td>Simple Intrinsics</td>
<td>C-2</td>
</tr>
<tr>
<td>C-2</td>
<td>Composite Intrinsics</td>
<td>C-14</td>
</tr>
</tbody>
</table>
CHAPTER 1
ABOUT THIS MANUAL

The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D: Instruction Set Reference (order numbers 253666, 253667, 326018, and 334569), is part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:

- The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers (order number 335592).

The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, describes the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C, & 3D, describes the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, addresses the programming environment for classes of software that host operating systems. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, describes the model-specific registers of Intel 64 and IA-32 processors.

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™ 2 Duo processor
- Intel® Core™ 2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™ 2 Extreme processor X7000 and X6800 series
- Intel® Core™ 2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
• Intel® Pentium® Dual-Core processor
• Intel® Xeon® processor 7200, 7300 series
• Intel® Xeon® processor 5200, 5400, 7400 series
• Intel® Core™ 2 Extreme processor QX9000 and X9000 series
• Intel® Core™ 2 Quad processor Q9000 series
• Intel® Core™ 2 Duo processor E8000, T9000 series
• Intel Atom® processor family
• Intel Atom® processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
• Intel® Core™ i7 processor
• Intel® Core™ i5 processor
• Intel® Xeon® processor E7-8800/4800/2800 product families
• Intel® Core™ i7-3930K processor
• 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
• Intel® Xeon® processor E3-1200 product family
• Intel® Xeon® processor E5-2400/1400 product family
• Intel® Xeon® processor E5-4600/2600/1600 product family
• 3rd generation Intel® Core™ processors
• Intel® Xeon® processor E3-1200 v2 product family
• Intel® Xeon® processor E5-2400/1400 v2 product families
• Intel® Xeon® processor E5-4600/2600/1600 v2 product families
• Intel® Xeon® processor E7-8800/4800/2800 v2 product families
• 4th generation Intel® Core™ processors
• The Intel® Core™ M processor family
• Intel® Core™ i7-59xx Processor Extreme Edition
• Intel® Core™ i7-49xx Processor Extreme Edition
• Intel® Xeon® processor E3-1200 v3 product family
• Intel® Xeon® processor E5-2600/1600 v3 product families
• 5th generation Intel® Core™ processors
• Intel® Xeon® processor D-1500 product family
• Intel® Xeon® processor E5 v4 family
• Intel Atom® processor X7-Z8000 and X5-Z8000 series
• Intel Atom® processor Z3400 series
• Intel Atom® processor Z3500 series
• 6th generation Intel® Core™ processors
• Intel® Xeon® processor E3-1500m v5 product family
• 7th generation Intel® Core™ processors
• Intel® Xeon® Phi™ Processor 3200, 5200, 7200 Series
• Intel® Xeon® Scalable Processor Family
• 8th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
• Intel® Xeon® E processors
• 9th generation Intel® Core™ processors
• 2nd generation Intel® Xeon® Scalable Processor Family
• 10th generation Intel® Core™ processors
• 11th generation Intel® Core™ processors
• 3rd generation Intel® Xeon® Scalable Processor Family
• 12th generation Intel® Core™ processors
• 13th generation Intel® Core™ processors
• 4th generation Intel® Xeon® Scalable Processor Family
• 5th generation Intel® Xeon® Scalable Processor Family
• Intel® Core™ Ultra 7 processors

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.


The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™ 2 Duo, Intel® Core™ 2 Quad, and Intel® Core™ 2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™ 2 Quad processor Q9000 series, and Intel® Core™ 2 Extreme processors QX9000, X9000 series, Intel® Core™ 2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.


P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™ 2 Duo, Intel® Core™ 2 Extreme, Intel® Core™ 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel Atom® processor Z8000 series is based on the Airmont microarchitecture.
The Intel Atom® processor Z3400 series and the Intel Atom® processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Scalable Processor Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor C series, the Intel Atom® processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Scalable Processor Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Scalable Processor Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

The 12th generation Intel® Core™ processors are based on the Alder Lake performance hybrid architecture and support Intel 64 architecture.

The 13th generation Intel® Core™ processors are based on the Raptor Lake performance hybrid architecture and support Intel 64 architecture.

The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture and supports Intel 64 architecture.

The 5th generation Intel® Xeon® Scalable Processor Family is based on Emerald Rapids microarchitecture and supports Intel 64 architecture.

The Intel® Core™ Ultra 7 processor is based on Meteor Lake hybrid architecture and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

1.2 OVERVIEW OF VOLUME 2A, 2B, 2C, AND 2D: INSTRUCTION SET REFERENCE

A description of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, content follows:
Chapter 1 — About This Manual. Gives an overview of all ten volumes of the Intel® 64 and IA-32 Architectures Software Developer’s Manual. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — Instruction Format. Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

Chapter 3 — Instruction Set Reference, A-L. Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3/SSSE3/SSE4 extensions, and system instructions are included.

Chapter 4 — Instruction Set Reference, M-U. Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It starts Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

Chapter 5 — Instruction Set Reference, V. Continues the description of Intel 64 and IA-32 instructions started in chapters 3 and 4. This chapter starts Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C.

Chapter 6 — Instruction Set Reference, W-Z. Continues the description of Intel 64 and IA-32 instructions started in chapters 3, 4, and 5. It provides the balance of the alphabetized list of instructions and starts Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

Chapter 7 — Safer Mode Extensions Reference. Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions.

Chapter 8— Instruction Set Reference Unique to Intel® Xeon Phi™ Processors. Describes the instruction set that is unique to Intel® Xeon Phi™ processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors.

Appendix A — Opcode Map. Gives an opcode map for the IA-32 instruction set.

Appendix B — Instruction Formats and Encodings. Gives the binary encoding of each form of each IA-32 instruction.

Appendix C — Intel® C/C++ Compiler Intrinsics and Functional Equivalents. Lists the Intel® C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.
1.3.2  **Reserved Bits and Software Compatibility**

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

**NOTE**

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

1.3.3  **Instruction Operands**

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1, argument2, and argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:
LOADREG: MOV EAX, SUBTOTAL
In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.4 Hexadecimal and Binary Numbers
Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

1.3.5 Segmented Addressing
The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an address space.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called segments. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

1.3.6 Exceptions
An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

1.3.7 A New Syntax for CPUID, CR, and MSR Values
Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.
1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

See also:
• The latest security information on Intel® products:
• Software developer resources, guidance, and insights for security advisories:
  https://software.intel.com/security-software-guidance/
• The data sheet for a particular Intel 64 or IA-32 processor
• The specification update for a particular Intel 64 or IA-32 processor
• Intel® C++ Compiler documentation and online help:
• Intel® Fortran Compiler documentation and online help: http://software.intel.com/en-us/articles/intel-compilers/
• Intel® 64 and IA-32 Architectures Software Developer’s Manual (in one, four or ten volumes): https://software.intel.com/en-us/articles/intel-sdm
• Intel® Software Guard Extensions (Intel® SGX) Information: https://software.intel.com/en-us/isa-extensions/intel-sgx
• Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor: https://software.intel.com/sites/default/files/22/30/25602
• Performance Monitoring Unit Sharing Guide: http://software.intel.com/file/30388

Literature related to select features in future Intel processors are available at:

More relevant links are:
• Intel® Developer Zone: https://software.intel.com/en-us
• Processor support general link: http://www.intel.com/support/processors/
This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. (F3H is also used as a mandatory prefix for some instructions.)

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.
— BND prefix is encoded using F2H if the following conditions are true:
  • CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.
  • BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
  • When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Appendix E, “Intel® Memory Protection Extensions,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

• Group 2
  — Segment override prefixes:
    • 2EH—CS segment override (use with any branch instruction is reserved).
    • 36H—SS segment override prefix (use with any branch instruction is reserved).
    • 3EH—DS segment override prefix (use with any branch instruction is reserved).
    • 26H—ES segment override prefix (use with any branch instruction is reserved).
    • 64H—FS segment override prefix (use with any branch instruction is reserved).
    • 65H—GS segment override prefix (use with any branch instruction is reserved).
  — Branch hints¹:
    • 2EH—Branch not taken (used only with Jcc instructions).
    • 3EH—Branch taken (used only with Jcc instructions).

• Group 3
  • Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).

• Group 4
  • 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, “Instruction Set Reference, A-L,” for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H,F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (Jcc). Other use of branch hint prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

¹ Some earlier microarchitectures used these as branch hints, but recent generations have not and they are reserved for future hint usage.
2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The mod field combines with the r/m field to form 32 possible values: eight registers and 24 addressing modes.
- The reg/opcode field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.
- The r/m field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. Sometimes, certain combinations of the mod field and the r/m field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The scale field specifies the scale factor.
- The index field specifies the register number of the index register.
- The base field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.
2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled “REG =”). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute.

If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled “/digit (Opcode)”). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label “Value of ModR/M Byte (in Hexadecimal)”) contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4, and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1, and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

**Figure 2-2. Table Interpretation of ModR/M Byte (C8H)**
### INSTRUCTION FORMAT

**Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte**

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BX+SI]</td>
<td>00</td>
<td>00 00 00 08 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BX+DI]</td>
<td>01</td>
<td>00 01 09 11 19 21 29 31 39 3F</td>
</tr>
<tr>
<td>[BP+SI]</td>
<td>01</td>
<td>00 01 0A 12 1A 22 2A 32 3A 3F</td>
</tr>
<tr>
<td>[BP+DI]</td>
<td>01</td>
<td>00 01 0B 13 1B 23 2B 33 3B 3F</td>
</tr>
<tr>
<td>[SI]</td>
<td>100</td>
<td>00 10 14 1C 24 2C 34 3C 3F 3F</td>
</tr>
<tr>
<td>[DI]</td>
<td>101</td>
<td>00 15 1D 25 2D 35 3D 3F 3F 3F</td>
</tr>
<tr>
<td>disp16</td>
<td>110</td>
<td>00 00 0E 16 1E 26 2E 36 3E 3F</td>
</tr>
<tr>
<td>[BX]</td>
<td>111</td>
<td>00 07 0F 17 1F 27 2F 37 3F 3F</td>
</tr>
<tr>
<td>[BX+SI]+disp8</td>
<td>01</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BX+DI]+disp8</td>
<td>01</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BP+SI]+disp8</td>
<td>01</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BP+DI]+disp8</td>
<td>01</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[SI]+disp8</td>
<td>100</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[DI]+disp8</td>
<td>101</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>disp16+disp8</td>
<td>110</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BX]+disp8</td>
<td>111</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BX]+disp16</td>
<td>10</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BX]+disp16</td>
<td>10</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BP]+disp16</td>
<td>10</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BP]+disp16</td>
<td>10</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[SI]+disp16</td>
<td>100</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[DI]+disp16</td>
<td>101</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>disp16+disp16</td>
<td>110</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[BX]+disp16</td>
<td>111</td>
<td>00 01 00 0A 10 18 20 28 30 38</td>
</tr>
</tbody>
</table>

### NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.
Table 2-3. 32-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>00</td>
<td>00 08 10 18 20 2B 30 3B</td>
</tr>
<tr>
<td>[ECX]</td>
<td>001</td>
<td>01</td>
<td>09 11 19 21 29 31 39</td>
</tr>
<tr>
<td>[EDX]</td>
<td>010</td>
<td>02</td>
<td>0A 12 1A 22 2A 32 3A</td>
</tr>
<tr>
<td>[EBX]</td>
<td>011</td>
<td>03</td>
<td>0B 13 1B 23 2B 33 3B</td>
</tr>
<tr>
<td>[ESI]</td>
<td>100</td>
<td>04</td>
<td>0C 14 1C 24 2C 34 3C</td>
</tr>
<tr>
<td>[EDI]</td>
<td>110</td>
<td>06</td>
<td>0E 16 1E 26 2E 36 3E</td>
</tr>
<tr>
<td>[EBP]</td>
<td>111</td>
<td>07</td>
<td>0F 17 1F 27 2F 37 3F</td>
</tr>
<tr>
<td>[ESP]</td>
<td>000</td>
<td>01</td>
<td>02 03 04 05 06 07</td>
</tr>
<tr>
<td>[DI]</td>
<td>000</td>
<td>01</td>
<td>02 03 04 05 06 07</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The [-][-] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte’s base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4, and 5) and the scaling factor (determined by SIB byte bits 6 and 7).
INSTRUCTION FORMAT

NOTES:
1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:
   MOD bits       Effective Address
   00             [scaled index] + disp32
   01             [scaled index] + disp8 + [EBP]
   10             [scaled index] + disp32 + [EBP]

2.2 IA-32E MODE
IA-32e mode has two sub-modes. These are:
• Compatibility Mode. Enables a 64-bit operating system to run most legacy protected mode software unmodified.
• 64-Bit Mode. Enables a 64-bit operating system to run applications written to access 64-bit address space.

2.2.1 REX Prefixes
REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:
• Specify GPRs and SSE registers.
INSTRUCTION FORMAT

- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored, as are individual bits in the prefix when they have no meaning.

Only one meaningful REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can immediately precede the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

<table>
<thead>
<tr>
<th>Legacy Prefixes</th>
<th>REX Prefix</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grp 1, Grp 2, Grp 3, Grp 4 (optional)</td>
<td>(optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

Figure 2-3. Prefix Ordering in 64-bit Mode

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:
- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 through Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:
- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
• REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Position</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>7:4</td>
<td>0100</td>
</tr>
<tr>
<td>W</td>
<td>3</td>
<td>0 = Operand size determined by CS.D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = 64 Bit Operand Size</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
<td>Extension of the ModR/M reg field</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>Extension of the SIB index field</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Extension of the ModR/M r/m field, SIB base field, or Opcode reg field</td>
</tr>
</tbody>
</table>

**Table 2-4. REX Prefix Fields [BITS: 0100WRXB]**

**Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used**

**Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used**
In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte’s reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations. Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.
2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

48 B8 8877665544332211 MOV RAX,1122334455667788H

Table 2-5. Special Cases of REX Encodings

<table>
<thead>
<tr>
<th>ModR/M or SIB Byte</th>
<th>SIB Sub-field Encodings</th>
<th>Compatibility Mode Operation</th>
<th>Compatibility Mode Implications</th>
<th>Additional Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModR/M Byte mod = 11</td>
<td>r/m = b*100(ESP)</td>
<td>SIB byte present.</td>
<td>SIB byte required for ESP-based addressing.</td>
<td>REX prefix adds a fourth bit (b) which is not decoded (don’t care). SIB byte also required for R12-based addressing.</td>
</tr>
<tr>
<td>ModR/M Byte mod = 0</td>
<td>r/m = b*101(EBP)</td>
<td>Base register not used.</td>
<td>EBP without a displacement must be done using mod = 01 with displacement of 0.</td>
<td>REX prefix adds a fourth bit (b) which is not decoded (don’t care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.</td>
</tr>
<tr>
<td>SIB Byte index = 0100(ESP)</td>
<td>Index register not used.</td>
<td>ESP cannot be used as an index register.</td>
<td>REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.</td>
<td></td>
</tr>
<tr>
<td>SIB Byte base = 0101(EBP)</td>
<td>Base register is unused if mod = 0.</td>
<td>Base register depends on mod encoding.</td>
<td>REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
* Don’t care about value of REX.B

Table 2-6. Direct Memory Offset Form of MOV

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>MOV AL, moffset</td>
</tr>
<tr>
<td>A1</td>
<td>MOV EAX, moffset</td>
</tr>
<tr>
<td>A2</td>
<td>MOV moffset, AL</td>
</tr>
<tr>
<td>A3</td>
<td>MOV moffset, EAX</td>
</tr>
</tbody>
</table>
2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

<table>
<thead>
<tr>
<th>ModR/M and SIB Sub-field Encodings</th>
<th>Compatibility Mode Operation</th>
<th>64-bit Mode Operation</th>
<th>Additional Implications in 64-bit mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModR/M Byte</td>
<td>mod = 00</td>
<td>Disp32</td>
<td>In 64-bit mode, if one wants to use a Disp32 without specifying a base register, one can use a SIB byte encoding (indicated by ModR/M.r/m=100) as described in the next row.</td>
</tr>
<tr>
<td></td>
<td>r/m = 101 (none)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIB Byte</td>
<td>base = 101 (none)</td>
<td>If mod = 00, Disp32</td>
<td>Same as legacy</td>
</tr>
<tr>
<td></td>
<td>index = 100 (none)</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>scale = 0, 1, 2, 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

2.2.1.7 Default 64-BitOperand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8-DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).
2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

- Instruction syntax support for three operands and up-to four operands when necessary. For example, the third source register used by VBLENDVPD is encoded using bits 7:4 of the immediate byte.
- Encoding support for vector length of 128 bits (using XMM registers) and 256 bits (using YMM registers).
- Encoding support for instruction syntax of non-destructive source operands.
- Elimination of escape opcode byte (0FH), SIMD prefix byte (66H, F2H, F3H) via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access, memory addressing, or accessing XMM8-XMM15 (including YMM8-YMM15).
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only because only a subset of SIMD instructions need them.
- Extensibility for future instruction extensions without significant instruction length increase.

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, “single-purpose” byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

![Figure 2-8. Instruction Encoding Format with VEX Prefix](image)

2.3.2 VEX and the LOCK prefix

Any VEX-encoded instruction with a LOCK prefix preceding VEX will #UD.

2.3.3 VEX and the 66H, F2H, and F3H prefixes

Any VEX-encoded instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.

2.3.4 VEX and the REX prefix

Any VEX-encoded instruction with a REX prefix proceeding VEX will #UD.
2.3.5   The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

The bit fields of the VEX prefix can be summarized by its functional purposes:

- **Non-destructive source register encoding (applicable to three and four operand syntax):** This is the first source operand in the instruction syntax. It is represented by the notation, VEX.vvvv. This field is encoded using 1’s complement form (inverted form), i.e., XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.

- **Vector length encoding:** This 1-bit field represented by the notation VEX.L. L= 0 means vector length is 128 bits wide, L=1 means 256 bit vector. The value of this field is written as VEX.128 or VEX.256 in this document to distinguish encoded values of other VEX bit fields.

- **REX prefix functionality:** Full REX prefix functionality is provided in the three-byte form of VEX prefix. However the VEX bit fields providing REX functionality are encoded using 1’s complement form, i.e., XMM0/YMM0/R0 is encoded as 1111B, XMM15/YMM15/R15 is encoded as 0000B.
  
  — Two-byte form of the VEX prefix only provides the equivalent functionality of REX.R, using 1’s complement encoding. This is represented as VEX.R.
  
  — Three-byte form of the VEX prefix provides REX.R, REX.X, REX.B functionality using 1’s complement encoding and three dedicated bit fields represented as VEX.R, VEX.X, VEX.B.
  
  — Three-byte form of the VEX prefix provides the functionality of REX.W only to specific instructions that need to override default 32-bit operand size for a general purpose register to 64-bit size in 64-bit mode. For those applicable instructions, VEX.W field provides the same functionality as REX.W. VEX.W field can provide completely different functionality for other instructions.

- **Compaction of SIMD prefix:** Legacy SSE instructions effectively use SIMD prefixes (66H, F2H, F3H) as an opcode extension field. VEX prefix encoding allows the functional capability of such legacy SSE instructions (operating on XMM registers, bits 255:128 of corresponding YMM unmodified) to be encoded using the VEX.pp field without the presence of any SIMD prefix. The VEX-encoded 128-bit instruction will zero-out bits 255:128 of the destination register. VEX-encoded instruction may have 128 bit vector length or 256 bits length.

- **Compaction of two-byte and three-byte opcode:** More recently introduced legacy SSE instructions employ two and three-byte opcode. The one or two leading bytes are: 0FH, and 0FH 3AH/0FH 38H. The one-byte escape (0FH) and two-byte escape (0FH 3AH, 0FH 38H) can also be interpreted as an opcode extension field. The VEX.mmmmm field provides compaction to allow many legacy instruction to be encoded without the constant byte sequence, 0FH, 0FH 3AH, 0FH 38H. These VEX-encoded instruction may have 128 bit vector length or 256 bits length.

The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

The 3-byte VEX leaves room for future expansion with 3 reserved bits. REX and the 66h/F2h/F3h prefixes are reclaimed for future use.

VEX prefix has a two-byte form and a three byte form. If an instruction syntax can be encoded using the two-byte form, it can also be encoded using the three byte form of VEX. The latter increases the length of the instruction by one byte. This may be helpful in some situations for code alignment.

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

The VEX prefix will #UD on any instruction containing MMX register sources or destinations.
### 2.3.5.1 VEX Byte 0, bits [7:0]

VEX Byte 0, bits [7:0] must contain the value 11000101b (C5h) or 11000100b (C4h). The 3-byte VEX uses the C4h first byte, while the 2-byte VEX uses the C5h first byte.

### 2.3.5.2 VEX Byte 1, bit [7] - 'R'

VEX Byte 1, bit [7] contains a bit analogous to a bit inverted REX.R. In protected and compatibility modes the bit must be set to '1' otherwise the instruction is LES or LDS.

---

**Figure 2-9. VEX bit fields**

The following subsections describe the various fields in two or three-byte VEX prefix.

<table>
<thead>
<tr>
<th>(Bit Position)</th>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-byte VEX</td>
<td>11000100</td>
<td>R X B</td>
<td>m-mmmm</td>
</tr>
<tr>
<td></td>
<td>vvvv</td>
<td>W</td>
<td>pp</td>
</tr>
</tbody>
</table>

| 2-byte VEX     | 11000101        | R       | vvvv    |
|                | L pp            |         |

- **R**: REX.R in 1’s complement (inverted) form
  - 1: Same as REX.R=0 (must be 1 in 32-bit mode)
  - 0: Same as REX.R=1 (64-bit mode only)

- **X**: REX.X in 1’s complement (inverted) form
  - 1: Same as REX.X=0 (must be 1 in 32-bit mode)
  - 0: Same as REX.X=1 (64-bit mode only)

- **B**: REX.B in 1’s complement (inverted) form
  - 1: Same as REX.B=0 (Ignored in 32-bit mode).
  - 0: Same as REX.B=1 (64-bit mode only)

- **W**: opcode specific (use like REX.W, or used for opcode extension, or ignored, depending on the opcode byte)

- **m-mmmm**:
  - 00000: Reserved for future use (will #UD)
  - 00001: implied 0F leading opcode byte
  - 00010: implied 0F 38 leading opcode bytes
  - 00011: implied 0F 3A leading opcode bytes
  - 00100-11111: Reserved for future use (will #UD)

- **vvvv**: a register specifier (in 1’s complement form) or 1111 if unused.

- **L**: Vector Length
  - 0: scalar or 128-bit vector
  - 1: 256-bit vector

- **pp**: opcode extension providing equivalent functionality of a SIMD prefix
  - 00: None
  - 01: 66
  - 10: F3
  - 11: F2
This bit is present in both 2- and 3-byte VEX prefixes. The usage of WRXB bits for legacy instructions is explained in detail section 2.2.1.2 of Intel 64 and IA-32 Architectures Software developer’s manual, Volume 2A. This bit is stored in bit inverted format.

### 2.3.5.3 3-byte VEX byte 1, bit[6] · ‘X’

Bit[6] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.X. It is an extension of the SIB Index field in 64-bit modes. In 32-bit modes, this bit must be set to ‘1’ otherwise the instruction is LES or LDS. This bit is available only in the 3-byte VEX prefix. This bit is stored in bit inverted format.

### 2.3.5.4 3-byte VEX byte 1, bit[5] · ‘B’

Bit[5] of the 3-byte VEX byte 1 encodes a bit analogous to a bit inverted REX.B. In 64-bit modes, it is an extension of the ModR/M r/m field, or the SIB base field. In 32-bit modes, this bit is ignored. This bit is available only in the 3-byte VEX prefix. This bit is stored in bit inverted format.

### 2.3.5.5 3-byte VEX byte 2, bit[7] · ‘W’

Bit[7] of the 3-byte VEX byte 2 is represented by the notation VEX.W. It can provide following functions, depending on the specific opcode.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes for these instructions, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don’t care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

### 2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- ‘vvvv’ the Source or Dest Register Specifier

In 32-bit mode the VEX first byte C4 and C5 alias onto the LES and LDS instructions. To maintain compatibility with existing programs the VEX 2nd byte, bits [7:6] must be 11b. To achieve this, the VEX payload bits are selected to place only inverted, 64-bit valid fields (extended register selectors) in these upper bits. The 2-byte VEX Byte 1, bits [6:3] and the 3-byte VEX, Byte 2, bits [6:3] encode a field (shorthand VEX.vvvv) that for instructions with 2 or more source registers and an XMM or YMM or memory destination encodes the first source register specifier stored in inverted (1’s complement) form. VEX.vvvv is not used by the instructions with one source (except certain shifts, see below) or on instructions with no XMM or YMM or memory destination. If an instruction does not use VEX.vvvv then it should be set to 1111b otherwise instruction will #UD. In 64-bit mode all 4 bits may be used. See Table 2-8 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).
The VEX.vvvv field is encoded in bit inverted format for accessing a register operand.

2.3.6 Instruction Operand Encoding and VEX.vvvv, ModR/M

VEX-encoded instructions support three-operand and four-operand instruction syntax. Some VEX-encoded instructions have syntax with less than three operands, e.g., VEX-encoded pack shift instructions support one source operand and one destination operand.

The roles of VEX.vvvv, reg field of ModR/M byte (ModR/M.reg), r/m field of ModR/M byte (ModR/M.r/m) with respect to encoding destination and source operands vary with different type of instruction syntax. The role of VEX.vvvv can be summarized to three situations:

- VEX.vvvv encodes the first source register operand, specified in inverted (1’s complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1’s complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the “Opcode” column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 2-9. Instructions with a VEX.vvvv destination

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F 73 /7 lb</td>
<td>VPSLLDQ xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 73 /3 lb</td>
<td>VPSRLDQ xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 71 /2 lb</td>
<td>VPSRLW xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 72 /2 lb</td>
<td>VPSRLD xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 73 /2 lb</td>
<td>VPSRLQ xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 71 /4 lb</td>
<td>VPSRAW xmm1, xmm2, imm8</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMAT

The role of ModR/M.r/m field can be summarized to two situations:
• ModR/M.r/m encodes the instruction operand that references a memory address.
• For some instructions that do not support memory addressing semantics, ModR/M.r/m encodes either the destination register operand or a source register operand.

The role of ModR/M.reg field can be summarized to two situations:
• ModR/M.reg encodes either the destination register operand or a source register operand.
• For some instructions, ModR/M.reg is treated as an opcode extension and not used to encode any instruction operand.

For instruction syntax that support four operands, VEX.vvvv, ModR/M.r/m, ModR/M.reg encodes three of the four operands. The role of bits 7:4 of the immediate byte serves the following situation:
• Imm8[7:4] encodes the third source register operand.

2.3.6.1 3-byte VEX byte 1, bits[4:0] - “m-mmmm”

Bits[4:0] of the 3-byte VEX byte 1 encode an implied leading opcode byte (0F, 0F 38, or 0F 3A). Several bits are reserved for future use and will #UD unless 0.

Table 2-10. VEX.m-mmmm interpretation

<table>
<thead>
<tr>
<th>VEX.m-mmmm</th>
<th>Implied Leading Opcode Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000B</td>
<td>Reserved</td>
</tr>
<tr>
<td>00001B</td>
<td>0F</td>
</tr>
<tr>
<td>00010B</td>
<td>0F 3B</td>
</tr>
<tr>
<td>00011B</td>
<td>0F 3A</td>
</tr>
<tr>
<td>00100-11111B</td>
<td>Reserved</td>
</tr>
<tr>
<td>(2-byte VEX)</td>
<td>0F</td>
</tr>
</tbody>
</table>

VEX.m-mmmm is only available on the 3-byte VEX. The 2-byte VEX implies a leading 0Fh opcode byte.

2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2] - “L”

The vector length field, VEX.L, is encoded in bit[2] of either the second byte of 2-byte VEX, or the third byte of 3-byte VEX. If "VEX.L = 1", it indicates 256-bit vector operation. "VEX.L = 0" indicates scalar and 128-bit vector operations.

The instruction VZEROUPPER is a special case that is encoded with VEX.L = 0, although its operation zero’s bits 255:128 of all YMM registers accessible in the current operating mode.

See the following table.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F 72 /4 ib</td>
<td>VPSRAD xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 71 /6 ib</td>
<td>VPSLLW xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 72 /6 ib</td>
<td>VPSLLD xmm1, xmm2, imm8</td>
</tr>
<tr>
<td>VEX.128.66.0F 73 /6 ib</td>
<td>VPSLLLQ xmm1, xmm2, imm8</td>
</tr>
</tbody>
</table>
### 2.3.6.3 2-byte VEX byte 1, bits\[1:0\], and 3-byte VEX byte 2, bits \[1:0\] - “pp”

Up to one implied prefix is encoded by bits\[1:0\] of either the 2-byte VEX byte 1 or the 3-byte VEX byte 2. The prefix behaves as if it was encoded prior to VEX, but after all other encoded prefixes.

See the following table.

#### Table 2-12. VEX.pp interpretation

<table>
<thead>
<tr>
<th>pp</th>
<th>Implies this prefix after other prefixes but before VEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B</td>
<td>None</td>
</tr>
<tr>
<td>01B</td>
<td>66</td>
</tr>
<tr>
<td>10B</td>
<td>F3</td>
</tr>
<tr>
<td>11B</td>
<td>F2</td>
</tr>
</tbody>
</table>

### 2.3.7 The Opcode Byte

One (and only one) opcode byte follows the 2 or 3 byte VEX. Legal opcodes are specified in Appendix B, in color. Any instruction that uses illegal opcode will #UD.

### 2.3.8 The ModR/M, SIB, and Displacement Bytes

The encodings are unchanged but the interpretation of reg_field or rm_field differs (see above).

### 2.3.9 The Third Source Operand (Immediate Byte)

VEX-encoded instructions can support instruction with a four operand syntax. VBLENDVPD, VBLENDVPS, and PBLENDVB use imm8[7:4] to encode one of the source registers.

### 2.3.10 Intel® AVX Instructions and the Upper 128-bits of YMM registers

If an instruction with a destination XMM register is encoded with a VEX prefix, the processor zeroes the upper bits (above bit 128) of the equivalent YMM register. Legacy SSE instructions without VEX preserve the upper bits.

#### 2.3.10.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255. (This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded...
with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

2.3.11 Intel® AVX Instruction Length

The Intel AVX instructions described in this document (including VEX and ignoring other prefixes) do not exceed 11 bytes in length, but may increase in the future. The maximum length of an Intel 64 and IA-32 instruction remains 15 bytes.

2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-13 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte’s base field. The register names also include R8D-R15D applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-13). In 32-bit mode, R8D-R15D does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e., XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.
### 2.3.12.1 64-bit Mode VSIB Memory Addressing

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose registers as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

### 2.4 INTEL® ADVANCED MATRIX EXTENSIONS (INTEL® AMX)

Intel® AMX instructions follow the general documentation convention established in previous sections. Additionally, Intel® Advanced Matrix Extensions use notation conventions as described below.

In the instruction encoding boxes, `sibmem` is used to denote an encoding where a ModR/M byte and SIB byte are used to indicate a memory operation where the base and displacement are used to point to memory, and the index
register (if present) is used to denote a stride between memory rows. The index register is scaled by the sib.scale field as usual. The base register is added to the displacement, if present.

In the instruction encoding, the ModR/M byte is represented several ways depending on the role it plays. The ModR/M byte has 3 fields: 2-bit ModR/M.mod field, a 3-bit ModR/M.reg field and a 3-bit ModR/M.r/m field. When all bits of the ModR/M byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the ModR/M byte must contain fixed values, those values are specified as follows:

• If only the ModR/M.mod must be 0b11, and ModR/M.reg and ModR/M.r/m fields are unrestricted, this is denoted as 11:rrr:bbb. The rrr correspond to the 3-bits of the ModR/M.reg field and the bbb correspond to the 3-bits of the ModR/M.r/m field.

• If the ModR/M.mod field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then the notation !(11) is used.

• If the ModR/M.reg field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

**NOTE**

Historically this document only specified the ModR/M.reg field restrictions with the notation /0 ... /7 and did not specify restrictions on the ModR/M.mod and ModR/M.r/m fields in the encoding boxes.

### 2.5 INTEL® AVX AND INTEL® SSE INSTRUCTION EXCEPTION CLASSIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.5.1 through 2.6.1. For example, ADDPS contains the entry: "See Exceptions Type 2"

In this entry, "Type2" can be looked up in Table 2-14.

The instruction’s corresponding CPUID feature flag can be identified in the fourth column of the Instruction summary table.

Note: #UD on CPUID feature flags=0 is not guaranteed in a virtualized environment if the hardware supports the feature flag.

**NOTE**

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 23.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction set</th>
<th>Mem arg</th>
<th>Floating-Point Exceptions (#XM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>AVX, Legacy SSE</td>
<td>16/32 byte explicitly aligned</td>
<td>None</td>
</tr>
<tr>
<td>Type 2</td>
<td>AVX, Legacy SSE</td>
<td>16/32 byte not explicitly aligned</td>
<td>Yes</td>
</tr>
<tr>
<td>Type 3</td>
<td>AVX, Legacy SSE</td>
<td>&lt; 16 byte</td>
<td>Yes</td>
</tr>
<tr>
<td>Type 4</td>
<td>AVX, Legacy SSE</td>
<td>16/32 byte not explicitly aligned</td>
<td>No</td>
</tr>
<tr>
<td>Type 5</td>
<td>AVX, Legacy SSE</td>
<td>&lt; 16 byte</td>
<td>No</td>
</tr>
<tr>
<td>Type 6</td>
<td>AVX (no Legacy SSE)</td>
<td>Varies</td>
<td>(At present, none do)</td>
</tr>
<tr>
<td>Type 7</td>
<td>AVX, Legacy SSE</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Type 8</td>
<td>AVX</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Type 11</td>
<td>F16C</td>
<td>8 or 16 byte, Not explicitly aligned, no AC#</td>
<td>Yes</td>
</tr>
<tr>
<td>Type 12</td>
<td>AVX2 Gathers</td>
<td>Not explicitly aligned, no AC#</td>
<td>No</td>
</tr>
</tbody>
</table>

See Table 2-15 for lists of instructions in each exception class.
<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>(V)MOVAPD, (V)MOVAPS, (V)MOVQ, (V)MOVDQA, (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQ, (V)MOVNTPD, (V)MOVNTPS</td>
</tr>
<tr>
<td>Type 2</td>
<td>(V)ADDPD, (V)ADDPS, (V)ADDSD, (V)ADDSP, (V)ADDSUBPD, (V)ADDSUBPS, (V)ADDSUBSD, (V)ADDSUBSP, (V)ADDSUBSD, (V)ADDSUBSP</td>
</tr>
<tr>
<td>Type 3</td>
<td>(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 4</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 5</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 6</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 7</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 8</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 9</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 10</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 11</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
<tr>
<td>Type 12</td>
<td>(V)AESDECLAST, (V)AESDECLAST, (V)AESENC, (V)AESENCCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDNDP, (V)ANDNPS, (V)ANDNPD, (V)ANDNPS, (V)ANDNPD,</td>
</tr>
</tbody>
</table>

(*) - Additional exception restrictions are present - see the Instruction description for details
(**) - Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s, i.e., no alignment checks are performed.

(***) - PCMPESTRI, PCMPSTRM, PCMPISTRI, PCMPISTRM, and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-17 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-16. #UD Exception and VEX.W=1 Encoding

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>#UD if VEX.W = 1 in all modes</th>
<th>#UD if VEX.W = 1 in non-64-bit modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>VBLENDVDPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLENDD, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMLPD, VPERMLPS, VPERM2F128</td>
<td></td>
</tr>
<tr>
<td>Type 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 4</td>
<td>VBLENDVDPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLENDD, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMLPD, VPERMLPS, VPERM2F128</td>
<td></td>
</tr>
<tr>
<td>Type 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 6</td>
<td>VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSETTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSETTI128</td>
<td></td>
</tr>
<tr>
<td>Type 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 11</td>
<td>VCVTPH2PS, VCVTPS2PH</td>
<td></td>
</tr>
<tr>
<td>Type 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Class</td>
<td>#UD If VEX.L = 0</td>
<td>#UD If (VEX.L = 1 &amp; AVX2 not present &amp; AVX present)</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>Type 1</td>
<td>VMOVNTDQA</td>
<td></td>
</tr>
<tr>
<td>Type 2</td>
<td>VDPPD</td>
<td>VDPPD</td>
</tr>
<tr>
<td>Type 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 5</td>
<td>VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLDPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVXSX/ZX, VLDMXCSR, VSTMXCSR</td>
<td>Same as column 3</td>
</tr>
<tr>
<td>Type 6</td>
<td>VEXTRACTF128, VPERMF128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,</td>
<td>VMOVHLPS, VMOVHLP</td>
</tr>
<tr>
<td>Type 7</td>
<td>VMOVLHP, VMOVHLPS, VPMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLL, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ</td>
<td>VMOVHLPS, VMOVHLP</td>
</tr>
<tr>
<td>Type 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.5.1 Exceptions Type 1 (Aligned Memory Reference)

#### Table 2-18. Type 1 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invalid Opcode, #UD</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td><strong>Device Not Available, #NM</strong></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td><strong>Stack, #SS(0)</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td><strong>General Protection, #GP(0)</strong></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>VEX.256: Memory operand is not 32-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VEX.128: Memory operand is not 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS seg-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td><strong>Page Fault #PF(fault-code)</strong></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
## 2.5.2 Exceptions Type 2 (>=16 Byte Memory Reference, Unaligned)

### Table 2-19. Type 2 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td>X X X X X X If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X Legacy SSE instruction:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If CR0.EM[bit 2] = 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If CR4.OSFXSR[bit 9] = 0.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If preceded by a LOCK prefix (F0H).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If any corresponding CPUID feature flag is '0'.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X Device Not Available, #NM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If CR0.TS[bit 3]=1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td>X X X X If a memory address referencing the SS segment is in a non-canonical form.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand and effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td>X X X X Legacy SSE; Memory operand is not 16-byte aligned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If the memory address is in a non-canonical form.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X X X X If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>X X X X For a page fault.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Exceptions Type 3 (<16 Byte Memory Argument)

#### Table 2-20. Type 3 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ≠ ‘11b’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (FOH).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
2.5.4 Exceptions Type 4 (>=16 Byte Mem Arg, No Alignment, No Floating-point Exceptions)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] ? '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>Legacy SSE: Memory operand is not 16-byte aligned.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>

NOTES:
1. LDDQU, MOVUPD, MOVUPS, PCMPESTRI, PCMPESTRM, PCMPISTRI, and PCMPISTRM instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.
### Exceptions Type 5 (<16 Byte Mem Arg and No FP Exceptions)

Table 2-22. Type 5 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td>x</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>x</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(O)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>x</td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
### Table 2-23. Type 6 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
### 2.5.7 Exceptions Type 7 (No FP Exceptions, No Memory Arg)

Table 2-24. Type 7 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] = '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>Legacy SSE instruction:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3] = 1.</td>
</tr>
</tbody>
</table>

### 2.5.8 Exceptions Type 8 (AVX and No Memory Argument)

Table 2-25. Type 8 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Always in Real or Virtual-8086 mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If XCR0[2:1] = '11b'.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CPUID.01H.ECX.AVX[bit 28] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If VEX.vvvv = 1111B.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If proceeded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3] = 1.</td>
</tr>
</tbody>
</table>
## 2.5.9 Exceptions Type 11 (VEX-only, Mem Arg, No AC, Floating-point Exceptions)

Table 2-26. Type 11 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>VEX prefix:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (FOH).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>SIMD Floating-Point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMEXCEPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
2.5.10 Exceptions Type 12 (VEX-only, VSIB Mem Arg, No AC, No Floating-point Exceptions)

Table 2-27. Type 12 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>VEX prefix.</td>
<td>If XCR0[2:1] = '11b'. If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>NA</td>
<td></td>
<td>If address size attribute is 16 bit.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If ModR/M.mod = '11b'.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If ModR/M.rm = '100b'.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any vector register is used more than once between the destination register,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mask register and the index register in VSIB addressing.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>segments.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
<td></td>
</tr>
</tbody>
</table>

2.6 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.
Any VEX-encoded GPR instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.
Any VEX-encoded GPR instruction with a REX prefix proceeding VEX will #UD.
VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

2.6.1 Exceptions Type 13 (VEX-Encoded GPR Instructions)
The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GRP instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

Table 2-28. VEX-Encoded GPR Instructions

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 13</td>
<td>ANDN, BEXTR, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX</td>
</tr>
</tbody>
</table>

(*) - Additional exception restrictions are present - see the Instruction description for details.

Table 2-29. Type 13 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X If BMI1/BMI2 CPUID feature flag is '0'. If BMI1/BMI2 CPUID feature flag is '0'.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If a VEX prefix is present.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If VEX.L = 1.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
<td></td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For an illegal address in the SS segment.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
<td></td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
<td></td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
<td></td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
<td></td>
</tr>
</tbody>
</table>

2.7 INTEL® AVX-512 ENCODING
The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.
The significant feature differences between EVEX and VEX are summarized below.

- **EVEX** is a 4-Byte prefix (the first byte must be 62H); **VEX** is either a 2-Byte (C5H is the first byte) or 3-Byte (C4H is the first byte) prefix.
- **EVEX** prefix can encode 32 vector registers (XMM/YMM/ZMM) in 64-bit mode.
- **EVEX** prefix can encode an opmask register for conditional processing or selection control in EVEX-encoded vector instructions. Opmask instructions, whose source/destination operands are opmask registers and treat the content of an opmask register as a single value, are encoded using the VEX prefix.
- **EVEX** memory addressing with disp8 form uses a compressed disp8 encoding scheme to improve the encoding density of the instruction byte stream.
- **EVEX** prefix can encode functionality that are specific to instruction classes (e.g., packed instruction with "load+op" semantic can support embedded broadcast functionality, floating-point instruction with rounding semantic can support static rounding functionality, floating-point instruction with non-rounding arithmetic semantic can support "suppress all exceptions" functionality).

### 2.7.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10. Note that the values contained within brackets are optional.

![Figure 2-10. Intel® AVX-512 Instruction Format and the EVEX Prefix](image)

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).

![Figure 2-11. Bit Field Layout of the EVEX Prefix](image)

**NOTES:**
1. See Table 2-30 for additional details on bit fields.
**INSTRUCTION FORMAT**

The bit fields in P[23:0] are divided into the following functional groups (Table 2-30 provides a tabular summary):

- **Reserved bits**: P[3] must be 0, otherwise #UD.
- **Fixed-value bit**: P[10] must be 1, otherwise #UD.
- **Compressed legacy prefix/escape bytes**: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- **EVEX.mmm**: P[2:0] provides access to up to eight decoding maps. Currently, only the following decoding maps are supported: 1, 2, 3, 5, and 6.
- **Operand specifier modifier bits for vector register, general purpose register, memory addressing**: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- **Operand specifier modifier bit for vector register**: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- **Non-destructive source /vector index operand specifier**: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- **Op-mask register specifiers**: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- **EVEX.W**: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- **Vector destination merging/zeroing**: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- **Broadcast/Static-rounding/SAE context bit**: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:

---

### Table 2-30. EVEX Prefix Bit Field Functional Grouping

<table>
<thead>
<tr>
<th>Notation</th>
<th>Bit field Group</th>
<th>Position</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.mmm</td>
<td>Access to up to eight decoding maps</td>
<td>P[2:0]</td>
<td>Currently, only the following decoding maps are supported: 1, 2, 3, 5, and 6.</td>
</tr>
<tr>
<td>--</td>
<td>Reserved</td>
<td>P[3]</td>
<td>Must be 0.</td>
</tr>
<tr>
<td>EVEX.R'</td>
<td>High-16 register specifier modifier</td>
<td>P[4]</td>
<td>Combine with EVEX.R and ModR/M.reg. This bit is stored in inverted format.</td>
</tr>
<tr>
<td>EVEX.RXB</td>
<td>Next-8 register specifier modifier</td>
<td>P[7:5]</td>
<td>Combine with ModR/M.reg, ModR/M.rm (base, index/vidx). This field is encoded in bit inverted format.</td>
</tr>
<tr>
<td>EVEX.vvvv</td>
<td>VVVV register specifier</td>
<td>P[14:11]</td>
<td>Same as VEX.vvvv. This field is encoded in bit inverted format.</td>
</tr>
<tr>
<td>EVEX.W</td>
<td>Operand size promotion/Opcode extension</td>
<td>P[15]</td>
<td></td>
</tr>
<tr>
<td>EVEX.aaa</td>
<td>Embedded opmask register specifier</td>
<td>P[18:16]</td>
<td></td>
</tr>
<tr>
<td>EVEX.V'</td>
<td>High-16 VVVV/VIDX register specifier</td>
<td>P[19]</td>
<td>Combine with EVEX.vvvv or when VSIB present. This bit is stored in inverted format.</td>
</tr>
<tr>
<td>EVEX.b</td>
<td>Broadcast/RC/SAE Context</td>
<td>P[20]</td>
<td></td>
</tr>
<tr>
<td>EVEX.L'L</td>
<td>Vector length/RC</td>
<td>P[22:21]</td>
<td></td>
</tr>
<tr>
<td>EVEX.z</td>
<td>Zeroing/Merging</td>
<td>P[23]</td>
<td></td>
</tr>
</tbody>
</table>
— Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
— Redirect L’L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies “Suppress all exceptions” (SAE).
— Enable SAE for floating-point instructions with arithmetic semantic that is not rounding.
— For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.

- Vector length/rounding control specifier: P[22:21] can serve one of three options.
  - Vector length information for packed vector instructions.
  - Ignored for instructions operating on vector register content as a single data element.
  - Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

### 2.7.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-31. Opmask register encoding is described in Section 2.7.3.

#### Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

<table>
<thead>
<tr>
<th>REG</th>
<th>EVEX.R’</th>
<th>REX.R</th>
<th>modrm.reg</th>
<th>Common Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VVVV</td>
<td>EVEX.V’</td>
<td>EVEX.vvvv</td>
<td>modrm.reg</td>
<td>Destination or Source</td>
</tr>
<tr>
<td>RM</td>
<td>EVEX.X</td>
<td>EVEX.B</td>
<td>modrm.r/m</td>
<td>2nd Source or Destination</td>
</tr>
<tr>
<td>BASE</td>
<td>0</td>
<td>EVEX.B</td>
<td>modrm.r/m</td>
<td>1st Source or Destination</td>
</tr>
<tr>
<td>INDEX</td>
<td>0</td>
<td>EVEX.X</td>
<td>sib.index</td>
<td>memory addressing</td>
</tr>
<tr>
<td>VIDX</td>
<td>EVEX.V’</td>
<td>EVEX.X</td>
<td>sib.index</td>
<td>Vector</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-32.

#### Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode

<table>
<thead>
<tr>
<th>REG</th>
<th>[2:0]</th>
<th>Reg. Type</th>
<th>Common Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VVVV</td>
<td>modrm.reg</td>
<td>GPR, Vector</td>
<td>Destination or Source</td>
</tr>
<tr>
<td>RM</td>
<td>modrm.r/m</td>
<td>GPR, Vector</td>
<td>1st Source or Destination</td>
</tr>
<tr>
<td>BASE</td>
<td>modrm.r/m</td>
<td>GPR</td>
<td>Memory Addressing</td>
</tr>
<tr>
<td>INDEX</td>
<td>sib.index</td>
<td>GPR</td>
<td>Memory Addressing</td>
</tr>
<tr>
<td>VIDX</td>
<td>sib.index</td>
<td>Vector</td>
<td>VSIB Memory Addressing</td>
</tr>
</tbody>
</table>
2.7.3 Opmask Register Encoding

There are eight opmask registers, k0-k7. Opmask register encoding falls into two categories:

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using standard modR/M byte’s reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.

- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.7.4).

- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte’s reg field and rm fields.

2.7.4 Masking Support in EVEX

EVEX can encode an opmask register to conditionally control per-element computational operation and updating of result of an instruction to the destination operand. The predicate operand is known as the opmask register. The EVEX.aaa field, P[18:16] of the EVEX prefix, is used to encode one out of a set of eight 64-bit architectural registers. Note that from this set of 8 architectural registers, only k1 through k7 can be addressed as predicate operands. k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

AVX-512 instructions support two types of masking with EVEX.z bit (P[23]) controlling the type of masking:

- Merging-masking, which is the default type of masking for EVEX-encoded vector instructions, preserves the old value of each element of the destination where the corresponding mask bit has a 0. It corresponds to the case of EVEX.z = 0.

- Zeroing-masking, is enabled by having the EVEX.z bit set to 1. In this case, an element of the destination is set to 0 when the corresponding mask bit has a 0 value.

AVX-512 Foundation instructions can be divided into the following groups:

- Instructions which support “zeroing-masking”.
  - Also allow merging-masking.

- Instructions which require aaa = 000.
  - Do not allow any form of masking.

- Instructions which allow merging-masking but do not allow zeroing-masking.
  - Require EVEX.z to be set to 0.
  - This group is mostly composed of instructions that write to memory.

- Instructions which require aaa < 000 do not allow EVEX.z to be set to 1.
  - Allow merging-masking and do not allow zeroing-masking, e.g., gather instructions.

### Table 2-33. Opmask Register Specifier Encoding

<table>
<thead>
<tr>
<th>[2:0]</th>
<th>Register Access</th>
<th>Common Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>modrm.reg</td>
<td>k0-k7</td>
</tr>
<tr>
<td>VVVV</td>
<td>VEX.vvvv</td>
<td>k0-k7</td>
</tr>
<tr>
<td>RM</td>
<td>modrm.r/m</td>
<td>k0-7</td>
</tr>
<tr>
<td>{k1}</td>
<td>EVEX.aaa</td>
<td>k01-k7</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Instructions that overwrite the conditional mask in opmask do not permit using k0 as the embedded mask.
2.7.5 Compressed Displacement (disp8*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-34 and Table 2-35 below, where EVEX encoded instructions are classified using the tupletype attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-34 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.7.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 2-35. Table 2-35 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-35. Instruction classified in Table 2-35 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8*N rules still apply when using 16b addressing.

<table>
<thead>
<tr>
<th>TupleType</th>
<th>EVEX.b</th>
<th>InputSize</th>
<th>EVEX.W</th>
<th>Broadcast</th>
<th>N (VL=128)</th>
<th>N (VL=256)</th>
<th>N (VL=512)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>0</td>
<td>32bit</td>
<td>0</td>
<td>none</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>Load+Op (Full Vector Dword/Qword)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>32bit</td>
<td>0</td>
<td>{1tox}</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>64bit</td>
<td>1</td>
<td>none</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>64bit</td>
<td>1</td>
<td>{1tox}</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Half</td>
<td>0</td>
<td>32bit</td>
<td>0</td>
<td>none</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>Load+Op (Half Vector)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>32bit</td>
<td>0</td>
<td>{1tox}</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TupleType</th>
<th>InputSize</th>
<th>EVEX.W</th>
<th>(VL=128)</th>
<th>(VL=256)</th>
<th>(VL=512)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Mem</td>
<td>N/A</td>
<td>N/A</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>Load/store or subDword full vector</td>
</tr>
<tr>
<td>Tuple1 Scalar</td>
<td>8bit</td>
<td>N/A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1Tuple</td>
</tr>
<tr>
<td></td>
<td>16bit</td>
<td>N/A</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32bit</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64bit</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Tuple1 Fixed</td>
<td>32bit</td>
<td>N/A</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1 Tuple, memsize not affected by EVEX.W</td>
</tr>
<tr>
<td></td>
<td>64bit</td>
<td>N/A</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Tuple2</td>
<td>32bit</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>Broadcast (2 elements)</td>
</tr>
<tr>
<td></td>
<td>64bit</td>
<td>1</td>
<td>NA</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Tuple4</td>
<td>32bit</td>
<td>0</td>
<td>NA</td>
<td>16</td>
<td>16</td>
<td>Broadcast (4 elements)</td>
</tr>
<tr>
<td></td>
<td>64bit</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Tuple8</td>
<td>32bit</td>
<td>0</td>
<td>NA</td>
<td>16</td>
<td>32</td>
<td>Broadcast (8 elements)</td>
</tr>
<tr>
<td>Half Mem</td>
<td>N/A</td>
<td>N/A</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>SubQword Conversion</td>
</tr>
<tr>
<td>Quarter Mem</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>SubDword Conversion</td>
</tr>
</tbody>
</table>
2.7.6 EVEX Encoding of Broadcast/Rounding/SAE Support

EVEX.b can provide three types of encoding context, depending on the instruction classes:

- Embedded broadcasting of one data element from a source memory operand to the destination for vector instructions with "load+op" semantic.
- Static rounding control overriding MXCSR.RC for floating-point instructions with rounding semantic.
- “Suppress All exceptions” (SAE) overriding MXCSR mask control for floating-point arithmetic instructions that do not have rounding semantic.

2.7.7 Embedded Broadcast Support in EVEX

EVEX encodes an embedded broadcast functionality that is supported on many vector instructions with 32-bit (double word or single precision floating-point) and 64-bit data elements, and when the source operand is from memory. EVEX.b (P[20]) bit is used to enable broadcast on load-op instructions. When enabled, only one element is loaded from memory and broadcasted to all other elements instead of loading the full memory size.

The following instruction classes do not support embedded broadcasting:

- Instructions with only one scalar result is written to the vector destination.
- Instructions with explicit broadcast functionality provided by its opcode.
- Instruction semantic is a pure load or a pure store operation.

2.7.8 Static Rounding Support in EVEX

Static rounding control embedded in the EVEX encoding system applies only to register-to-register flavor of floating-point instructions with rounding semantic at two distinct vector lengths: (i) scalar, (ii) 512-bit. In both cases, the field EVEX.L’L expresses rounding mode control overriding MXCSR.RC if EVEX.b is set. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

2.7.9 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, “suppress all exceptions” is implied. The processor behaves as if all MXCSR masking controls are set.

2.7.10 Vector Length Orthogonality

The architecture of EVEX encoding scheme can support SIMD instructions operating at multiple vector lengths. Many AVX-512 Foundation instructions operate at 512-bit vector length. The vector length of EVEX encoded vector instructions are generally determined using the L’L field in EVEX prefix, except for 512-bit floating-point, reg-reg instructions with rounding semantic. The table below shows the vector length corresponding to various values of the L’L bits. When EVEX is used to encode scalar instructions, L’L is generally ignored.

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L’L fields are summarized in Table 2-36.
2.7.11 #UD Equations for EVEX

Instructions encoded using EVEX can face three types of UD conditions: state dependent, opcode independent and opcode dependent.

2.7.11.1 State Dependent #UD

In general, attempts of execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-37 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-37 will cause #UD.

2.7.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-38.
2.7.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-39 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Position</th>
<th>Operand Encoding</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.R</td>
<td>P[7]</td>
<td>ModRM.reg encodes k-reg</td>
<td>If EVEX.R = 0</td>
<td>None (BOUND if EVEX.RX != 11b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg is opcode extension</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg encodes all other registers</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td>EVEX.X</td>
<td>P[6]</td>
<td>ModRM.r/m encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m encodes k-reg or GPR</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m without SIB/VSIB</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m with SIB/VSIB</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td>EVEX.B</td>
<td>P[5]</td>
<td>ModRM.r/m encodes k-reg</td>
<td>None (valid)</td>
<td>None (valid)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m encodes other registers</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m base present</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.r/m base not present</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td>EVEX.R'</td>
<td>P[4]</td>
<td>ModRM.reg encodes k-reg or GPR</td>
<td>If 0</td>
<td>None (ignored)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg is opcode extension</td>
<td>None (ignored)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ModRM.reg encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td></td>
</tr>
<tr>
<td>EVEX.vvvv</td>
<td>P[14:11]</td>
<td>vvvv encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td>None (valid)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise</td>
<td>If != 1111b</td>
<td></td>
</tr>
<tr>
<td>EVEX.V'</td>
<td>P[19]</td>
<td>Encodes ZMM/YMM/XMM</td>
<td>None (valid)</td>
<td>If 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise</td>
<td>If 0</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-40 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

<table>
<thead>
<tr>
<th>Notation</th>
<th>Position</th>
<th>Operand Encoding</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.aaa</td>
<td>P[18:16]</td>
<td>Instructions do not use opmask for conditional processing</td>
<td>If aaaa != 000b</td>
<td>If aaaa != 000b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opmask used as conditional processing mask and updated at completion</td>
<td>If aaaa = 000b</td>
<td>If aaaa = 000b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opmask used as conditional processing</td>
<td>None (valid)</td>
<td>None (valid)</td>
</tr>
<tr>
<td>EVEX.z</td>
<td>P[23]</td>
<td>Vector instruction using opmask as source or destination</td>
<td>If EVEX.z != 0</td>
<td>If EVEX.z != 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store instructions or gather/scatter instructions</td>
<td>If EVEX.z != 0</td>
<td>If EVEX.z != 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instructions with EVEX.aaa = 000b.</td>
<td>If EVEX.z != 0</td>
<td>If EVEX.z != 0</td>
</tr>
<tr>
<td>VEX.vvvv</td>
<td></td>
<td>K-reg is instruction operands not mask control.</td>
<td>If vvvv = 0xxxb</td>
<td>None</td>
</tr>
</tbody>
</table>

NOTES:
1. E.g., VPBROADCASTMxxx, VPMOVVM2x, VPMOVx2M.
2. E.g., Gather/Scatter family.
3. aaaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of ‘all ones’ regardless of the actual value in K0.
4. E.g.,VFCLASSPD/PS, VCMPB/D/Q/W family, VPMOVVM2x, VPMOVx2M.
Table 2-41 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

### Table 2-41. #UD Conditions Dependent on EVEX.b Context

<table>
<thead>
<tr>
<th>Notation</th>
<th>Position</th>
<th>Operand Encoding</th>
<th>64-bit #UD</th>
<th>Non-64-bit #UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.L’Lb</td>
<td>P[22:20]</td>
<td>Reg-reg, FP instructions with rounding semantic.</td>
<td>None (valid 1)</td>
<td>None (valid 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other reg-reg, FP instructions that can cause #XM.</td>
<td>None (valid 2)</td>
<td>None (valid 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other reg-mem instructions in Table 2-34.</td>
<td>None (valid 3)</td>
<td>None (valid 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other instruction classes in Table 2-35.</td>
<td>If EVEX.b = 1</td>
<td>If EVEX.b = 1</td>
</tr>
</tbody>
</table>

**NOTES:**
1. L’L specifies rounding control, see Table 2-36, supports {er} syntax.
2. L’L is ignored.
3. L’L specifies vector length, see Table 2-36, supports embedded broadcast syntax
4. L’L specifies either vector length or ignored.

#### 2.7.12 Device Not Available

EVEX-encoded instructions follow the same rules when it comes to generating #NM (Device Not Available) exception. In particular, it is generated when CR0.TS[bit 3] = 1.

#### 2.7.13 Scalar Instructions

EVEX-encoded scalar SIMD instructions can access up to 32 registers in 64-bit mode. Scalar instructions support masking (using the least significant bit of the opmask register), but broadcasting is not supported.

#### 2.8 Exception Classifications of EVEX-Encoded Instructions

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

### Table 2-42. EVEX-Encoded Instruction Exception Class Summary

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction set</th>
<th>Mem arg</th>
<th>(#XM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type E1</td>
<td>Vector Moves/Load/Stores</td>
<td>Explicitly aligned, w/ fault suppression</td>
<td>None</td>
</tr>
<tr>
<td>Type E1NF</td>
<td>Vector Non-temporal Stores</td>
<td>Explicitly aligned, no fault suppression</td>
<td>None</td>
</tr>
<tr>
<td>Type E2</td>
<td>FP Vector Load+op</td>
<td>Support fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E2NF</td>
<td>FP Vector Load+op</td>
<td>No fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E3</td>
<td>FP Scalar/Partial Vector, Load+Op</td>
<td>Support fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E3NF</td>
<td>FP Scalar/Partial Vector, Load+Op</td>
<td>No fault suppression</td>
<td>Yes</td>
</tr>
<tr>
<td>Type E4</td>
<td>Integer Vector Load+op</td>
<td>Support fault suppression</td>
<td>No</td>
</tr>
<tr>
<td>Type E4NF</td>
<td>Integer Vector Load+op</td>
<td>No fault suppression</td>
<td>No</td>
</tr>
<tr>
<td>Type E5</td>
<td>Legacy-like Promotion</td>
<td>Varies, Support fault suppression</td>
<td>No</td>
</tr>
<tr>
<td>Type E5NF</td>
<td>Legacy-like Promotion</td>
<td>Varies, No fault suppression</td>
<td>No</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMAT

Table 2-42. EVEX-Encoded Instruction Exception Class Summary (Contd.)
Exception Class

Instruction set

Mem arg

(#XM)

Type E6

Post AVX Promotion

Varies, w/ fault suppression

No

Type E6NF

Post AVX Promotion

Varies, no fault suppression

No

Type E7NM

Register-to-register op

None

None

Type E9NF

Miscellaneous 128-bit

Vector-length Specific, no fault suppression

None

Type E10

Non-XF Scalar

Vector Length ignored, w/ fault suppression

None

Type E10NF

Non-XF Scalar

Vector Length ignored, no fault suppression

None

Type E11

VCVTPH2PS, VCVTPS2PH

Half Vector Length, w/ fault suppression

Yes

Type E12

Gather and Scatter Family

VSIB addressing, w/ fault suppression

None

Type E12NP

Gather and Scatter Prefetch Family

VSIB addressing, w/o page fault

None

Table 2-43 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-43. EVEX Instructions in Each Exception Class
Exception Class
Type E1
Type E1NF

Instruction
VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS

Type E2

VADDPD, VADDPH, VADDPS, VCMPPD, VCMPPH, VCMPPS, VCVTDQ2PH, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PH,
VCVTPD2PS, VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPH2DQ, VCVTPH2PD, VCVTPH2QQ, VCVTPH2UDQ,
VCVTPH2UQQ, VCVTPH2UW, VCVTPH2W, VCVTPS2DQ, VCVTPS2UDQS, VCVTQQ2PD, VCVTQQ2PH, VCVTQQ2PS,
VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPH2DQ, VCVTTPH2QQ, VCVTTPH2UDQ,
VCVTTPH2UQQ, VCVTTPH2UW, VCVTTPH2W, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PH, VCVTUDQ2PS,
VCVTUQQ2PD, VCVTUQQ2PH, VCVTUQQ2PS, VCVTUW2PH, VCVTW2PH, VDIVPD, VDIVPH, VDIVPS, VEXP2PD,
VEXP2PS, VFIXUPIMMPD, VFIXUPIMMPS, VFMADDxxxPD, VFMADDxxxPH, VFMADDxxxPS, VFMADDSUBxxxPD,
VFMADDSUBxxxPH, VFMADDSUBxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPH, VFMSUBADDxxxPS,
VFMSUBxxxPD, VFMSUBxxxPH, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPH, VFNMADDxxxPS,
VFNMSUBxxxPD, VFNMSUBxxxPH, VFNMSUBxxxPS, VGETEXPPD, VGETEXPPH, VGETEXPPS, VGETMANTPD,
VGETMANTPH, VGETMANTPS, VGETMANTSH, VMAXPD, VMAXPH, VMAXPS, VMINPD, VMINPH, VMINPS, VMULPD,
VMULPH, VMULPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPH, VREDUCEPS, VRNDSCALEPD,
VRNDSCALEPH, VRNDSCALEPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS, VSCALEFPD, VSCALEFPS,
VSQRTPD, VSQRTPH, VSQRTPS, VSUBPD, VSUBPH, VSUBPS

Type E3

VADDSD, VADDSH, VADDSS, VCMPSD, VCMPSH, VCMPSS, VCVTPS2QQ, VCVTPS2UQQ, VCVTPS2PD, VCVTSD2SH,
VCVTSD2SS, VCVTSH2SD, VCVTSH2SS, VCVTSS2SD, VCVTSS2SH, VCVTTPS2QQ, VCVTTPS2UQQ, VDIVSD, VDIVSH,
VDIVSS, VFMADDxxxSD, VFMADDxxxSH, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSH, VFMSUBxxxSS,
VFNMADDxxxSD, VFNMADDxxxSH, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSH, VFNMSUBxxxSS,
VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSH, VGETEXPSS, VGETMANTSD, VGETMANTSH,
VGETMANTSS, VMAXSD, VMAXSH, VMAXSS, VMINSD, VMINSH, VMINSS, VMULSD, VMULSH, VMULSS, VRANGESD,
VRANGESS, VREDUCESD, VREDUCESH, VREDUCESS, VRNDSCALESD, VRNDSCALESH, VRNDSCALESS, VSCALEFSD,
VSCALEFSH, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS, VSQRTSD, VSQRTSH, VSQRTSS,
VSUBSD, VSUBSH, VSUBSS

Type E3NF

2-46 Vol. 2A

VCOMISD, VCOMISH, VCOMISS, VCVTSD2SI, VCVTSD2USI, VCVTSH2SI, VCVTSH2USI, VCVTSI2SD, VCVTSI2SH,
VCVTSI2SS, VCVTSS2SI, VCVTSS2USI, VCVTTSD2SI, VCVTTSD2USI, VCVTTSH2SI, VCVTTSH2USI, VCVTTSS2SI,
VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SH, VCVTUSI2SS, VUCOMISD, VUCOMISH, VUCOMISS


INSTRUCTION FORMAT

Table 2-43. EVEX Instructions in Each Exception Class (Contd.)
Exception Class

Instruction

Type E4

VANDPD, VANDPS, VANDNPD, VANDNPS, VBLENDMPD, VBLENDMPS, VFCMADDCPH, VFCMULCPH, VFMADDCPH,
VFMULCPH, VFPCLASSPD, VFPCLASSPH, VFPCLASSPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ,
VPANDD, VPANDQ, VPANDND, VPANDNQ, VPBLENDMB, VPBLENDMD, VPBLENDMQ, VPBLENDMW, VPCMPD,
VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ,
VPMADD52LUQ, VPMADD52HUQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD,
VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPROLD, VPROLQ, VPROLVD, VPROLVQ,
VPRORD, VPRORQ, VPRORVD, VPRORVQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRAVW, VPSRAVD, VPSRAVW,
VPSRAVQ, VPSRLD, VPSRLQ)1, VPSUBD, VPSUBQ, VPSUBUSB, VPSUBUSW, VPTERNLOGD, VPTERNLOGQ,
VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VPXORD, VPXORQ, VPSLLVD, VPSLLVQ, VRCP14PD,
VRCP14PS, VRCPPH, VRSQRT14PD, VRSQRT14PS, VRSQRTPH, VXORPD, VXORPS

E4.nb2

VCOMPRESSPD, VCOMPRESSPS, VEXPANDPD, VEXPANDPS, VMOVDQU8, VMOVDQU16, VMOVDQU32,
VMOVDQU64, VMOVUPD, VMOVUPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB,
VPADDUSW, VPAVGB, VPAVGW, VPCMPB, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPCMPW, VPCMPUB,
VPCMPUW, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VPMAXSB, VPMAXSW, VPMAXUB,
VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW,
VPSLLVW, VPSLLW, VPSRAW, VPSRLVW, VPSRLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB,
VPTESTMW, VPTESTNMB, VPTESTNMW

Type E4NF

VALIGND, VALIGNQ, VPACKSSDW, VPACKUSDW, VPCONFLICTD, VPCONFLICTQ, VPERMD, VPERMI2D, VPERMI2PS,
VPERMI2PD, VPERMI2Q, VPERMPD, VPERMPS, VPERMQ, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD,
VPERMILPD, VPERMILPS, VPMULTISHIFTQB, VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ,
VPUNPCKLQDQ, VSHUFF32X4, VSHUFF64X2, VSHUFI32X4, VSHUFI64X2, VSHUFPD, VSHUFPS, VUNPCKHPD,
VUNPCKHPS, VUNPCKLPD, VUNPCKLPS

E4NF.nb

VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP,
VPSADBW, VPSHUFB, VPSHUFHW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD,
VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)3, VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD,
VPERMW, VPERMI2W, VPERMT2W

Type E5

PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW,
PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ, VCVTDQ2PD, VCVTUDQ2PD, VMOVSH,
VPMOVSXxx, VPMOVZXxx,

2

Type E5NF

Type E6

Type E6NF
Type
E7NM.1284

VMOVDDUP
VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4,
VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4,
VBROADCASTSD, VBROADCASTSS, VFPCLASSSD, VFPCLASSSS, VPBROADCASTB, VPBROADCASTD,
VPBROADCASTW, VPBROADCASTQ, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW,
VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW,
VPMOVUSDW, VPMOVWB, VPMOVSWB, VPMOVUSWB
VEXTRACTF32X4, VEXTRACTF32X8, VEXTRACTF64X2, VEXTRACTF64X4, VEXTRACTI32X4, VEXTRACTI32X8,
VEXTRACTI64X2, VEXTRACTI64X4, VINSERTF32X4, VINSERTF32X8, VINSERTF64X2, VINSERTF64X4,
VINSERTI32X4, VINSERTI32X8, VINSERTI64X2, VINSERTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D
VMOVHLPS, VMOVLHPS

Type E7NM.

(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW)5, VPMOVB2M, VPMOVD2M, VPMOVM2B,
VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVQ2M, VPMOVW2M

Type E9NF

VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VMOVW, VPEXTRB,
VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ

Type E10

VFCMADDCSH, VFMADDCSH, VFCMULCSH, VFMULCSH, VFPCLASSSH, VMOVSD, VMOVSS, VRCP14SD, VRCP14SS,
VRCPSH, VRSQRT14SD, VRSQRT14SS, VRSQRTSH

Type E10NF
Type E11

(VCVTSI2SD, VCVTUSI2SD)6
VCVTPH2PS, VCVTPS2PH

Vol. 2A 2-47


### Table 2-43. EVEX Instructions in Each Exception Class (Contd.)

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type E12</td>
<td>VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDQ, VSCATTERQD, VSCATTERQQ</td>
</tr>
<tr>
<td>Type E12NP</td>
<td>VGATHERPF0DPD, VGATHERPF0DPD, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS</td>
</tr>
</tbody>
</table>

### NOTES:
1. Operand encoding Full tuple type with immediate.
2. Embedded broadcast is not supported with the “.nb” suffix.
3. Operand encoding Mem128 tuple type.
4. #UD raised if EVEX.L’L != 00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.
## 2.8.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

EVEX-encoded instructions with memory alignment restrictions, and supporting memory fault suppression follow exception class E1.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L'L restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>EVEX.512: Memory operand is not 64-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EVEX.256: Memory operand is not 32-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EVEX.128: Memory operand is not 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
</tbody>
</table>
EVEX-encoded instructions with memory alignment restrictions, but do not support memory fault suppression follow exception class E1NF.

### Table 2-45. Type E1NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L'L restriction not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>EVEX.512: Memory operand is not 64-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EVEX.256: Memory operand is not 32-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EVEX.128: Memory operand is not 16-byte aligned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X For an illegal memory operand effective address in the CS, DS, ES, FS or GS seg-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
### 2.8.2 Exceptions Type E2 of EVEX-Encoded Instructions

EVEX-encoded vector instructions with arithmetic semantic follow exception class E2.

#### Table 2-46. Type E2 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCP[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCP[bit 10] = 1.</td>
</tr>
</tbody>
</table>
2.8.3 **Exceptions Type E3 and E3NF of EVEX-Encoded Instructions**

EVEX-encoded scalar instructions with arithmetic semantic that support memory fault suppression follow exception class E3.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(O)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
EVEX-encoded scalar instructions with arithmetic semantic that do not support memory fault suppression follow exception class E3NF.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (FOH).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede an EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.</td>
</tr>
</tbody>
</table>
2.8.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

EVEX-encoded vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E4.

### Table 2-49. Type E4 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X X</td>
<td>X X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41 and in E4.nb subclass (see E4.nb</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>entries in Table 2-43).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X X</td>
<td>X X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X X</td>
<td>X X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E4NF.

### Table 2-50. Type E4NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-43).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
### 2.8.5 Exceptions Type E5 and E5NF

EVEX-encoded scalar/partial-vector instructions that cause no SIMD FP exception and support memory fault suppression follow exception class E5.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>- State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>- Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>- Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>- Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>- EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>- Instruction specific EVEX.L’ restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

EVEX-encoded scalar/partial vector instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E5NF.
### Table 2-52. Type E5NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X X X X X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE(bit 18)=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X X X X X</td>
<td>If preceded by a LOCK prefix (FOH).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X X X X X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
### 2.8.6 Exceptions Type E6 and E6NF

#### Table 2-53. Type E6 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility 64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L'L restriction not met.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X X If preceded by a LOCK prefix (F0H).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X X If any corresponding CPUID feature flag is ‘0’.</td>
<td></td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
<td></td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
<td></td>
</tr>
</tbody>
</table>
EVEX-encoded instructions that do not cause SIMD FP exception nor support memory fault suppression follow exception class E6NF.

### Table 2-54. Type E6NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
</tbody>
</table>
|                                  |      |               |                             |        | If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies:  
|                                  |      |               |                             |        | • State requirement, Table 2-37 not met.  
|                                  |      |               |                             |        | • Opcode independent #UD condition in Table 2-38.  
|                                  |      |               |                             |        | • Operand encoding #UD conditions in Table 2-39.  
|                                  |      |               |                             |        | • Opmask encoding #UD condition of Table 2-40.  
|                                  |      |               |                             |        | • EVEX.b encoding #UD condition of Table 2-41.  
|                                  |      |               |                             |        | • Instruction specific EVEX.L’L restriction not met.                                                                                               |
| Device Not Available, #NM        | X    | X             |                             |        | If preceded by a LOCK prefix (F0H).                                                                                                               |
| Stack, #SS(0)                    | X    |               |                             |        | If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.                                                                                           |
|                                  |      |               |                             |        | If any corresponding CPUID feature flag is ‘0’.                                                                                                   |
| General Protection, #GP(0)       | X    |               |                             |        | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.                                                          |
|                                  |      |               |                             |        | If the memory address is in a non-canonical form.                                                                                                 |
| Page Fault #PF(fault-code)       | X    | X             |                             |        | For a page fault.                                                                                                                               |
| Alignment Check #AC(0)           | X    | X             |                             |        | For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.       |
2.8.7 Exceptions Type E7NM

EVEX-encoded instructions that cause no SIMD FP exception and do not reference memory follow exception class E7NM.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>• Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If preceded by a LOCK prefix (FOH).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
</tbody>
</table>
### 2.8.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

#### Table 2-56. Type E9 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If fault suppression not set, and a memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded vector or partial-vector instructions that must be encoded with VEX.L’L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E9NF.

Table 2-57. Type E9NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility 64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td>If an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td>X</td>
<td>If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
### 2.8.9 Exceptions Type E10 and E10NF

EVEX-encoded scalar instructions that ignore EVEX.L'L vector length encoding, do not cause a SIMD FP exception, and support memory fault suppression follow exception class E10.

#### Table 2-58. Type E10 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (FOH).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If CR0. TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and a memory address referencing the SS segment is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>If fault suppression not set, and any part of the operand lies outside the effective</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
EVEX-encoded scalar instructions that ignore EVEX.L’L vector length encoding, do not cause a SIMD FP exception, and do not support memory fault suppression follow exception class E10NF.

### Table 2-59. Type E10NF Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is '0'.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(O)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td>General Protection, #GP(O)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Alignment Check #AC(O)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
2.8.10 Exceptions Type E11 (EVEX-only, Mem Arg, No AC, Floating-point Exceptions)

EVEX-encoded instructions that can cause SIMD FP exception, memory operand support fault suppression but do not cause #AC follow exception class E11.

Table 2-60. Type E11 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Instruction specific EVEX.L'L restriction not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If fault suppression not set, and an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If fault suppression not set, and a page fault.</td>
</tr>
</tbody>
</table>
### 2.8.11 Exceptions Type E12 and E12NP (VSIB Mem Arg, No AC, No Floating-point Exceptions)

#### Table 2-61. Type E12 Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Opcode mask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ Instruction specific EVEX.L’L restriction not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>▪ If vvvv != 1111b.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X NA If address size attribute is 16 bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If ModR/M.mod = ’11b’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If ModR/M.rm != ’100b’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If k0 is used (gather or scatter operation).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If index = destination register (gather operation).</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>Page Fault #PF (fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault.</td>
</tr>
</tbody>
</table>
EVEX-encoded prefetch instructions that do not cause #PF follow exception class E12NP.

### Table 2-62. Type E12NP Class Exception Conditions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If EVEX prefix present.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Opmask encoding #UD condition of Table 2-40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• EVEX.b encoding #UD condition of Table 2-41.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>• Instruction specific EVEX.L‘L restriction not met.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If preceded by a LOCK prefix (F0H).</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>NA</td>
<td></td>
<td>If address size attribute is 16 bit.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If ModR/M.mod = ‘11b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If ModR/M.rm != ‘100b’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If any corresponding CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If k0 is used (gather or scatter operation).</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
</tbody>
</table>
2.9 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS, TYPE K20 AND TYPE K21

The exception behavior of VEX-encoded opmask instructions are listed below.

2.9.1 Exceptions Type K20

Exception conditions of Opmask instructions that do not address memory are listed as Type K20.

Table 2-63. TYPE K20 Exception Definition (VEX-Encodced OpMask Instructions w/o Memory Arg)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If relevant CPUID feature flag is '0'.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If a VEX prefix is present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CR4.OSXSAVE[bit 18]=0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any one of following conditions applies:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• State requirement, Table 2-37 not met.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Opcode independent #UD condition in Table 2-38.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Operand encoding #UD conditions in Table 2-39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
</tbody>
</table>
### 2.9.2 Exceptions Type K21

Exception conditions of Opmask instructions that address memory are listed as Type K21.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual 80x86</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If relevant CPUID feature flag is ‘0’.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If a VEX prefix is present.</td>
</tr>
<tr>
<td>Device Not Available, #NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>If any REX, F2, F3, or 66 prefixes precede a VEX prefix.</td>
</tr>
<tr>
<td>Stack, #SS(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form.</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>Page Fault #PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For a page fault.</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>For 2, 4, or 8 byte memory access if alignment checking is enabled and an unaligned memory access is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>
2.10 INTEL® AMX INSTRUCTION EXCEPTION CLASSES

Alignment exceptions: The Intel AMX instructions that access memory will never generate #AC exceptions.

Table 2-65. Intel® AMX Exception Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
</table>
| AMX-E1 | • #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.  
  • #UD if CR4.OSXSAVE ≠ 1.  
  • #UD if XCR0[18:17] ≠ 0b11.  
  • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1.  
  • #UD if VVVV ≠ 0b1111.  
  • #GP based on palette and configuration checks (see pseudocode).  
  • #GP if the memory address is in a non-canonical form.  
  • #SS(0) if the memory address referencing the SS segment is in a non-canonical form.  
  • #PF if a page fault occurs. |
| AMX-E2 | • #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.  
  • #UD if CR4.OSXSAVE ≠ 1.  
  • #UD if XCR0[18:17] ≠ 0b11.  
  • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1.  
  • #UD if VVVV ≠ 0b1111.  
  • #GP if the memory address is in a non-canonical form.  
  • #SS(0) if the memory address referencing the SS segment is in a non-canonical form.  
  • #PF if a page fault occurs. |
| AMX-E3 | • #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.  
  • #UD if CR4.OSXSAVE ≠ 1.  
  • #UD if XCR0[18:17] ≠ 0b11.  
  • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1.  
  • #UD if VVVV ≠ 0b1111.  
  • #UD if not using SIB addressing.  
  • #UD if TILES_CONFIGURED == 0.  
  • #UD if tsrctest are not valid tiles.  
  • #UD if tsrctest are ≥ palette_table[tilecfg.palette_id].max_names.  
  • #UD if tsrctest.colbytes mod 4 ≠ 0 OR tdest.colbytes mod 4 ≠ 0.  
  • #UD if tilecfg.start_row ≥ tsrctest.rows OR tilecfg.start_row ≥ tdest.rows.  
  • #GP if the memory address is in a non-canonical form.  
  • #SS(0) if the memory address referencing the SS segment is in a non-canonical form.  
  • #PF if any memory operand causes a page fault.  
  • #NM if XFD[18] == 1. |
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
</table>
| **AMX-E4** | • #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.  
• #UD if CR4.OSXSAVE ≠ 1.  
• #UD if XCR0[18:17] ≠ 0b11.  
• #UD if IA32_EFER.LMA ≠ 1 OR CS:L ≠ 1.  
• #UD if srcdest == src1 OR src1 == src2 OR srcdest == src2.  
• #UD if TILES_CONFIGURED == 0.  
• #UD if srcdest.colbytes mod 4 ≠ 0.  
• #UD if src1.colbytes mod 4 ≠ 0.  
• #UD if src2.colbytes mod 4 ≠ 0.  
• #UD if srcdest/src1/src2 are not valid tiles.  
• #UD if srcdest/src1/src2 are ≥ palette_table[tilecfgpalette_id].max_names.  
• #UD if srcdest.colbytes ≠ src2.colbytes.  
• #UD if srcdest.rows ≠ src1.rows.  
• #UD if src1.colbytes / 4 ≠ src2.rows.  
• #UD if srcdest.colbytes > tmul_maxn.  
• #UD if src2.colbytes > tmul_maxn.  
• #UD if src1.colbytes/4 > tmul_maxk.  
• #UD if src2.rows > tmul_maxk.  
• #NM if XFD[18] == 1. |
| **AMX-E5** | • #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.  
• #UD if CR4.OSXSAVE ≠ 1.  
• #UD if XCR0[18:17] ≠ 0b11.  
• #UD if IA32_EFER.LMA ≠ 1 OR CS:L ≠ 1.  
• #UD if VVVV ≠ 0b1111.  
• #UD if TILES_CONFIGURED == 0.  
• #UD if tdest is not a valid tile.  
• #UD if tdest is ≥ palette_table[tilecfgpalette_id].max_names.  
• #NM if XFD[18] == 1. |
| **AMX-E6** | • #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes.  
• #UD if CR4.OSXSAVE ≠ 1.  
• #UD if XCR0[18:17] ≠ 0b11.  
• #UD if IA32_EFER.LMA ≠ 1 OR CS:L ≠ 1.  
• #UD if VVVV ≠ 0b1111. |
CHAPTER 3
INSTRUCTION SET REFERENCE, A-L


For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

3.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The heading below introduces the example. The table below provides an example summary table.

CMC—Complement Carry Flag [this is an example]

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>CMC</td>
<td>ZO</td>
<td>V/V</td>
<td>N/A</td>
<td>Complement carry flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX Prefix)

The “Opcode” column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.

- **NFx** — Indicates the use of F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.

- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.

- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

- **/r** — Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.

- **cb, cw, cd, cp, co, ct** — A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.

- **ib, iw, id, io** — A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords, and quadwords are given with the low-order byte first.

- **+rb, +rw, +rd, +ro** — Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. “+ro” is applicable only in 64-bit mode. See Table 3-1 for the codes.

- **+i** — A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

### Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

<table>
<thead>
<tr>
<th>byte register</th>
<th>word register</th>
<th>dword register</th>
<th>quadword register (64-Bit Mode only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>REX.B</td>
<td>Reg Field</td>
<td>Register</td>
</tr>
<tr>
<td>AL</td>
<td>None</td>
<td>0</td>
<td>AX</td>
</tr>
<tr>
<td>CL</td>
<td>None</td>
<td>1</td>
<td>CX</td>
</tr>
<tr>
<td>DL</td>
<td>None</td>
<td>2</td>
<td>DX</td>
</tr>
<tr>
<td>BL</td>
<td>None</td>
<td>3</td>
<td>BX</td>
</tr>
<tr>
<td>AH</td>
<td>Not encodable (N.E.)</td>
<td>4</td>
<td>SP</td>
</tr>
<tr>
<td>CH</td>
<td>N.E.</td>
<td>5</td>
<td>BP</td>
</tr>
<tr>
<td>DH</td>
<td>N.E.</td>
<td>6</td>
<td>SI</td>
</tr>
<tr>
<td>BH</td>
<td>N.E.</td>
<td>7</td>
<td>DI</td>
</tr>
<tr>
<td>SPL</td>
<td>Yes</td>
<td>4</td>
<td>SP</td>
</tr>
<tr>
<td>BPL</td>
<td>Yes</td>
<td>5</td>
<td>BP</td>
</tr>
</tbody>
</table>
3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix)

In the Instruction Summary Table, the Opcode column presents each instruction encoded using the VEX prefix in following form (including the modR/M byte if applicable, the immediate byte if applicable):

```
VEX.[128,256].[66,F2,F3].OF/0F3A/0F38.[W0,W1] opcode [/r] [/ib,/is4]
```

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

The encoding of various sub-fields of the VEX prefix is described using the following notations:

- **128,256**: VEX.L.field can be 0 (denoted by VEX.128, VEX.L0, or VEX.LZ) or 1 (denoted by VEX.256 or VEX.L1). The VEX.L field can be encoded using either the 2-byte or 3-byte form of the VEX prefix. The presence of the notation VEX.256 or VEX.128 in the opcode column should be interpreted as follows:
  - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
  - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g., VMOVLP).
  - If VEX.L0 or VEX.L1 is present in the opcode column: The specified VEX.L value is required for encoding this instruction but does not have the connotation of specifying vector length.
  - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distinguished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either “SS”, “SD”, or “SI” for SIMD floating-point conversion instructions.
  - If VEX.LZ is present in the opcode column: The VEX.L must be encoded to be 0B, an #UD occurs if VEX.L is not zero.
— **66,F2,F3:** The presence or absence of these values map to the VEX.pp field encodings. If absent, this corresponds to VEX.pp=00B. If present, the corresponding VEX.pp value affects the “opcode” byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix. The VEX.pp field may be encoded using either the 2-byte or 3-byte form of the VEX prefix.

— **0F,0F3A,0F38:** The presence maps to a valid encoding of the VEX.mmmmmm field. Only three encoded values of VEX.mmmmmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH, and 0F38H. The effect of a valid VEX.mmmmmm encoding on the ensuing opcode byte is same as if the corresponding escape byte sequence on the ensuing opcode byte for non-VEX encoded instructions. Thus a valid encoding of VEX.mmmmmm may be consider as an implies escape byte sequence of either 0FH, 0F3AH or 0F38H. The VEX.mmmmmm field must be encoded using the 3-byte form of VEX prefix.

— **0F,0F3A,0F38 and 2-byte/3-byte VEX:** The presence of 0F3A and 0F38 in the opcode column implies that opcode can only be encoded by the three-byte form of VEX. The presence of 0F in the opcode column does not preclude the opcode to be encoded by the two-byte of VEX if the semantics of the opcode does not require any subfield of VEX not present in the two-byte form of the VEX prefix.

— **W0:** VEX.W=0.
— **W1:** VEX.W=1.

The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the CSH form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.

— **WIG:** can use C5H form (if not requiring VEX.mmmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.

If WIG is present, the instruction may be encoded using either the two-byte form or the three-byte form of VEX. When encoding the instruction using the three-byte form of VEX, the value of VEX.W is ignored.

- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in either imm8[7:4] (for 64-bit mode) or imm8[6:4] (for 32-bit mode), and instruction-specific payload in imm8[3:0].

In general, the encoding of VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

**EVEX.[128,256,512,LLIG].[66,F2,F3].0F/0F3A/0F38.[W0,W1,WIG] opcode [/r] [ib]**

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 2.7.1 for more detail on the EVEX prefix.

The encoding of various sub-fields of the EVEX prefix is described using the following notations:

— **128, 256, 512, LLIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.

— **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the “opcode” byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.

— **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH, and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.

— **W0:** EVEX.W=0.
- **W1**: EVEX.W=1.
- **WIG**: EVEX.W bit ignored

**opcode** — Instruction opcode.

In general, the encoding of EVEX.R and R', EVEX.X and X', and EVEX.B and B' fields are not shown explicitly in the opcode column.

**NOTE**

Previously, the terms NDS, NDD, and DDS were used in instructions with an EVEX (or VEX) prefix. These terms indicated that the vvvv field was valid for encoding, and specified register usage. These terms are no longer necessary and are redundant with the instruction operand encoding tables provided with each instruction. The instruction operand encoding tables give explicit details on all operands, indicating where every operand is stored and if they are read or written. If vvvv is not listed as an operand in the instruction operand encoding table, then EVEX (or VEX) vvvv must be 0b1111.

### 3.1.1.3 Instruction Column in the Opcode Summary Table

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

- **rel16, rel32** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.

- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation 16:16 indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.

- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL, and SIL; or one of the byte registers (R8B - R15B) available using REX.R and 64-bit mode.

- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.

- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.

- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15. These are available when using REX.R and 64-bit mode.

- **imm8** — An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.

- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.

- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.

- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL, and SIL) or a byte from memory. Byte registers R8B - R15B are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.

- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.

- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

- **reg** — A general-purpose register used for instructions when the width of the register does not matter to the semantics of the operation of the instruction. The register can be r16, r32, or r64.

- **m** — A 16-, 32- or 64-bit operand in memory.

- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.

- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.

- **m32** — A doubleword operand in memory. The contents of memory are found at the address provided by the effective address computation.

- **m64** — A memory quadword operand in memory.

- **m128** — A memory doublequadword operand in memory.

- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&32 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.

- **m80bcd** — A Binary Coded Decimal (BCD) operand in memory, 80 bits.

- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.

- **m32fp, m64fp, m80fp** — A single precision, double precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.

- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.

- **ST or ST(0)** — The top element of the FPU register stack.

- **ST(i)** — The \( i \)th element from the top of the FPU register stack (\( i := 0 \) through 7).

- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.

- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
• **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

• **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.

• **xmm/m32** — An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

• **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

• **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

• **<XMM0>** — Indicates implied use of the XMM0 register.

  When there is ambiguity, xmm1 indicates the first source operand using an XMM register and xmm2 the second source operand using an XMM register.

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.

• **ymm** — A YMM register. The 256-bit YMM registers are: YMM0 through YMM7; YMM8 through YMM15 are available in 64-bit mode.

• **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX instructions.

• **ymm/m256** — A YMM register or 256-bit memory operand.

• **<YMM0>** — Indicates use of the YMM0 register as an implicit argument.

• **bnd** — A 128-bit bounds register. BND0 through BND3.

• **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.

• **m512** — A 64-byte operand in memory.

• **zmm/m512** — A ZMM register or 512-bit memory operand.

• **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.

• **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.

• **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.

• **mV** — A vector memory operand; the operand size is dependent on the instruction.

• **vm32{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).

• **vm64{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).

• **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.

• **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.
• `<ZMM0>` — Indicates use of the ZMM0 register as an implicit argument.
• `{er}` — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
• `{sae}` — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
• `SRC1` — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
• `SRC2` — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
• `SRC3` — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
• `SRC` — The source in a single-source instruction.
• `DST` — The destination in an instruction. This field is encoded by reg_field.

In the instruction encoding, the MODRM byte is represented several ways depending on the role it plays. The MODRM byte has 3 fields: 2-bit MODRM.MOD field, a 3-bit MODRM.REG field and a 3-bit MODRM.RM field. When all bits of the MODRM byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the MODRM byte must contain fixed values, those values are specified as follows:
• If only the MODRM.MOD must be 0b11, and MODRM.REG and MODRM.RM fields are unrestricted, this is denoted as `11:rrr:bbb`. The `rrr` correspond to the 3-bits of the MODRM.REG field and the `bbb` correspond to the 3-bits of the MODRM.RM field.
• If the MODRM.MOD field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then we use the notation !(11).
• If the MODRM.REG field had a specific required value, e.g., 0b101, that would be denoted as `mm:101:bbb`.

3.1.1.4  Operand Encoding Column in the Instruction Summary Table

The "operand encoding" column is abbreviated as Op/En in the Instruction Summary table heading. Instruction operand encoding information is provided for each assembly instruction syntax using a letter to cross reference to a row entry in the operand encoding definition table that follows the instruction summary table. The operand encoding table in each instruction reference page lists each instruction operand (according to each instruction syntax and operand ordering shown in the instruction column) relative to the ModRM byte, VEX.vvvv field or additional operand encoding placement.

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-34 and Table 2-35, according to tupletypes. The tupletype for an instruction is listed in the operand encoding definition table where applicable.

NOTES
• The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.
• In the encoding definition table, the letter 'r' within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter 'w' within a pair of parenthesis denotes the content of the operand will be updated by the processor.

3.1.1.5  64/32-bit Mode Column in the Instruction Summary Table

The "64/32-bit Mode" column indicates whether the opcode sequence is supported in (a) 64-bit mode or (b) the Compatibility mode and other IA-32 modes that apply in conjunction with the CPUID feature flag associated specific instruction extensions.

The 64-bit mode support is to the left of the ‘slash’ and has the following notation:
• `V` — Supported.
• `I` — Not supported.
• **N.E.** — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence of valid instructions in other modes).
• **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
• **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
• **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.

The Compatibility/Legacy Mode support is to the right of the 'slash' and has the following notation:
• **V** — Supported.
• **I** — Not supported.
• **N.E.** — Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

### 3.1.1.6 CPUID Support Column in the Instruction Summary Table

The fourth column holds abbreviated CPUID feature flags (e.g., appropriate bit in CPUID.01H.ECX, CPUID.01H.EDX for SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2/AESNI/PCLMULQDQ/AVX/RDRAND support) that indicate processor support for the instruction. If the corresponding flag is ‘0’, the instruction will #UD.

### 3.1.1.7 Description Column in the Instruction Summary Table

The "Description" column briefly explains forms of the instruction.

### 3.1.1.8 Description Section

Each instruction is then described by number of information sections. The "Description" section describes the purpose of the instructions and required operands in more detail.

Summary of terms that may be used in the description section:
• **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ, and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
• **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1’s complement form).
• **rm_field** — shorthand for the ModR/M r/m field and any REX.B
• **reg_field** — shorthand for the ModR/M reg field and any REX.R

### 3.1.1.9 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:
• Comments are enclosed within the symbol pairs “(*” and “*)”.
• Compound statements are enclosed in keywords, such as: IF, THEN, ELSE, and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
• A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register’s default segment (DS) or the overridden segment.
• Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, \([SRC]\) indicates that the content of the source operand is a segment-relative offset.

A \(\Rightarrow B\) indicates that the value of \(B\) is assigned to \(A\).

The symbols =, \(\neq\), >, <, \(\geq\), and \(\leq\) are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as \(A = B\) is TRUE if the value of \(A\) is equal to \(B\); otherwise it is FALSE.

The expression "\(\ll C\) COUNT" and "\(\gg C\) COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize** and **AddressSize** — The **OperandSize** identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The **AddressSize** identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```
IF Instruction = MOVW
    THEN OperandSize := 16;
ELSE
    IF Instruction = MOVD
        THEN OperandSize := 32;
    ELSE
        IF Instruction = MOVQ
            THEN OperandSize := 64;
    FI;
FI;
```

See "Operand-Size and Address-Size Attributes" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for guidelines on how these attributes are determined.

- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

- **SRC** — Represents the source operand.

- **DEST** — Represents the destination operand.

- **MAXVL** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's encoding but is instead determined by the current value of XCR0. For details, refer to the table below. Note that the value of MAXVL is the largest of the features enabled. Future processors may define new bits in XCR0 whose setting may imply other values for MAXVL.

<table>
<thead>
<tr>
<th>XCR0 Component</th>
<th>MAXVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCR0.SSE</td>
<td>128</td>
</tr>
<tr>
<td>XCR0.AVX</td>
<td>256</td>
</tr>
<tr>
<td>XCR0.{ZMM_Hi256, Hi16_ZMM, OPMASK}</td>
<td>512</td>
</tr>
</tbody>
</table>

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of \(-10\) converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value \(-10\) converts the byte...
from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.

- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than -128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).

- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than -32768, it is represented by the saturated value -32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).

- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).

- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).

- **SaturateToUnsignedByte** — Represents the result of an operation as an unsigned 8-bit value. If the result is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).

- **SaturateToUnsignedWord** — Represents the result of an operation as an unsigned 16-bit value. If the result is less than zero, it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).

- **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.

- **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.

- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword, or Quadword Onto the Stack" section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

- **Pop()** — removes the value from the top of the stack and returns it. The statement EAX := Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value From the Stack" section of Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.

- **Switch-Tasks** — Performs a task switch.

- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

![Figure 3-1. Bit Offset for BIT[RAX, 21]](image-url)
If BitBase is a memory address, the BitOffset has different ranges depending on the operand size (see Table 3-2).

### Table 3-2. Range of Bit Positions Specified by Bit Offset Operands

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Immediate BitOffset</th>
<th>Register BitOffset</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0 to 15</td>
<td>$2^{15}$ to $2^{15} - 1$</td>
</tr>
<tr>
<td>32</td>
<td>0 to 31</td>
<td>$2^{31}$ to $2^{31} - 1$</td>
</tr>
<tr>
<td>64</td>
<td>0 to 63</td>
<td>$2^{63}$ to $2^{63} - 1$</td>
</tr>
</tbody>
</table>

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

---

**3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section**

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- **__m128, __m256, and __m512** can represent 4, 8, or 16 packed single precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128 data type is also used with various single precision floating-point scalar instructions that perform calculations using only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.

- **__m128d, __m256d, and __m512d** can represent 2, 4, or 8 packed double precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128d data type is also used with various double precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.

- **__m128i, __m256i, and __m512i** can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.
Each of these data types incorporates in its name the number of bits it can hold. For example, the __m128 type holds 128 bits, and because each single precision floating-point value is 32 bits long the __m128 type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- __mm_ indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- __mm256_ indicates the function operates on 256-bit vectors.
- __mm512_ indicates that the function operates on 512-bit vectors.

The suffixes include:

- _ps, which indicates a function that operates on packed single precision floating-point data. Packed single precision floating-point data corresponds to arrays of the C/C++ type float with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the __mm_loadu_ps, __mm256_loadu_ps, or __mm512_loadu_ps functions, or created from individual values using __mm_set_ps, __mm256_set_ps, or __mm512_set_ps functions, and they can be stored in an array using __mm_storeu_ps, __mm256_storeu_ps, or __mm512_storeu_ps.

- _ss, which indicates a function that operates on scalar single precision floating-point data. Single precision floating-point data corresponds to the C/C++ type float, and values of type float can be converted to type __m128 for use with these functions using the __mm_set_ss function, and converted back using the __mm_cvtsf_ss function. When used with functions that operate on packed single precision floating-point data the scalar element corresponds with the first packed value.

- _pd, which indicates a function that operates on packed double precision floating-point data. Packed double precision floating-point data corresponds to arrays of the C/C++ type double with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the __mm_loadu_pd, __mm256_loadu_pd, or __mm512_loadu_pd functions, or created from individual values using __mm_set_pd, __mm256_set_pd, or __mm512_set_pd functions, and they can be stored in an array using __mm_storeu_pd, __mm256_storeu_pd, or __mm512_storeu_pd.

- _sd, which indicates a function that operates on scalar double precision floating-point data. Double-precision floating-point data corresponds to the C/C++ type double, and values of type double can be converted to type __m128d for use with these functions using the __mm_set_sd function, and converted back using the __mm_cvtsd_sd function. When used with functions that operate on packed double precision floating-point data the scalar element corresponds with the first packed value.

- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of signed char with 16, 32 or 64 elements. Values of this type can be created from individual elements using __mm_set_epi8, __mm256_set_epi8, or __mm512_set_epi8 functions.

- _epi16, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of short with 8, 16 or 32 elements. Values of this type can be created from individual elements using __mm_set_epi16, __mm256_set_epi16, or __mm512_set_epi16 functions.

- _epi32, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of int with 4, 8 or 16 elements. Values of this type can be created from individual elements using __mm_set_epi32, __mm256_set_epi32, or __mm512_set_epi32 functions.

- _epi64, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of long long (or long if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using __mm_set_epi32, __mm256_set_epi32, or __mm512_set_epi32 functions.

- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of unsigned char with 16, 32 or 64 elements.
• \_epu16, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of unsigned short with 8, 16 or 32 elements.
• \_epu32, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of unsigned with 4, 8 or 16 elements.
• \_epu64, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of unsigned long long (or unsigned long if it is a 64-bit data type) with 2, 4 or 8 elements.
• \_si128, which indicates a function that operates on a single 128-bit value of type __m128i.
• \_si256, which indicates a function that operates on a single a 256-bit value of type __m256i.
• \_si512, which indicates a function that operates on a single a 512-bit value of type __m512i.

Values of any packed integer type can be loaded from an array using the \_mm_loadu_si128, \_mm256_loadu_si256, or \_mm512_loadu_si512 functions, and they can be stored in an array using \_mm_storeu_si128, \_mm256_storeu_si256, or \_mm512_storeu_si512.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as __int64. The C Standard header stdint.h defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

<table>
<thead>
<tr>
<th>Non-standard Type</th>
<th>Standard Type (from stdint.h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__int64</td>
<td>int64_t</td>
</tr>
<tr>
<td>unsigned __int64</td>
<td>uint64_t</td>
</tr>
<tr>
<td>__int32</td>
<td>int32_t</td>
</tr>
<tr>
<td>unsigned __int32</td>
<td>uint32_t</td>
</tr>
<tr>
<td>__int16</td>
<td>int16_t</td>
</tr>
<tr>
<td>unsigned __int16</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>


3.1.1.11 Flags Affected Section

The “Flags Affected” section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, “EFLAGS Cross-Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Non-conventional assignments are described in the “Operation” section. The values of flags listed as undefined may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an “FPU Flags Affected” section that describes how each instruction can affect the four condition code flags of the FPU status word.

3.1.1.13 Protected Mode Exceptions Section

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound
sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

### Table 3-4. Intel 64 and IA-32 General Exceptions

<table>
<thead>
<tr>
<th>Vector</th>
<th>Name</th>
<th>Source</th>
<th>Protected Mode 1</th>
<th>Real Address Mode</th>
<th>Virtual 8086 Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE—Divide Error</td>
<td>DIV and IDIV instructions.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>#DB—Debug</td>
<td>Any code or data reference.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>#BP—Breakpoint</td>
<td>INT3 instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>#OF—Overflow</td>
<td>INTO instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>#BR—BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>#UD—Invalid Opcode (Undefined Opcode)</td>
<td>UD instruction or reserved opcode.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>#NM—Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>#DF—Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>#TS—Invalid TSS</td>
<td>Task switch or TSS access.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>#NP—Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>#SS—Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>#GP—General Protection²</td>
<td>Any memory reference and other protection checks.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>#PF—Page Fault</td>
<td>Any memory reference.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>#MF—Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>#AC—Alignment Check</td>
<td>Any data reference in memory.</td>
<td>Yes</td>
<td>Reserved</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>#MC—Machine Check</td>
<td>Model dependent machine check errors.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>#XM—SIMD Floating-Point Numeric Error</td>
<td>SSE/SSE2/SSE3 floating-point instructions.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.

### 3.1.1.14 Real-Address Mode Exceptions Section

The “Real-Address Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

### 3.1.1.15 Virtual-8086 Mode Exceptions Section

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).
3.1.1.16 Floating-Point Exceptions Section

The “Floating-Point Exceptions” section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See “Floating-Point Exception Conditions” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a detailed description of these exceptions.

Table 3-5. x87 FPU Floating-Point Exceptions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>#IS</td>
<td>Floating-point invalid operation:</td>
<td>x87 FPU stack overflow or underflow</td>
</tr>
<tr>
<td>#IA</td>
<td>Floating-point invalid operation:</td>
<td>Invalid FPU arithmetic operation</td>
</tr>
<tr>
<td>#Z</td>
<td>Floating-point divide-by-zero</td>
<td>Divide-by-zero</td>
</tr>
<tr>
<td>#D</td>
<td>Floating-point denormal operand</td>
<td>Source operand that is a denormal number</td>
</tr>
<tr>
<td>#O</td>
<td>Floating-point numeric overflow</td>
<td>Overflow in result</td>
</tr>
<tr>
<td>#U</td>
<td>Floating-point numeric underflow</td>
<td>Underflow in result</td>
</tr>
<tr>
<td>#P</td>
<td>Floating-point inexact result (precision)</td>
<td>Inexact result (precision)</td>
</tr>
</tbody>
</table>

3.1.1.17 SIMD Floating-Point Exceptions Section

The “SIMD Floating-Point Exceptions” section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to “SSE and SSE2 Exceptions”, in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Table 3-6. SIMD Floating-Point Exceptions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>#I</td>
<td>Floating-point invalid operation</td>
<td>Invalid arithmetic operation or source operand</td>
</tr>
<tr>
<td>#Z</td>
<td>Floating-point divide-by-zero</td>
<td>Divide-by-zero</td>
</tr>
<tr>
<td>#D</td>
<td>Floating-point denormal operand</td>
<td>Source operand that is a denormal number</td>
</tr>
<tr>
<td>#O</td>
<td>Floating-point numeric overflow</td>
<td>Overflow in result</td>
</tr>
<tr>
<td>#U</td>
<td>Floating-point numeric underflow</td>
<td>Underflow in result</td>
</tr>
<tr>
<td>#P</td>
<td>Floating-point inexact result</td>
<td>Inexact result (precision)</td>
</tr>
</tbody>
</table>

3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.

3.2 INTEL® AMX CONSIDERATIONS

The following implementation parameters and helper functions are applicable to the Intel® AMX instructions.
3.2.1 Implementation Parameters

The parameters are reported via CPUID leaf 1DH. Index 0 reports all zeros for all fields.

```c
#define palette_table[id]:
    uint16_t total_tile_bytes
    uint16_t bytes_per_tile
    uint16_t bytes_per_row
    uint16_t max_names
    uint16_t max_rows
```

The tile parameters are set by LDTILECFG or XRSTOR* of TILECFG:

```c
#define tile[tid]:
    byte rows
    word colsb // bytes_per_row
    bool valid
```

3.2.2 Helper Functions

The helper functions used in Intel AMX instructions are defined below.

```c
define write_row_and_zero(treg, r, data, nbytes):
    for j in 0 ... nbytes-1:
        treg.row[r].byte[j] := data.byte[j]

    // zero the rest of the row
    for j in nbytes ... palette_table[tilecfg.palette_id].bytes_per_row-1:
        treg.row[r].byte[j] := 0

define zero_upper_rows(treg, r):
    for i in r ... palette_table[tilecfg.palette_id].max_rows-1:
        for j in 0 ... palette_table[tilecfg.palette_id].bytes_per_row-1:
            treg.row[i].byte[j] := 0

define zero_tilecfg_start():
    tilecfg.start_row := 0

define zero_all_tile_data():
    if XCR0[TILEDATA]:
        b := CPUID(0xD, TILEDATA).EAX // size of feature
        for j in 0 ... b:
            TILEDATA.byte[j] := 0
```
3.3 INSTRUCTIONS (A-L)

AAA—ASCII Adjust After Addition

Description
Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AX := AX + 106H;
                AF := 1;
                CF := 1;
            ELSE
                AF := 0;
                CF := 0;
            FI;
        FI;
    FI;

Flags Affected
The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as protected mode.
Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
AAD—ASCII Adjust AX Before Division

### Description
Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to \((AL + (10 * AH))\), and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the “Operation” section below), by setting the \(imm8\) byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 \(imm8\)).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation
IF 64-Bit Mode
THEN
  #UD;
ELSE
  tempAL := AL;
  tempAH := AH;
  AL := (tempAL + (tempAH + \(imm8\))) AND FFH;  (* \(imm8\) is set to 0AH for the AAD mnemonic.*)
  AH := 0;
FI;

The immediate value \((imm8)\) is taken from the second byte of the instruction.

### Flags Affected
The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

### Protected Mode Exceptions
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions
Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions
Same exceptions as protected mode.
Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
AAM—ASCII Adjust AX After Multiply

Description
Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the imm8 byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 imm8).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL := AL;
        AH := tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
        AL := tempAL MOD imm8;
    FI;

The immediate value (imm8) is taken from the second byte of the instruction.

Flags Affected
The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

Protected Mode Exceptions
#DE If an immediate value of 0 is used.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as protected mode.
Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
AAS—ASCII Adjust AL After Subtraction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F</td>
<td>AAS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>ASCII adjust AL after subtraction.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-bit mode
THEN
   #UD;
ELSE
   IF ((AL AND 0FH) > 9) or (AF = 1)
   THEN
      AX := AX - 6;
      AH := AH - 1;
      AF := 1;
      CF := 1;
      AL := AL AND 0FH;
   ELSE
      CF := 0;
      AF := 0;
      AL := AL AND 0FH;
   FI;
FI;

### Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.
Compatibility Mode Exceptions
Same exceptions as protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
ADC—Add With Carry

**Opcode**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 ib</td>
<td>ADC AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm8 to AL.</td>
</tr>
<tr>
<td>15 iw</td>
<td>ADC AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm16 to AX.</td>
</tr>
<tr>
<td>15 id</td>
<td>ADC EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm32 to EAX.</td>
</tr>
<tr>
<td>REX.W + 15 id</td>
<td>ADC RAX, imm32</td>
<td>I</td>
<td>N.E.</td>
<td></td>
<td>Add with carry imm32 sign extended to 64-bits to RAX.</td>
</tr>
<tr>
<td>80 /2 lb</td>
<td>ADC r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm8 to r/m8.</td>
</tr>
<tr>
<td>REX + 80 /2 lb</td>
<td>ADC r/m8 , imm8</td>
<td>MI</td>
<td>N.E.</td>
<td></td>
<td>Add with carry imm8 to r/m8.</td>
</tr>
<tr>
<td>81 /2 iw</td>
<td>ADC r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm16 to r/m16.</td>
</tr>
<tr>
<td>81 /2 id</td>
<td>ADC r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF imm32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /2 id</td>
<td>ADC r/m64, imm32</td>
<td>MI</td>
<td>N.E.</td>
<td></td>
<td>Add with CF imm32 sign extended to 64-bits to r/m64.</td>
</tr>
<tr>
<td>83 /2 lb</td>
<td>ADC r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF sign-extended imm8 to r/m16.</td>
</tr>
<tr>
<td>83 /2 id</td>
<td>ADC r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF sign-extended imm8 into r/m32.</td>
</tr>
<tr>
<td>REX.W + 83 /2 id</td>
<td>ADC r/m64, imm32</td>
<td>MI</td>
<td>N.E.</td>
<td></td>
<td>Add with CF sign-extended imm8 into r/m64.</td>
</tr>
<tr>
<td>10 /r</td>
<td>ADC r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry byte register to r/m8.</td>
</tr>
<tr>
<td>REX + 10 /r</td>
<td>ADC r/m8 , r8</td>
<td>MR</td>
<td>N.E.</td>
<td></td>
<td>Add with carry byte register to r/m8.</td>
</tr>
<tr>
<td>11 /r</td>
<td>ADC r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry r16 to r/m16.</td>
</tr>
<tr>
<td>11 /r</td>
<td>ADC r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF r32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 11 /r</td>
<td>ADC r/m64, r/m8</td>
<td>MR</td>
<td>N.E.</td>
<td></td>
<td>Add with CF r64 to r/m64.</td>
</tr>
<tr>
<td>12 /r</td>
<td>ADC r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry r/m8 to byte register.</td>
</tr>
<tr>
<td>REX + 12 /r</td>
<td>ADC r8 , r/m8</td>
<td>RM</td>
<td>N.E.</td>
<td></td>
<td>Add with carry r/m8 to byte register.</td>
</tr>
<tr>
<td>13 /r</td>
<td>ADC r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry r/m16 to r16.</td>
</tr>
<tr>
<td>13 /r</td>
<td>ADC r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF r/m32 to r32.</td>
</tr>
<tr>
<td>REX.W + 13 /r</td>
<td>ADC r64, r/m64</td>
<td>RM</td>
<td>N.E.</td>
<td></td>
<td>Add with CF r/m64 to r64.</td>
</tr>
</tbody>
</table>

**NOTES:**

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.
The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST := DEST + SRC + CF;

**Intel C/C++ Compiler Intrinsic Equivalent**

ADC extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);
ADC extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);
ADC extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
ADC extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

**Flags Affected**

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

**Protected Mode Exceptions**

- #GP(0) If the destination is located in a non-writable segment.
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
ADCX—Unsigned Integer Addition of Two Operands With Carry Flag

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

### Operation

IF OperandSize is 64-bit
FI;

### Flags Affected

CF is updated based on result. OF, SF, ZF, AF, and PF flags are unmodified.

### Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

### SIMD Floating-Point Exceptions

None.
Protected Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) For an illegal address in the SS segment.
#GP(0) If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
ADD—Add

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (r, w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST := DEST + SRC;

**Flags Affected**

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

**Protected Mode Exceptions**

- **#GP(0)** If the destination is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
### ADDPD—Add Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66H 0F 58 /r ADDPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed double precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66H.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed double precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66H.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed double precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66H.0F.W1 58 /r VADDPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66H.0F.W1 58 /r VADDPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66H.0F.W1 58 /r VADDPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Adds two, four or eight packed double precision floating-point values from the first source operand to the second source operand, and stores the packed double precision floating-point result in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Vector Register

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

IF \((VL = 512)\) AND \((EVEX.b = 1)\)
THEN
  \[\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION}(\text{EVEX}.RC);\]
ELSE
  \[\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION}(\text{MXCSR}.RC);\]
FI;
FOR \(j := 0\) TO \(KL-1\)
\(i := j \times 64\)
IF \(k1[j]\) OR \(*\text{no writemask}\)*
THEN \(\text{DEST}[i+63:i] := \text{SRC1}[i+63:i] + \text{SRC2}[i+63:i]\)
ELSE
  IF \(*\text{merging-masking}\)*
  THEN \(*\text{DEST}[i+63:i] \text{ remains unchanged}\)*
  ELSE \(*\text{zeroing-masking}\)*
  \(\text{DEST}[i+63:i] := 0\)
FI
ENDFOR
\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

VADDPD (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0\) TO \(KL-1\)
\(i := j \times 64\)
IF \(k1[j]\) OR \(*\text{no writemask}\)*
THEN
  IF \((EVEX.b = 1)\)
  THEN
    \(\text{DEST}[i+63:i] := \text{SRC1}[i+63:i] + \text{SRC2}[63:0]\)
  ELSE
    \(\text{DEST}[i+63:i] := \text{SRC1}[i+63:i] + \text{SRC2}[i+63:i]\)
  FI;
ELSE
  IF \(*\text{merging-masking}\)*
  THEN \(*\text{DEST}[i+63:i] \text{ remains unchanged}\)*
  ELSE \(*\text{zeroing-masking}\)*
  \(\text{DEST}[i+63:i] := 0\)
FI
ENDFOR
\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

VADDPD (VEX.256 Encoded Version)

\(\text{DEST}[63:0] := \text{SRC1}[63:0] + \text{SRC2}[63:0]\)
\(\text{DEST}[127:64] := \text{SRC1}[127:64] + \text{SRC2}[127:64]\)
\(\text{DEST}[255:192] := \text{SRC1}[255:192] + \text{SRC2}[255:192]\)
\(\text{DEST}[\text{MAXVL}-1:256] := 0\).
VADDPD (VEX.128 Encoded Version)
DEST[63:0] := SRC1[63:0] + SRC2[63:0]
DEST[MAXVL-1:128] := 0

ADDPD (128-bit Legacy SSE Version)
DEST[63:0] := DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VADDPD __m512d _mm512_add_pd (__m512d a, __m512d b);
VADDPD __m512d __m512d_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VADDPD __m512d __m512d_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);
VADDPD __m256d __m256d_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VADDPD __m256d __m256d_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);
VADDPD __m128d __m128d_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDPD __m128d __m128d_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);
VADDPD __m512d __m512d_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VADDPD __m512d __m512d_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VADDPD __m256d __m256d_mask_add_round_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int);
VADDPD __m256d __m256d_maskz_add_round_pd (__mmask8 k, __m256d a, __m256d b, int);
ADDPD __m128d __m128d_mask_add_round_pd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
ADDPD __m128d __m128d_maskz_add_round_pd (__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
ADDPS—Add Packed Single Precision Floating-Point Values

**Opcode/Instruction** | **Op / En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
NP 0F 58 /r | A | V/V | SSE | Add packed single precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.128.0F.WIG 58 /r | B | V/V | AVX | Adds four, eight or sixteen packed single precision floating-point values from the first source operand with the second source operand, and stores the packed single precision floating-point result in the destination operand.
VEX.256.0F.WIG 58 /r | B | V/V | AVX | Add packed single precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.128.0F.W0 58 /r | C | V/V | AVX512VL AVX512F | Add packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.256.0F.W0 58 /r | C | V/V | AVX512VL AVX512F | Add packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.512.0F.W0 58 /r | C | V/V | AVX512F | Add packed single precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Adds four, eight or sixteen packed single precision floating-point values from the first source operand with the second source operand, and stores the packed single precision floating-point result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
**Operation**

**VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

    i := j * 32

    IF k1[j] OR *no writemask*

        THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]

        ELSE

            IF *merging-masking* ; merging-masking

                THEN *DEST[i+31:i] remains unchanged*

            ELSE ; zeroing-masking

                DEST[i+31:i] := 0

            FI

        FI

ENDFOR;

DEST[MAXVL-1:VL] := 0

**VADDPS (EVEX Encoded Versions) When SRC2 Operand is a Memory Source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

    i := j * 32

    IF k1[j] OR *no writemask*

        THEN IF (EVEX.b = 1)

            THEN

                DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]

            ELSE

                DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]

            FI;

        ELSE

            IF *merging-masking* ; merging-masking

                THEN *DEST[i+31:i] remains unchanged*

            ELSE ; zeroing-masking

                DEST[i+31:i] := 0

            FI

        FI

ENDFOR;

DEST[MAXVL-1:VL] := 0
VADDPS (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[MAXVL-1:256] := 0

VADDPS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[MAXVL-1:128] := 0

ADDPS (128-bit Legacy SSE Version)
DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[95:64] := SRC1[95:64] + SRC2[95:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VADDPS __m512 _mm512_add_ps (__m512 a, __m512 b);
VADDPS __m512 __mm512_mask_add_ps (__m512 s, __m512 a, __m512 k, __m512 b);
VADDPS __m512 __mm512_maskz_add_ps (__m512 k, __m512 a, __m512 b);
VADDPS __m256 __mm256_mask_add_ps (__m256 s, __m256 k, __m256 a, __m256 k, __m256 b);
VADDPS __m256 __mm256_maskz_add_ps (__m256 k, __m256 a, __m256 b);
VADDPS __m128 __mm_mask_add_ps (__m128 s, __m128 k, __m128 a, __m128 b);
VADDPS __m128 __mm_maskz_add_ps (__m128 k, __m128 a, __m128 b);
VADDPS __m128 __mm512_mask_add_round_ps (__m512 s, __m512 k, __m512 a, __m512 b, int);
VADDPS __m512 __mm512_mask_add_round_ps (__m512 s, __m512 k, __m512 a, __m512 b, int);
VADDPS __m512 __mm512_maskz_add_round_ps (__m512 k, __m512 a, __m512 b, int);
ADDPS __m256 __mm256_add_ps (__m256 a, __m256 b);
ADDPS __m128 __mm_add_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
ADDSD—Add Scalar Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add the low double precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 58 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add the low double precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 58 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Add the low double precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Adds the low double precision floating-point values from the second source operand and the first source operand and stores the double precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VADDSD (EVEX Encoded Version)

IF (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;

IF $k1[0]$ or *no writemask*
  THEN   \[ \text{DEST}[63:0] := \text{SRC1}[63:0] + \text{SRC2}[63:0] \]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged* 
    ELSE ; zeroing-masking
      THEN \[ \text{DEST}[63:0] := 0 \]
  FI;

FI;

\[ \text{DEST}[127:64] := \text{SRC1}[127:64] \]

\[ \text{DEST}[\text{MAXVL}-1:128] := 0 \]

VADDSD (VEX.128 Encoded Version)

\[ \text{DEST}[63:0] := \text{SRC1}[63:0] + \text{SRC2}[63:0] \]

\[ \text{DEST}[127:64] := \text{SRC1}[127:64] \]

\[ \text{DEST}[\text{MAXVL}:-1:128] := 0 \]

ADDSD (128-bit Legacy SSE Version)

\[ \text{DEST}[63:0] := \text{DEST}[63:0] + \text{SRC}[63:0] \]

\[ \text{DEST}[\text{MAXVL}:-1:64] \] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

\[ \text{VADDSD}_{\text{m}128d}_{\text{mm}_\text{mask}\text{add}_{\text{sd}(\text{m}128d_{s}, \text{mmask8}_{k}, \text{m}128d_{a}, \text{m}128d_{b})}}; \]
\[ \text{VADDSD}_{\text{m}128d}_{\text{mm_maskz}_{\text{add}_{\text{sd}(\text{mmask8}_{k}, \text{m}128d_{a}, \text{m}128d_{b})}}; \]
\[ \text{VADDSD}_{\text{m}128d}_{\text{mm}_{\text{add_round}_{\text{sd}(\text{m}128d_{a}, \text{m}128d_{b}, \text{int})}}; \]
\[ \text{VADDSD}_{\text{m}128d}_{\text{mm_mask}_{\text{add_round}_{\text{sd}(\text{mmask8}_{k}, \text{m}128d_{a}, \text{m}128d_{b}, \text{int})}}; \]
\[ \text{ADDSD}_{\text{m}128d}_{\text{mm}_{\text{add}_{\text{sd}(\text{m}128d_{a}, \text{m}128d_{b})}}; \]

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
ADDSS—Add Scalar Single Precision Floating-Point Values

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 58 /r ADDSS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Add the low single precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1 IG 58 /r VADDSS xmm1,xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add the low single precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{</td>
<td>er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMfr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMfr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMfr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Adds the low single precision floating-point values from the second source operand and the first source operand, and stores the double precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
**Operation**

**VADDSS (EVEX Encoded Versions)**

IF (EVEX.b = 1) AND SRC2 *is a register*

THEN

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] := SRC1[31:0] + SRC2[31:0]
ELSE

    IF *merging-masking* ; merging-masking
    THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[31:0] := 0
    FI;
FI;

DEST[MAXVL-1:128] := 0

**VADDSS DEST, SRC1, SRC2 (VEX.128 Encoded Version)**

DEST[31:0] := SRC1[31:0] + SRC2[31:0]
DEST[MAXVL-1:128] := 0

**ADDSS DEST, SRC (128-bit Legacy SSE Version)**

DEST[31:0] := DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
ADDSUBPD—Packed Double Precision Floating-Point Add/Subtract

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F D0 /r ADDSUBPD xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Add/subtract double precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D0 /r VADDSUBPD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add/subtract packed double precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D0 /r VADDSUBPD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add / subtract packed double precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Adds odd-numbered double precision floating-point values of the first source operand (second operand) with the corresponding double precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double precision floating-point values from the second source operand from the corresponding double precision floating point values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**Operation**

**ADDSUBPD (128-bit Legacy SSE Version)**
\[
\begin{align*}
\text{DEST}[63:0] &: \text{DEST}[63:0] - \text{SRC}[63:0] \\
\text{DEST}[127:64] &: \text{DEST}[127:64] + \text{SRC}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] &: \text{Unmodified}
\end{align*}
\]

**VADDUBPD (VEX.128 Encoded Version)**
\[
\begin{align*}
\text{DEST}[63:0] &: \text{SRC1}[63:0] - \text{SRC2}[63:0] \\
\text{DEST}[127:64] &: \text{SRC1}[127:64] + \text{SRC2}[127:64] \\
\text{DEST}[\text{MAXVL}-1:128] &: 0
\end{align*}
\]

**VADDUBPD (VEX.256 Encoded Version)**
\[
\begin{align*}
\text{DEST}[63:0] &: \text{SRC1}[63:0] - \text{SRC2}[63:0] \\
\text{DEST}[127:64] &: \text{SRC1}[127:64] + \text{SRC2}[127:64] \\
\text{DEST}[255:192] &: \text{SRC1}[255:192] + \text{SRC2}[255:192]
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**
\[
\begin{align*}
\text{ADDUBPD} \quad \text{__m128d} \quad \text{_mm_addsub_pd(__m128d} \ a, \ __m128d} \ b) \\
\text{VADDUBPD} \quad \text{__m256d} \quad \text{_mm256_addsub_pd} \quad (\text{__m256d} \ a, \ __m256d} \ b)
\end{align*}
\]

**Exceptions**

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Table 2-19, ”Type 2 Class Exception Conditions.”
ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F D0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Add/subtract single precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.F2.0F.WIG D0 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add/subtract single precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F2.0F.WIG D0 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add / subtract single precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Adds odd-numbered single precision floating-point values of the first source operand (second operand) with the corresponding single precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single precision floating-point values from the second source operand from the corresponding single precision floating-point values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
### ADDSUBPS—Packed Single Precision Floating-Point Add/Subtract

#### Operation

**ADDSUBPS (128-bit Legacy SSE Version)**
- \( \text{DEST}[31:0] := \text{DEST}[31:0] - \text{SRC}[31:0] \)
- \( \text{DEST}[95:64] := \text{DEST}[95:64] - \text{SRC}[95:64] \)
- \( \text{DEST}[\text{MAXVL}-1:128] (\text{Unmodified}) \)

**VADDSUBPS (VEX.128 Encoded Version)**
- \( \text{DEST}[31:0] := \text{SRC1}[31:0] - \text{SRC2}[31:0] \)
- \( \text{DEST}[95:64] := \text{SRC1}[95:64] - \text{SRC2}[95:64] \)
- \( \text{DEST}[\text{MAXVL}-1:128] := 0 \)

**VADDSUBPS (VEX.256 Encoded Version)**
- \( \text{DEST}[31:0] := \text{SRC1}[31:0] - \text{SRC2}[31:0] \)
- \( \text{DEST}[95:64] := \text{SRC1}[95:64] - \text{SRC2}[95:64] \)

#### Intel C/C++ Compiler Intrinsic Equivalent

- \( \text{ADDSUBPS } \_m128 \_mm_addsub_ps(\_m128 a, \_m128 b) \)
- \( \text{VADDSUBPS } \_m256 \_mm256_addsub_ps(\_m256 a, \_m256 b) \)

#### Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.
SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
See Table 2-19, "Type 2 Class Exception Conditions."
**ADOX — Unsigned Integer Addition of Two Operands With Overflow Flag**

### Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

### Operation

If OperandSize is 64-bit

\[
\text{THEN OF:DEST}[63:0] := \text{DEST}[63:0] + \text{SRC}[63:0] + \text{OF};
\]

\[
\text{ELSE OF:DEST}[31:0] := \text{DEST}[31:0] + \text{SRC}[31:0] + \text{OF};
\]

Fi;

### Flags Affected

OF is updated based on result. CF, SF, ZF, AF, and PF flags are unmodified.

### Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);

unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

### SIMD Floating-Point Exceptions

None.
Protected Mode Exceptions

#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) For an illegal address in the SS segment.

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
    If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) For an illegal address in the SS segment.

#GP(0) For an illegal address in the SS segment.

Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) For an illegal address in the SS segment.

#GP(0) For an illegal address in the SS segment.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
AESDEC—Perform One Round of an AES Decryption Flow

**Opcode/Instruction** | **Op/En** | **64/32-bit Mode** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---

| 66 0F 38 DE /r | A | V/V | AES | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.

| VEX.128.66.0F38.WIG DE /r | B | V/V | AES AVX | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.

| VEX.256.66.0F38.WIG DE /r | B | V/V | VAES | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from xmm2 with two 128-bit round keys from xmm3/m256; store the result in xmm1.

| EVEX.128.66.0F38.WIG DE /r | C | V/V | VAES AVX512VL | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.

| EVEX.256.66.0F38.WIG DE /r | C | V/V | VAES AVX512VL | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from xmm2 with two 128-bit round keys from xmm3/m256; store the result in xmm1.

| EVEX.512.66.0F38.WIG DE /r | C | V/V | VAES AVX512F | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from xmm2 with four 128-bit round keys from xmm3/m512; store the result in xmm1.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand and one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDECLAST instruction.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.
Operation

**AESDEC**

STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
STATE := InvMixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)

**VAESDEC (128b and 256b VEX Encoded Versions)**

(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

**VAESDEC (EVEX Encoded Version)**

(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    STATE := InvMixColumns( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)
VAESDEC__m256i__mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC__m512i__mm512_aesdec_epi128(__m512i, __m512i);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."
AESDEC128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker Using 128-Bit Key

### Description
The AESDEC128KL instruction performs 10 rounds of AES to decrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

### Operation

**AESDEC128KL**

Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.

Illegal Handle = (HandleReservedBitSet (Handle)) ||
(Hand[0] AND (CPL > 0)) ||
Handle[2] ||
HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);

IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
    IF (Authentic == 0)
        THEN RFLAGS.ZF := 1;
    ELSE
        DEST := AES128Decrypt (DEST, UnwrappedKey) ;
        RFLAGS.ZF := 0;
    FI;
FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

**Flags Affected**
ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

**Intel C/C++ Compiler Intrinsic Equivalent**
AESDEC128KL unsigned char _mm_aesdec128kl_u8(__m128i* odata, __m128i idata, const void* h);
Exceptions (All Operating Modes)

#UD  If the LOCK prefix is used.
     If CPUID.07H:ECX.KL[bit 23] = 0.
     If CR4.KL = 0.
     If CPUID.19H:EBX.AESKLE[bit 0] = 0.
     If CR0.EM = 1.
     If CR4.OSFXSR = 0.

#NM  If CR0.TS = 1.

#PF  If a page fault occurs.

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
        If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
        If the memory address is in a non-canonical form.

#SS(0)  If a memory operand effective address is outside the SS segment limit.
        If a memory address referencing the SS segment is in a non-canonical form.
AESDEC256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker Using 256-Bit Key

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 DF l{11}rrr:bbb AESDEC256KL xmm, m512</td>
<td>A</td>
<td>V/V</td>
<td>AESKLE</td>
<td>Decrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
The AESDEC256KL instruction performs 14 rounds of AES to decrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation
AESDEC256KL
Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.
Illegal Handle = (HandleReservedBitSet (Handle) ||
  (Handle[0] AND (CPL > 0)) ||
  Handle[2] ||
  HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256);
IF (Illegal Handle)
  THEN RFLAGS.ZF := 1;
  ELSE
    (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], lwKey);
    IF (Authentic == 0)
      THEN RFLAGS.ZF := 1;
    ELSE
      DEST := AES256Decrypt (DEST, UnwrappedKey);
      RFLAGS.ZF := 0;
    FI;
  FI;
RFLAGS.OF, SF, AF, PF, CF := 0;

Flags Affected
ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent
AESDEC256KL unsigned char _mm_aesdec256kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here:
Exceptions (All Operating Modes)

#UD  If the LOCK prefix is used.
    If CPUID.07H:ECX.KL[bit 23] = 0.
    If CR4.KL = 0.
    If CPUID.19H:EBX.AESKLE[bit 0] = 0.
    If CR0.EM = 1.
    If CR4.OSFXSR = 0.

#NM  If CR0.TS = 1.

#PF  If a page fault occurs.

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
        If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
        If the memory address is in a non-canonical form.

#SS(0)  If a memory operand effective address is outside the SS segment limit.
        If a memory address referencing the SS segment is in a non-canonical form.
## AESDECLAST—Perform Last Round of an AES Decryption Flow

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AES</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AES AVX</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>VAES</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512VL</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512VL</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512F</td>
<td>Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.
Operation

AESDECLAST
STATE := SRC1;
RoundKey := SRC2;
STATE := InvShiftRows( STATE );
STATE := InvSubBytes( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)

VAESDECLAST (128b and 256b VEX Encoded Versions)
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

VAESDECLAST (EVEX Encoded Version)
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE := SRC1.xmm[i]
    RoundKey := SRC2.xmm[i]
    STATE := InvShiftRows( STATE )
    STATE := InvSubBytes( STATE )
    DEST.xmm[i] := STATE XOR RoundKey
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)
VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded: See Table 2-50, “Type E4NF Class Exception Conditions.”
AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 D8 (11):001:bbb AESDECWIDE128KL m384, &lt;XMM0-7&gt;</td>
<td>A</td>
<td>V/V</td>
<td>AESKLEWIDE_KL</td>
<td>Decrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operands 2—9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRm:r/m (r)</td>
<td>Implicit XMM0-7 (r, w)</td>
</tr>
</tbody>
</table>

### Description

The AESDECWIDE128KL instruction performs ten rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

### Operation

**AESDECWIDE128KL**

Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.

Illegal Handle = (HandleReservedBitSet (Handle) || (Handle[0] AND (CPL > 0)) || Handle[2] || HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128);

IF (Illegal Handle)

THEN RFLAGS.ZF := 1;

ELSE

(UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IvKey);

IF Authentic == 0 {

THEN RFLAGS.ZF := 1;

ELSE

    XMM0 := AES128Decrypt (XMM0, UnwrappedKey);
    XMM1 := AES128Decrypt (XMM1, UnwrappedKey);
    XMM2 := AES128Decrypt (XMM2, UnwrappedKey);
    XMM3 := AES128Decrypt (XMM3, UnwrappedKey);
    XMM4 := AES128Decrypt (XMM4, UnwrappedKey);
    XMM5 := AES128Decrypt (XMM5, UnwrappedKey);
    XMM6 := AES128Decrypt (XMM6, UnwrappedKey);
    XMM7 := AES128Decrypt (XMM7, UnwrappedKey);

    RFLAGS.ZF := 0;

} FI;

FI;

RFLAGS.OF, SF, AF, PF, CF := 0;

### Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

---

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html
AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

Intel C/C++ Compiler Intrinsic Equivalent

AESDECWIDE128KL unsigned char _mm_aesdecwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
	If CPUID.07H:ECX.KL[bit 23] = 0.
	If CR4.KL = 0.
	If CPUID.19H:EBX.AESKLE[bit 0] = 0.
	If CR0.EM = 1.
	If CR4.OSFXSR = 0.
	If CPUID.19H:EBX.WIDE_KL[bit 2] = 0.

#NM If CR0.TS = 1.

#PF If a page fault occurs.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If the memory address is in a non-canonical form.

#SS(0) If a memory operand effective address is outside the SS segment limit.
	If a memory address referencing the SS segment is in a non-canonical form.
AESDECWIDE256KL—Perform 14 Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 D8 11 011 bb</td>
<td>A</td>
<td>V/V</td>
<td>AESKLEWIDE_KL</td>
<td>Decrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operands 2—9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (r)</td>
<td>Implicit XMM0-7 (r, w)</td>
</tr>
</tbody>
</table>

**Description**

The AESDECWIDE256KL instruction performs 14 rounds of AES to decrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding decrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

**Operation**

**AESDECWIDE256KL**

Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.

Illegal Handle = (HandleReservedBitSet (Handle))

   ||
   Handle[0] AND (CPL > 0)) ||
   Handle [2] ||
   HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256;

IF (Illegal Handle) {
   THEN RFLAGS.ZF := 1;
   ELSE
      (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IvKey);
      IF (Authentic == 0)
         THEN RFLAGS.ZF := 1;
      ELSE
         XMM0 := AES256Decrypt (XMM0, UnwrappedKey);
         XMM1 := AES256Decrypt (XMM1, UnwrappedKey);
         XMM2 := AES256Decrypt (XMM2, UnwrappedKey);
         XMM3 := AES256Decrypt (XMM3, UnwrappedKey);
         XMM4 := AES256Decrypt (XMM4, UnwrappedKey);
         XMM5 := AES256Decrypt (XMM5, UnwrappedKey);
         XMM6 := AES256Decrypt (XMM6, UnwrappedKey);
         XMM7 := AES256Decrypt (XMM7, UnwrappedKey);
         RFLAGS.ZF := 0;
      FI;
   FI;
   RFLAGS.OF, SF, AF, PF, CF := 0;

**Flags Affected**

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

---

1. Further details on Key Locker and usage of this instruction can be found here:
**Intel C/C++ Compiler Intrinsic Equivalent**

```c
AESDECWIDE256KL
unsigned char _mm_aesdecwide256kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);
```

**Exceptions (All Operating Modes)**

- **#UD**
  - If the LOCK prefix is used.
  - If CPUID.07H:ECX.KL[bit 23] = 0.
  - If CR4.KL = 0.
  - If CPUID.19H:EBX.AESKLE[bit 0] = 0.
  - If CR0.EM = 1.
  - If CR4.OSFXSR = 0.
  - If CPUID.19H:EBX.WIDE_KL[bit 2] = 0.

- **#NM**
  - If CR0.TS = 1.

- **#PF**
  - If a page fault occurs.

- **#GP(0)**
  - If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
  - If the memory address is in a non-canonical form.

- **#SS(0)**
  - If a memory operand effective address is outside the SS segment limit.
  - If a memory address referencing the SS segment is in a non-canonical form.
### AESENC—Perform One Round of an AES Encryption Flow

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DC /r</td>
<td>A</td>
<td>V/V</td>
<td>AES</td>
<td>Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>AESENC xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG DC /r</td>
<td>B</td>
<td>V/V</td>
<td>AES AVX</td>
<td>Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.</td>
</tr>
<tr>
<td>VAESENC xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG DC /r</td>
<td>B</td>
<td>V/V</td>
<td>VAES</td>
<td>Perform one round of an AES encryption flow, using two 128-bit data (state) from xmm2 with two 128-bit round keys from the xmm3/m256; store the result in xmm1.</td>
</tr>
<tr>
<td>VAESENC ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG DC /r</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512VL</td>
<td>Perform one round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from the xmm3/m128; store the result in xmm1.</td>
</tr>
<tr>
<td>VAESENC xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG DC /r</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512VL</td>
<td>Perform one round of an AES encryption flow, using two 128-bit data (state) from xmm2 with two 128-bit round keys from the xmm3/m256; store the result in xmm1.</td>
</tr>
<tr>
<td>VAESENC ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG DC /r</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512F</td>
<td>Perform one round of an AES encryption flow, using four 128-bit data (state) from xmm2 with four 128-bit round keys from the xmm3/m512; store the result in xmm1.</td>
</tr>
<tr>
<td>VAESENC zmm1, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction performs a single round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

VEX and EVEX encoded versions of the instruction allow 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

### Operation

**AESENC**

STATE := SRC1;
RoundKey := SRC2;
STATE := ShiftRows( STATE );
STATE := SubBytes( STATE );
STATE := MixColumns( STATE );
DEST[127:0] := STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
VAESENCE (128b and 256b VEX Encoded Versions)
(KL,VL) = (1,128), (2,256)
FOR I := 0 to KL-1:
  STATE := SRC1.xmm[i]
  RoundKey := SRC2.xmm[i]
  STATE := ShiftRows( STATE )
  STATE := SubBytes( STATE )
  STATE := MixColumns( STATE )
  DEST.xmm[i] := STATE XOR RoundKey
  DEST[MAXVL-1:VL] := 0

VAESENCE (EVEX Encoded Version)
(KL,VL) = (1,128), (2,256), (4,512)
FOR i := 0 to KL-1:
  STATE := SRC1.xmm[i] // xmm[i] is the i'th xmm word in the SIMD register
  RoundKey := SRC2.xmm[i]
  STATE := ShiftRows( STATE )
  STATE := SubBytes( STATE )
  STATE := MixColumns( STATE )
  DEST.xmm[i] := STATE XOR RoundKey
  DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
(V)AESENCE __m128i _mm_aesenc (__m128i, __m128i)
VAESENCE __m256i _mm256_aesenc_epi128(__m256i, __m256i);
VAESENCE __m512i _mm512_aesenc_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."
AESENCL28KL—Perform Ten Rounds of AES Encryption Flow With Key Locker Using 128-Bit Key

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 DC l(11):rrr:bbb</td>
<td>A</td>
<td>V/V</td>
<td>AESKLE</td>
<td>Encrypt xmm using 128-bit AES key indicated by handle at m384 and store result in xmm.</td>
</tr>
<tr>
<td>AESENCL28KL xmm, m384</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The AESENCL28KL instruction performs ten rounds of AES to encrypt the first operand using the 128-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

### Operation

**AESENCL28KL**

Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.

```
Illegal Handle = (  
    HandleReservedBitSet (Handle) ||  
    (Handle[0] AND (CPL > 0)) ||  
    Handle[1] ||  
    HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128  
);
```

IF (Illegal Handle) {
    THEN RFLAGS.ZF := 1;
    ELSE
        (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IwKey);
        IF (Authentic == 0)
            THEN RFLAGS.ZF := 1;
        ELSE
            DEST := AES128Encrypt (DEST, UnwrappedKey);
            RFLAGS.ZF := 0;
        FI;
    FI;
    RFLAGS.OF, SF, AF, PF, CF := 0;

### Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

### Intel C/C++ Compiler Intrinsic Equivalent

AESENCL28KL unsigned char _mm_aesenc128kl_u8(__m128i* odata, __m128i idata, const void* h);

---

1. Further details on Key Locker and usage of this instruction can be found here:  
Exceptions (All Operating Modes)

#UD        If the LOCK prefix is used.
            If CPUID.07H:ECX.KL[bit 23] = 0.
            If CR4.KL = 0.
            If CPUID.19H:EBX.AESKLE[bit 0] = 0.
            If CR0.EM = 1.
            If CR4.OSFXSR = 0.

#NM        If CR0.TS = 1.

#PF        If a page fault occurs.

#GP(0)     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
            If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment
            selector.
            If the memory address is in a non-canonical form.

#SS(0)     If a memory operand effective address is outside the SS segment limit.
            If a memory address referencing the SS segment is in a non-canonical form.
AESENC256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker Using 256-Bit Key

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 DE l(11):rrr:bbb AESENC256KL xmm, m512</td>
<td>A</td>
<td>V/V</td>
<td>AESKLE</td>
<td>Encrypt xmm using 256-bit AES key indicated by handle at m512 and store result in xmm.</td>
</tr>
</tbody>
</table>

Description

The AESENC256KL instruction performs 14 rounds of AES to encrypt the first operand using the 256-bit key indicated by the handle from the second operand. It stores the result in the first operand if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

**AESENC256KL**

Handle := UnalignedLoad of 512 bit (SRC); // Load is not guaranteed to be atomic.

Illegal Handle = ( HANDLEReservedBitSet (Handle) || (Handle[0] AND (CPL > 0)) || Handle [1] || HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256 );

IF (Illegal Handle) THEN RFLAGS.ZF := 1; ELSE (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IwKey);

IF (Authentic == 0) THEN RFLAGS.ZF := 1; ELSE DEST := AES256Encrypt (DEST, UnwrappedKey) ; RFLAGS.ZF := 0; FI;

RFLAGS.OF, SF, AF, PF, CF := 0;

Flags Affected

ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent

AESENC256KL unsigned char _mm_aesenc256kl_u8(__m128i* odata, __m128i idata, const void* h);

1. Further details on Key Locker and usage of this instruction can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.
Exceptions (All Operating Modes)

#UD  If the LOCK prefix is used.
     If CPUID.07H:ECX.KL[bit 23] = 0.
     If CR4.KL = 0.
     If CPUID.19H:EBX.AESKLE[bit 0] = 0.
     If CR0.EM = 1.
     If CR4.OSFXSR = 0.

#NM  If CR0.TS = 1.

#PF  If a page fault occurs.

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
     If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
     If the memory address is in a non-canonical form.

#SS(0)  If a memory operand effective address is outside the SS segment limit.
     If a memory address referencing the SS segment is in a non-canonical form.
### AESENCLAST—Perform Last Round of an AES Encryption Flow

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DD /r</td>
<td>A</td>
<td>V/V</td>
<td>AES</td>
<td>Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm1 with one 128-bit round key from xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG DD /r</td>
<td>B</td>
<td>V/V</td>
<td>AES AVX</td>
<td>Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG DD /r</td>
<td>B</td>
<td>V/V</td>
<td>VAES</td>
<td>Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG DD /r</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512VL</td>
<td>Perform the last round of an AES encryption flow, using one 128-bit data (state) from xmm2 with one 128-bit round key from xmm3/m128; store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG DD /r</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512VL</td>
<td>Perform the last round of an AES encryption flow, using two 128-bit data (state) from ymm2 with two 128-bit round keys from ymm3/m256; store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG DD /r</td>
<td>C</td>
<td>V/V</td>
<td>VAES AVX512F</td>
<td>Perform the last round of an AES encryption flow, using four 128-bit data (state) from zmm2 with four 128-bit round keys from zmm3/m512; store the result in zmm1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

This instruction performs the last round of an AES encryption flow using one/two/four (depending on vector length) 128-bit data (state) from the first source operand with one/two/four (depending on vector length) round key(s) from the second source operand, and stores the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.
**Operation**

**AESENCLAST**

\[
\begin{align*}
\text{STATE} & := \text{SRC1}; \\
\text{RoundKey} & := \text{SRC2}; \\
\text{STATE} & := \text{ShiftRows}(\text{STATE}); \\
\text{STATE} & := \text{SubBytes}(\text{STATE}); \\
\text{DEST}[127:0] & := \text{STATE XOR RoundKey}; \\
\text{DEST}[^{\text{MAXVL}-1:128}] & \text{ (Unmodified)}
\end{align*}
\]

**VAESENCLAST (128b and 256b VEX Encoded Versions)**

\((KL, VL) = (1,128), (2,256)\)

FOR \(i = 0\) to \(KL-1\):

\[
\begin{align*}
\text{STATE} & := \text{SRC1.xmm}[i] \\
\text{RoundKey} & := \text{SRC2.xmm}[i] \\
\text{STATE} & := \text{ShiftRows}(\text{STATE}) \\
\text{STATE} & := \text{SubBytes}(\text{STATE}) \\
\text{DEST.xmm}[i] & := \text{STATE XOR RoundKey} \\
\text{DEST}[^{\text{MAXVL}-1:VL}] & := 0
\end{align*}
\]

**VAESENCLAST (EVEX Encoded Version)**

\((KL, VL) = (1,128), (2,256), (4,512)\)

FOR \(i = 0\) to \(KL-1\):

\[
\begin{align*}
\text{STATE} & := \text{SRC1.xmm}[i] \\
\text{RoundKey} & := \text{SRC2.xmm}[i] \\
\text{STATE} & := \text{ShiftRows}(\text{STATE}) \\
\text{STATE} & := \text{SubBytes}(\text{STATE}) \\
\text{DEST.xmm}[i] & := \text{STATE XOR RoundKey} \\
\text{DEST}[^{\text{MAXVL}-1:VL}] & := 0
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
\text{(V)AESENCLAST } & \text{ __m128i } \_\text{mm_aesenclast (__m128i, __m128i)} \\
\text{VAESENCLAST } & \text{ __m256i } \_\text{mm256_aesenclast_epi128(__m256i, __m256i)}; \\
\text{VAESENCLAST } & \text{ __m512i } \_\text{mm512_aesenclast_epi128(__m512i, __m512i)};
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."
AESENCE128KL—Perform Ten Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 D8 l11:000:0bb AESENCE128KL m384, &lt;XMM0-7&gt;</td>
<td>A</td>
<td>V/V</td>
<td>AESKLE WIDE_KL</td>
<td>Encrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operands 2—9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (r)</td>
<td>Implicit XMM0-7 (r, w)</td>
</tr>
</tbody>
</table>

### Description

The AESENCE128KL instruction performs ten rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 128-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

### Operation

**AESENCE128KL**

Handle := UnalignedLoad of 384 bit (SRC); // Load is not guaranteed to be atomic.

Illegal Handle = {
  HandleReservedBitSet (Handle) ||
  (Handle[0] AND (CPL > 0)) ||
  Handle[1] ||
  HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES128
};

IF (Illegal Handle)
  THEN RFLAGS.ZF := 1;
ELSE
  (UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate384 (Handle[383:0], IWKey);
  IF Authentic == 0
    THEN RFLAGS.ZF := 1;
  ELSE
    XMM0 := AES128Encrypt (XMM0, UnwrappedKey) ;
    XMM1 := AES128Encrypt (XMM1, UnwrappedKey) ;
    XMM2 := AES128Encrypt (XMM2, UnwrappedKey) ;
    XMM3 := AES128Encrypt (XMM3, UnwrappedKey) ;
    XMM4 := AES128Encrypt (XMM4, UnwrappedKey) ;
    XMM5 := AES128Encrypt (XMM5, UnwrappedKey) ;
    XMM6 := AES128Encrypt (XMM6, UnwrappedKey) ;
    XMM7 := AES128Encrypt (XMM7, UnwrappedKey) ;
    RFLAGS.ZF := 0;
  FI;
FI;

RFLAGS.OF, SF, AF, PF, CF := 0;

1. Further details on Key Locker and usage of this instruction can be found here:

**Flags Affected**
ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

**Intel C/C++ Compiler Intrinsic Equivalent**

```c
AESENCWIDE128KL
unsigned char _mm_aesencwide128kl_u8(__m128i odata[8], const __m128i idata[8], const void* h);
```

**Exceptions (All Operating Modes)**

- **#UD** If the LOCK prefix is used.
  - If CPUID.07H:ECX.KL[bit 23] = 0.
  - If CR4.KL = 0.
  - If CPUID.AESKLE = 0.
  - If CR0.EM = 1.
  - If CR4.OSFXSR = 0.
  - If CPUID.19H:EBX.WIDE_KL[bit 2] = 0.

- **#NM** If CR0.TS = 1.

- **#PF** If a page fault occurs.

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
  - If the memory address is in a non-canonical form.

- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
  - If a memory address referencing the SS segment is in a non-canonical form.
AESENCWIDE256KL—Perform 14 Rounds of AES Encryption Flow With Key Locker on 8 Blocks Using 256-Bit Key

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 D8 (11:0)10 bbb AESENCWIDE256KL m512, &lt;XMM0-7&gt;</td>
<td>A</td>
<td>V/V</td>
<td>AESKLE WIDE_KL</td>
<td>Encrypt XMM0-7 using 256-bit AES key indicated by handle at m512 and store each resultant block back to its corresponding register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operands 2—9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (r)</td>
<td>Implicit XMM0-7 (r, w)</td>
</tr>
</tbody>
</table>

Description

The AESENCWIDE256KL instruction performs 14 rounds of AES to encrypt each of the eight blocks in XMM0-7 using the 256-bit key indicated by the handle from the second operand. It replaces each input block in XMM0-7 with its corresponding encrypted block if the operation succeeds (e.g., does not run into a handle violation failure).

Operation

AESENCWIDE256KL

Handle := UnalignedLoad of 512 bit (SRC);  // Load is not guaranteed to be atomic.

Illegal Handle = {  
HandleReservedBitSet (Handle) ||  
(Handle[0] AND (CPL > 0)) ||  
Handle [1] ||  
HandleKeyType (Handle) != HANDLE_KEY_TYPE_AES256  
};

IF (Illegal Handle)  
THEN RFLAGS.ZF := 1;  
ELSE  
(UnwrappedKey, Authentic) := UnwrapKeyAndAuthenticate512 (Handle[511:0], IWKey);  
IF (Authentic == 0)  
THEN RFLAGS.ZF := 1;  
ELSE  
XMM0 := AES256Encrypt (XMM0, UnwrappedKey);  
XMM1 := AES256Encrypt (XMM1, UnwrappedKey);  
XMM2 := AES256Encrypt (XMM2, UnwrappedKey);  
XMM3 := AES256Encrypt (XMM3, UnwrappedKey);  
XMM4 := AES256Encrypt (XMM4, UnwrappedKey);  
XMM5 := AES256Encrypt (XMM5, UnwrappedKey);  
XMM6 := AES256Encrypt (XMM6, UnwrappedKey);  
XMM7 := AES256Encrypt (XMM7, UnwrappedKey);  
RFLAGS.ZF := 0;  
FI;  
FI;  
RFLAGS.OF, SF, AF, PF, CF := 0;  

1. Further details on Key Locker and usage of this instruction can be found here:  
Flags Affected
ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to a handle violation. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent
AESENcWIDE256KL

Exceptions (All Operating Modes)
#UD  If the LOCK prefix is used.
   If CPUID.07H:ECX.KL[bit 23] = 0.
   If CR4.KL = 0.
   If CPUID.19H:EBX.AESKLE[bit 0] = 0.
   If CR0.EM = 1.
   If CR4.OSFXSR = 0.
   If CPUID.19H:EBX.WIDE_KL[bit 2] = 0.

#NM  If CR0.TS = 1.

#PF  If a page fault occurs.

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
   If the memory address is in a non-canonical form.

#SS(0)  If a memory operand effective address is outside the SS segment limit.
   If a memory address referencing the SS segment is in a non-canonical form.
AESIMC—Perform the AES InvMixColumn Transformation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 DB /r</td>
<td>RM</td>
<td>V/V</td>
<td>AES</td>
<td>Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.</td>
</tr>
<tr>
<td>AESIMC xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG DB /r</td>
<td>RM</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.</td>
</tr>
<tr>
<td>VAESIMC xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Perform the InvMixColumns transformation on the source operand and store the result in the destination operand. The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

Note: the AESIMC instruction should be applied to the expanded AES round keys (except for the first and last round key) in order to prepare them for decryption using the "Equivalent Inverse Cipher" (defined in FIPS 197).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**AESIMC**

DEST[127:0] := InvMixColumns( SRC );
DEST[MAXVL-1:128] (Unmodified)

**VAESIMC**

DEST[127:0] := InvMixColumns( SRC );
DEST[MAXVL-1:128] := 0;

### Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC _m128i _mm_aesimc (_m128i)

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD If VEX.vvvv ≠ 1111B.
AESKEYGENASSIST—AES Round Key Generation Assist

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A DF /r ib AESKEYGENASSIST xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AES</td>
<td>Assist in AES round key generation using an 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG DF /r ib VAESKEYGENASSIST xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>Both AES and AVX flags</td>
<td>Assist in AES round key generation using 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

AESKEYGENASSIST

\[
\begin{align*}
X3[31:0] & : = \text{SRC}[127:96]; \\
X2[31:0] & : = \text{SRC}[95:64]; \\
X1[31:0] & : = \text{SRC}[63:32]; \\
X0[31:0] & : = \text{SRC}[31:0]; \\
\text{RCON}[31:0] & : = \text{ZeroExtend}(\text{imm8}[7:0]); \\
\text{DEST}[31:0] & : = \text{SubWord}(X1); \\
\text{DEST}[63:32] & : = \text{RotWord}(\text{SubWord}(X1)) \text{ XOR RCON}; \\
\text{DEST}[95:64] & : = \text{SubWord}(X3); \\
\text{DEST}[127:96] & : = \text{RotWord}(\text{SubWord}(X3)) \text{ XOR RCON}; \\
\text{DEST}[\text{MAXVL}-1:128] & : (\text{Unmodified})
\end{align*}
\]
VAESKEYGENASSIST
X3[31:0] := SRC [127: 96];
X2[31:0] := SRC [95: 64];
X1[31:0] := SRC [63: 32];
X0[31:0] := SRC [31: 0];
RCON[31:0] := ZeroExtend(imm8[7:0]);
DEST[31:0] := SubWord(X1);
DEST[63:32 ] := RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64]: = SubWord(X3);
DEST[127:96] := RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] := 0;

Intel C/C++ Compiler Intrinsic Equivalent
(V)AESKEYGENASSIST __m128i _mm_aeskeygenassist (__m128i, const int)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions," additionally:
#UD If VEX.vvvv ≠ 1111B.
## AND—Logical AND

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 iw</td>
<td>AND AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>AX AND imm16.</td>
<td></td>
</tr>
<tr>
<td>25 id</td>
<td>AND EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>EAX AND imm32.</td>
<td></td>
</tr>
<tr>
<td>REX.W + 25 id</td>
<td>AND RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX AND imm32 sign-extended to 64-bits.</td>
</tr>
<tr>
<td>80 /4 ib</td>
<td>AND r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 AND imm8.</td>
</tr>
<tr>
<td>REX + 80 /4 ib</td>
<td>AND r/m8 , imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 AND imm8.</td>
</tr>
<tr>
<td>81 /4 iw</td>
<td>AND r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 AND imm16.</td>
</tr>
<tr>
<td>81 /4 id</td>
<td>AND r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 AND imm32.</td>
</tr>
<tr>
<td>REX.W + 81 /4 id</td>
<td>AND r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 AND imm32 sign extended to 64-bits.</td>
</tr>
<tr>
<td>83 /4 ib</td>
<td>AND r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 AND imm8 (sign-extended).</td>
</tr>
<tr>
<td>83 /4 id</td>
<td>AND r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 AND imm8 (sign-extended).</td>
</tr>
<tr>
<td>REX.W + 83 /4 ib</td>
<td>AND r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 AND imm8 (sign-extended).</td>
</tr>
<tr>
<td>20 /r</td>
<td>AND r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 AND r8.</td>
</tr>
<tr>
<td>REX + 20 /r</td>
<td>AND r/m8 , r8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 AND r8 (sign-extended).</td>
</tr>
<tr>
<td>21 /r</td>
<td>AND r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 AND r16.</td>
</tr>
<tr>
<td>21 /r</td>
<td>AND r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 AND r32.</td>
</tr>
<tr>
<td>REX.W + 21 /r</td>
<td>AND r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 AND r32.</td>
</tr>
<tr>
<td>22 /r</td>
<td>AND r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r8 AND r/m8.</td>
</tr>
<tr>
<td>REX + 22 /r</td>
<td>AND r8 , r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 AND r8 (sign-extended).</td>
</tr>
<tr>
<td>23 /r</td>
<td>AND r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 AND r/m16.</td>
</tr>
<tr>
<td>23 /r</td>
<td>AND r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r32 AND r/m32.</td>
</tr>
<tr>
<td>REX.W + 23 /r</td>
<td>AND r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r64 AND r/m64.</td>
</tr>
</tbody>
</table>

**NOTES:**

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.*

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description:**

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.
Operation

DEST := DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0) If the destination operand points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used but the destination is not a memory operand.
ANDN—Logical AND NOT

**Description**

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

\[
\text{DEST} := \text{(NOT SRC1)} \text{ bitwiseAND SRC2}; \\
\text{SF} := \text{DEST[OperandSize -1];} \\
\text{ZF} := \text{(DEST = 0);}
\]

**Flags Affected**

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

Auto-generated from high-level language.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-29, “Type 13 Class Exception Conditions.”
ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 54 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical AND of packed double precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>ANDPD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F 54 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VANDPD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F 54 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>VANDPD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 54 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Return the bitwise logical AND of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>VANDPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 54 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Return the bitwise logical AND of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>VANDPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 54 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>VANDPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise logical AND of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
**Operation**

**VANDPD (EVEX Encoded Versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\[i := j * 64\]

IF \(k1[j]\) OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

\[\text{DEST}[i+63:i] := \text{SRC1}[i+63:i] \text{ BITWISE AND SRC2}[63:0]\]

ELSE

\[\text{DEST}[i+63:i] := \text{SRC1}[i+63:i] \text{ BITWISE AND SRC2}[i+63:i]\]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+63:i] remains unchanged*

ELSE ; zeroing-masking

\[\text{DEST}[i+63:i] = 0\]

FI;

FI;

ENDFOR

\[\text{DEST}[\text{MAXVL}-1:VL] := 0\]

**VANDPD (VEX.256 Encoded Version)**

\[\text{DEST}[63:0] := \text{SRC1}[63:0] \text{ BITWISE AND SRC2}[63:0]\]

\[\text{DEST}[127:64] := \text{SRC1}[127:64] \text{ BITWISE AND SRC2}[127:64]\]


\[\text{DEST}[\text{MAXVL}-1:256] := 0\]

**VANDPD (VEX.128 Encoded Version)**

\[\text{DEST}[63:0] := \text{SRC1}[63:0] \text{ BITWISE AND SRC2}[63:0]\]

\[\text{DEST}[127:64] := \text{SRC1}[127:64] \text{ BITWISE AND SRC2}[127:64]\]

\[\text{DEST}[\text{MAXVL}-1:128] := 0\]

**ANDPD (128-bit Legacy SSE Version)**

\[\text{DEST}[63:0] := \text{DEST}[63:0] \text{ BITWISE AND SRC}[63:0]\]

\[\text{DEST}[127:64] := \text{DEST}[127:64] \text{ BITWISE AND SRC}[127:64]\]

\[\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[\text{VANDPD} \quad \text{__m512d \_mm512\_and\_pd (\text{\_m512d a, \_m512d b});}\]
\[\text{VANDPD} \quad \text{__m512d \_mm512\_mask\_and\_pd (\text{\_m512d s, \_mmask8 k, \_m512d a, \_m512d b});}\]
\[\text{VANDPD} \quad \text{__m512d \_mm512\_maskz\_and\_pd (\text{\_mmask8 k, \_m512d a, \_m512d b});}\]
\[\text{VANDPD} \quad \text{__m256d \_mm256\_mask\_and\_pd (\text{\_m256d s, \_mmask8 k, \_m256d a, \_m256d b});}\]
\[\text{VANDPD} \quad \text{__m256d \_mm256\_maskz\_and\_pd (\text{\_mmask8 k, \_m256d a, \_m256d b});}\]
\[\text{VANDPD} \quad \text{__m128d \_mm\_mask\_and\_pd (\text{\_m128d s, \_mmask8 k, \_m128d a, \_m128d b});}\]
\[\text{VANDPD} \quad \text{__m128d \_mm\_maskz\_and\_pd (\text{\_mmask8 k, \_m128d a, \_m128d b});}\]
\[\text{VANDPD} \quad \text{__m256d \_mm256\_and\_pd (\text{\_m256d a, \_m256d b});}\]
\[\text{ANDPD} \quad \text{__m128d \_mm\_and\_pd (\text{\_m128d a, \_m128d b});}\]

**SIMD Floating-Point Exceptions**

None.
Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
## ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 54 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.0F 54 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.0F 54 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 54 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 54 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 54 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical AND of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX encoded versions:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

**VANDPS (EVEX Encoded Versions)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times 32\]

IF \(k1[j] \text{ OR } *\text{no writemask}\)

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN

\[\text{DEST}[i+63:i] := \text{SRC1}[i+31:i] \text{ BITWISE AND } \text{SRC2}[31:0]\]

ELSE

\[\text{DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE AND } \text{SRC2}[i+31:i]\]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+31:i] remains unchanged*

ELSE ; zeroing-masking

\[\text{DEST}[i+31:i] := 0\]

FI;

ENDIF

\[\text{DEST}[\text{MAXVL-1}:\text{VL}] := 0;\]

**VANDPS (VEX.256 Encoded Version)**

\[\text{DEST}[31:0] := \text{SRC1}[31:0] \text{ BITWISE AND } \text{SRC2}[31:0]\]


\[\text{DEST}[95:64] := \text{SRC1}[95:64] \text{ BITWISE AND } \text{SRC2}[95:64]\]

\[\text{DEST}[127:96] := \text{SRC1}[127:96] \text{ BITWISE AND } \text{SRC2}[127:96]\]

\[\text{DEST}[159:128] := \text{SRC1}[159:128] \text{ BITWISE AND } \text{SRC2}[159:128]\]


\[\text{DEST}[223:192] := \text{SRC1}[223:192] \text{ BITWISE AND } \text{SRC2}[223:192]\]

\[\text{DEST}[255:224] := \text{SRC1}[255:224] \text{ BITWISE AND } \text{SRC2}[255:224]\]

\[\text{DEST}[\text{MAXVL-1}:256] := 0;\]

**VANDPS (VEX.128 Encoded Version)**

\[\text{DEST}[31:0] := \text{SRC1}[31:0] \text{ BITWISE AND } \text{SRC2}[31:0]\]


\[\text{DEST}[95:64] := \text{SRC1}[95:64] \text{ BITWISE AND } \text{SRC2}[95:64]\]

\[\text{DEST}[127:96] := \text{SRC1}[127:96] \text{ BITWISE AND } \text{SRC2}[127:96]\]

\[\text{DEST}[\text{MAXVL-1}:128] := 0;\]

**ANDPS (128-bit Legacy SSE Version)**

\[\text{DEST}[31:0] := \text{DEST}[31:0] \text{ BITWISE AND } \text{SRC}[31:0]\]


\[\text{DEST}[95:64] := \text{DEST}[95:64] \text{ BITWISE AND } \text{SRC}[95:64]\]

\[\text{DEST}[127:96] := \text{DEST}[127:96] \text{ BITWISE AND } \text{SRC}[127:96]\]

\[\text{DEST}[\text{MAXVL-1}:128] \text{ (Unmodified)}\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VANDPS __m512 __mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 __mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 __mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 __mm256_mask_z_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 __mm256_mask_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 __mm128_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 __mm128_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 __mm256_and_ps (__m256 a, __m256 b);
VANDPS __m128 __mm128_and_ps (__m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
### ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 55 /r ANDNPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical AND NOT of packed double precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 55 /r VANDNPD zmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND NOT of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 55 /r VANDNPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND NOT of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 55 /r VANDNPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND NOT of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Perform a bitwise logical AND NOT of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVM encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

VANDNPD (EVEX Encoded Versions)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[ \text{FOR } j := 0 \text{ TO } KL-1 \]
\[ i := j \times 64 \]

\[ \text{IF } k1[j] \text{ OR *no writemask*} \]
\[ \text{IF (EVEX.b == 1) AND (SRC2 *is memory*)} \]
\[ \text{THEN} \]
\[ \text{DEST}[i+63:i] := (\text{NOT}(\text{SRC1}[i+63:i])) \text{ BITWISE AND SRC2}[63:0] \]
\[ \text{ELSE} \]
\[ \text{DEST}[i+63:i] := (\text{NOT}(\text{SRC1}[i+63:i])) \text{ BITWISE AND SRC2}[i+63:i] \]
\[ \text{FI}; \]

\[ \text{ELSE} \]
\[ \text{IF *merging-masking*} \]
\[ \text{THEN *DEST}[i+63:i] \text{ remains unchanged*} \]
\[ \text{ELSE} \]
\[ \text{DEST}[i+63:i] = 0 \]
\[ \text{FI}; \]
\[ \text{FI}; \]
\[ \text{ENDFOR} \]
\[ \text{DEST}[\text{MAXVL-1:VL}] := 0 \]

VANDNPD (VEX.256 Encoded Version)

\[ \text{DEST}[63:0] := (\text{NOT}(\text{SRC1}[63:0])) \text{ BITWISE AND SRC2}[63:0] \]
\[ \text{DEST}[127:64] := (\text{NOT}(\text{SRC1}[127:64])) \text{ BITWISE AND SRC2}[127:64] \]
\[ \text{DEST}[255:192] := (\text{NOT}(\text{SRC1}[255:192])) \text{ BITWISE AND SRC2}[255:192] \]
\[ \text{DEST}[\text{MAXVL-1:256}] := 0 \]

VANDNPD (VEX.128 Encoded Version)

\[ \text{DEST}[63:0] := (\text{NOT}(\text{SRC1}[63:0])) \text{ BITWISE AND SRC2}[63:0] \]
\[ \text{DEST}[127:64] := (\text{NOT}(\text{SRC1}[127:64])) \text{ BITWISE AND SRC2}[127:64] \]
\[ \text{DEST}[\text{MAXVL-1:128}] := 0 \]

ANDNPD (128-bit Legacy SSE Version)

\[ \text{DEST}[63:0] := (\text{NOT}(\text{DEST}[63:0])) \text{ BITWISE AND SRC}[63:0] \]
\[ \text{DEST}[127:64] := (\text{NOT}(\text{DEST}[127:64])) \text{ BITWISE AND SRC}[127:64] \]
\[ \text{DEST}[\text{MAXVL-1:128}] \text{ (Unmodified)} \]

Intel C/C++ Compiler Intrinsic Equivalent

\[ \text{VANDNPD } _\text{__m512d } _\text{__mm512_andnot_pd } (_\text{__m512d } a, _\text{__m512d } b); \]
\[ \text{VANDNPD } _\text{__m512d } _\text{__mm512_mask_andnot_pd } (_\text{__m512d } s, _\text{__mmask8 } k, _\text{__m512d } a, _\text{__m512d } b); \]
\[ \text{VANDNPD } _\text{__m512d } _\text{__mm512_maskz_andnot_pd } (_\text{__mmask8 } k, _\text{__m512d } a, _\text{__m512d } b); \]
\[ \text{VANDNPD } _\text{__m256d } _\text{__mm256_mask_andnot_pd } (_\text{__m256d } s, _\text{__mmask8 } k, _\text{__m256d } a, _\text{__m256d } b); \]
\[ \text{VANDNPD } _\text{__m256d } _\text{__mm256_maskz_andnot_pd } (_\text{__mmask8 } k, _\text{__m256d } a, _\text{__m256d } b); \]
\[ \text{VANDNPD } _\text{__m128d } _\text{__mm128_mask_andnot_pd } (_\text{__m128d } s, _\text{__mmask8 } k, _\text{__m128d } a, _\text{__m128d } b); \]
\[ \text{VANDNPD } _\text{__m128d } _\text{__mm128_maskz_andnot_pd } (_\text{__mmask8 } k, _\text{__m128d } a, _\text{__m128d } b); \]
\[ \text{ANDNPD } _\text{__m128d } _\text{__mm_andnot_pd } (_\text{__m128d } a, _\text{__m128d } b); \]

SIMD Floating-Point Exceptions

None.
Other Exceptions

VEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."
ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 55 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical AND NOT of packed single precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>ANDNPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.0F 55 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed single precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VANDNPS xmm1, xmm2,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.0F 55 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical AND NOT of packed single precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>VANDNPS ymm1, ymm2,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.0F.W0 55 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>VANDNPS xmm1 [k1]z,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W0 55 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>VANDNPS ymm1 [k1]z,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W0 55 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical AND of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>VANDNPS zmm1 [k1]z,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise logical AND NOT of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

VANDNPS (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN
        DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
      ELSE
        DEST[i+31:i] := (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] = 0
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VANDNPS (VEX.256 Encoded Version)
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:256] := 0

VANDNPS (VEX.128 Encoded Version)
DEST[31:0] := (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[95:64] := (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] := (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] := 0

ANDNPS (128-bit Legacy SSE Version)
DEST[31:0] := (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[95:64] := (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)


Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m512 __mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 __mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 __mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 __mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 __mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 __mm128_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 __mm128_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
ANDNPS __m128 __mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
ARPL—Adjust RPL Field of Segment Selector

### Description

Compareres the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program’s code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program’s code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode. See “Checking Caller Access Privileges” in Chapter 3, “Protected-Mode Memory Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information about the use of this instruction.

### Operation

IF 64-BIT MODE
THEN
  See MOVSD;
ELSE
  IF DEST[RPL] < SRC[RPL]
    THEN
      ZF := 1;
      DEST[RPL] := SRC[RPL];
    ELSE
      ZF := 0;
  FI;
FI;

### Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The ARPL instruction is not recognized in real-address mode.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD The ARPL instruction is not recognized in virtual-8086 mode.
If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Not applicable.
BEXTR—Bit Field Extract

**Description**

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize - 1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

\[
\begin{align*}
\text{START} & := \text{SRC2}[7:0]; \\
\text{LEN} & := \text{SRC2}[15:8]; \\
\text{TEMP} & := \text{ZERO}_\text{EXTEND}_\text{TO}_\text{512} (\text{SRC1}); \\
\text{DEST} & := \text{ZERO}_\text{EXTEND}(\text{TEMP}[\text{START+LEN}-1: \text{START}]); \\
\text{ZF} & := (\text{DEST} = 0);
\end{align*}
\]

**Flags Affected**

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

**Intel C/C++ Compiler Intrinsic Equivalent**

BEXTR unsigned __int32 _bextr_u32(unsigned __int32 src, unsigned __int32 start, unsigned __int32 len);
BEXTR unsigned __int64 _bextr_u64(unsigned __int64 src, unsigned __int32 start, unsigned __int32 len);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-29, “Type 13 Class Exception Conditions,” additionally:

#UD If VEX.W = 1.
BLENDPD—Blend Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0D /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select packed double precision floating-point values from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VBLENDPD xmm1, xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed double precision floating-point Values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1.</td>
</tr>
<tr>
<td>VBLENDPD ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed double precision floating-point Values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8[3:0]</td>
</tr>
</tbody>
</table>

Description

Double-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [3:0] determine whether the corresponding double precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is “1”, then the double precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or 256-bit memory location. The destination operand is a YMM register.

Operation

BLENDPD (128-bit Legacy SSE Version)

IF (IMM8[0] = 0) THEN DEST[63:0] := DEST[63:0]
ELSE DEST[63:0] := SRC[63:0] FI

IF (IMM8[1] = 0) THEN DEST[127:64] := DEST[127:64]
ELSE DEST[127:64] := SRC[127:64] FI

DEST[MAXVL-1:128] := (Unmodified)

VBLENDPD (VEX.128 Encoded Version)

IF (IMM8[0] = 0) THEN DEST[63:0] := SRC[1][63:0]
ELSE DEST[63:0] := SRC2[63:0] FI

IF (IMM8[1] = 0) THEN DEST[127:64] := SRC[1][127:64]
ELSE DEST[127:64] := SRC2[127:64] FI

DEST[MAXVL-1:128] := 0
VBLENDPD (VEX.256 Encoded Version)

IF (IMM8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
ELSE DEST[63:0] := SRC2[63:0] FI

IF (IMM8[1] = 0) THEN DEST[127:64] := SRC1[127:64]
ELSE DEST[127:64] := SRC2[127:64] FI


Intel C/C++ Compiler Intrinsic Equivalent

BLENDPD _m128d _mm_blend_pd (_m128d v1, _m128d v2, const int mask);

VBLENDPD _m256d _mm256_blend_pd (_m256d a, _m256d b, const int mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions.”
BLENDPS—Blend Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0C /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLENDPS xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select packed single precision floating-point values from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 0C /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VBLENDPS xmm1, xmm2, xmm3/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed single precision floating-point values from xmm2 and xmm3/m128 from mask in imm8 and store the values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.WIG 0C /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VBLENDPS ymm1, ymm2, ymm3/m256, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select packed single precision floating-point values from ymm2 and ymm3/m256 from mask in imm8 and store the values in ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMi</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Packed single precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [7:0] determine whether the corresponding single precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is “1”, then the single precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**Operation**

**BLENDPS (128-bit Legacy SSE Version)**

IF (IMM8[0] = 0) THEN DEST[31:0] := DEST[31:0]
ELSE DEST[31:0] := SRC[31:0] FI


IF (IMM8[2] = 0) THEN DEST[95:64] := DEST[95:64]
ELSE DEST[95:64] := SRC[95:64] FI


DEST[MAXVL-1:128] (Unmodified)
VBLENDPS (VEX.128 Encoded Version)
IF (IMM8[0] = 0) THEN DEST[31:0] := SRC1[31:0]
ELSE DEST[31:0] := SRC2[31:0] FI
IF (IMM8[2] = 0) THEN DEST[95:64] := SRC1[95:64]
ELSE DEST[95:64] := SRC2[95:64] FI
DEST[MAXVL-1:128] := 0

VBLENDPS (VEX.256 Encoded Version)
IF (IMM8[0] = 0) THEN DEST[31:0] := SRC1[31:0]
ELSE DEST[31:0] := SRC2[31:0] FI
IF (IMM8[2] = 0) THEN DEST[95:64] := SRC1[95:64]
ELSE DEST[95:64] := SRC2[95:64] FI

Intel C/C++ Compiler Intrinsic Equivalent
BLENDPS __m128 _mm_blend_ps (__m128 v1, __m128 v2, const int mask);
VBLENDPS __m256 _mm256_blend_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions."
## BLENDVPD—Variable Blend Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 15 /r BLENDVPD xmm1, xmm2/m128,  &lt;XMM0&gt;</td>
<td>RM0</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select packed double precision floating-point values from xmm1 and xmm2 from mask specified in XMM0 and store the values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 4B /r /is4 VBLENDVPD xmm1, xmm2, xmm3/m128, xmm4</td>
<td>RVMR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally copy double precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 4B /r /is4 VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4</td>
<td>RVMR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally copy double precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM0</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>implicit XMM0</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMR</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8[7:4]</td>
</tr>
</tbody>
</table>

### Description

Conditionally copy each quadword data element of double precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each quadword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding quadword element in the second source operand, if a mask bit is “1”; or
- the corresponding quadword element in the first source operand, if a mask bit is “0”

The register assignment of the implicit mask operand for BLENDVPD is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPD permits the mask to be any XMM or YMM register. In contrast, BLENDVPD treats XMM0 implicitly as the mask and do not support non-destructive destination operation.
Operation
BLENDVPD (128-bit Legacy SSE Version)
M̂ASK := XM̂M0
IF (M̂ASK[63] = 0) THEN DEST[63:0] := DEST[63:0]
    ELSE DEST[63:0] := SRC[63:0] FI
IF (M̂ASK[127] = 0) THEN DEST[127:64] := DEST[127:64]
    ELSE DEST[127:64] := SRC[127:64] FI
DEST[MAXVL-1:128] (Unmodified)

VBLENDVPD (VEX.128 Encoded Version)
M̂ASK := SRC3
IF (M̂ASK[63] = 0) THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC2[63:0] FI
IF (M̂ASK[127] = 0) THEN DEST[127:64] := SRC1[127:64]
    ELSE DEST[127:64] := SRC2[127:64] FI
DEST[MAXVL-1:128] := 0

VBLENDVPD (VEX.256 Encoded Version)
M̂ASK := SRC3
IF (M̂ASK[63] = 0) THEN DEST[63:0] := SRC1[63:0]
    ELSE DEST[63:0] := SRC2[63:0] FI
IF (M̂ASK[127] = 0) THEN DEST[127:64] := SRC1[127:64]
    ELSE DEST[127:64] := SRC2[127:64] FI
IF (M̂ASK[255] = 0) THEN DEST[255:192] := SRC1[255:192]

Intel C/C++ Compiler Intrinsic Equivalent
BLENDVPD __m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3);
VBLENDVPD __m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);
VBLENDVPD __m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions," additionally:
#UD If VEX.W = 1.
BLENDVPS—Variable Blend Packed Single Precision Floating-Point Values

### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM0</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>implicit XMM0</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMR</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8[7:4]</td>
</tr>
</tbody>
</table>

### Description

Conditionally copy each dword data element of single precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each dword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding dword element in the second source operand, if a mask bit is “1”; or
- the corresponding dword element in the first source operand, if a mask bit is “0”.

The register assignment of the implicit mask operand for BLENDVPS is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPS permits the mask to be any XMM or YMM register. In contrast, BLENDVPS treats XMM0 implicitly as the mask and do not support non-destructive destination operation.
Operation

**BLENDVPS (128-bit Legacy SSE Version)**

\[
\text{MASK} := \text{XMM0} \\
\text{IF} (\text{MASK}[31] = 0) \text{ THEN } \text{DEST}[31:0] := \text{DEST}[31:0] \\
\text{ELSE } \text{DEST}[31:0] := \text{SRC}[31:0] \text{ FI} \\
\text{IF} (\text{MASK}[63] = 0) \text{ THEN } \text{DEST}[63:32] := \text{DEST}[63:32] \\
\text{ELSE } \text{DEST}[63:32] := \text{SRC}[63:32] \text{ FI} \\
\text{IF} (\text{MASK}[95] = 0) \text{ THEN } \text{DEST}[95:64] := \text{DEST}[95:64] \\
\text{ELSE } \text{DEST}[95:64] := \text{SRC}[95:64] \text{ FI} \\
\text{IF} (\text{MASK}[127] = 0) \text{ THEN } \text{DEST}[127:96] := \text{DEST}[127:96] \\
\text{ELSE } \text{DEST}[127:96] := \text{SRC}[127:96] \text{ FI} \\
\text{DEST}[\text{MAXVL}-1:128] (\text{Unmodified})
\]

**VBLENDVPS (VEX.128 Encoded Version)**

\[
\text{MASK} := \text{SRC3} \\
\text{IF} (\text{MASK}[31] = 0) \text{ THEN } \text{DEST}[31:0] := \text{SRC1}[31:0] \\
\text{ELSE } \text{DEST}[31:0] := \text{SRC2}[31:0] \text{ FI} \\
\text{IF} (\text{MASK}[63] = 0) \text{ THEN } \text{DEST}[63:32] := \text{SRC1}[63:32] \\
\text{ELSE } \text{DEST}[63:32] := \text{SRC2}[63:32] \text{ FI} \\
\text{IF} (\text{MASK}[95] = 0) \text{ THEN } \text{DEST}[95:64] := \text{SRC1}[95:64] \\
\text{ELSE } \text{DEST}[95:64] := \text{SRC2}[95:64] \text{ FI} \\
\text{IF} (\text{MASK}[127] = 0) \text{ THEN } \text{DEST}[127:96] := \text{SRC1}[127:96] \\
\text{ELSE } \text{DEST}[127:96] := \text{SRC2}[127:96] \text{ FI} \\
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

**VBLENDVPS (VEX.256 Encoded Version)**

\[
\text{MASK} := \text{SRC3} \\
\text{IF} (\text{MASK}[31] = 0) \text{ THEN } \text{DEST}[31:0] := \text{SRC1}[31:0] \\
\text{ELSE } \text{DEST}[31:0] := \text{SRC2}[31:0] \text{ FI} \\
\text{IF} (\text{MASK}[63] = 0) \text{ THEN } \text{DEST}[63:32] := \text{SRC1}[63:32] \\
\text{ELSE } \text{DEST}[63:32] := \text{SRC2}[63:32] \text{ FI} \\
\text{IF} (\text{MASK}[95] = 0) \text{ THEN } \text{DEST}[95:64] := \text{SRC1}[95:64] \\
\text{ELSE } \text{DEST}[95:64] := \text{SRC2}[95:64] \text{ FI} \\
\text{IF} (\text{MASK}[127] = 0) \text{ THEN } \text{DEST}[127:96] := \text{SRC1}[127:96] \\
\text{ELSE } \text{DEST}[127:96] := \text{SRC2}[127:96] \text{ FI} \\
\text{IF} (\text{MASK}[159] = 0) \text{ THEN } \text{DEST}[159:128] := \text{SRC1}[159:128] \\
\text{ELSE } \text{DEST}[159:128] := \text{SRC2}[159:128] \text{ FI} \\
\text{IF} (\text{MASK}[191] = 0) \text{ THEN } \text{DEST}[191:160] := \text{SRC1}[191:160] \\
\text{ELSE } \text{DEST}[191:160] := \text{SRC2}[191:160] \text{ FI} \\
\text{IF} (\text{MASK}[223] = 0) \text{ THEN } \text{DEST}[223:192] := \text{SRC1}[223:192] \\
\text{ELSE } \text{DEST}[223:192] := \text{SRC2}[223:192] \text{ FI} \\
\text{IF} (\text{MASK}[255] = 0) \text{ THEN } \text{DEST}[255:224] := \text{SRC1}[255:224] \\
\text{ELSE } \text{DEST}[255:224] := \text{SRC2}[255:224] \text{ FI}
\]

*Intel C/C++ Compiler Intrinsic Equivalent*

\[
\text{BLENDVPS} \_\_m128 \_\_mm_blendv\_ps(\_\_m128 v1, \_\_m128 v2, \_\_m128 v3); \\
\text{VBLENDVPS} \_\_m128 \_\_mm_blendv\_ps (\_\_m128 a, \_\_m128 b, \_\_m128 mask); \\
\text{VBLENDVPS} \_\_m256 \_\_mm256\_blendv\_ps (\_\_m256 a, \_\_m256 b, \_\_m256 mask);
\]

*SIMD Floating-Point Exceptions*

None.
**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

- #UD If VEX.W = 1.
BLSI—Extract Lowest Set Isolated Bit

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.0F38.W0 F3 /3</td>
<td>VM</td>
<td>V/V</td>
<td>BMI1</td>
<td>Extract lowest set bit from r/m32 and set that bit in r32.</td>
</tr>
<tr>
<td>BLSI r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LZ.0F38.W1 F3 /3</td>
<td>VM</td>
<td>V/N.E.</td>
<td>BMI1</td>
<td>Extract lowest set bit from r/m64, and set that bit in r64.</td>
</tr>
<tr>
<td>BLSI r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>VEX:vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

temp := (-SRC) bitwiseAND (SRC);
SF := temp[(OperandSize -1)];
ZF := (temp = 0);
IF SRC = 0
   CF := 0;
ELSE
   CF := 1;
FI
DEST := temp;

Flags Affected

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSI unsigned __int32  _blsi_u32(unsigned __int32 src);
BLSI unsigned __int64  _blsi_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, “Type 13 Class Exception Conditions.”
BLSMSK—Get Mask Up to Lowest Set Bit

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.0F38.W0 F3 /2</td>
<td>VM</td>
<td>V/V</td>
<td>BMI1</td>
<td>Set all lower bits in r32 to “1” starting from bit 0 to lowest set bit in r/m32.</td>
</tr>
<tr>
<td>BLSMSK r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LZ.0F38.W1 F3 /2</td>
<td>VM</td>
<td>V/N.E.</td>
<td>BMI1</td>
<td>Set all lower bits in r64 to “1” starting from bit 0 to lowest set bit in r/m64.</td>
</tr>
<tr>
<td>BLSMSK r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>VEX.vvvv (w)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Sets all the lower bits of the destination operand to "1" up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

```c
temp := (SRC-1) XOR (SRC);
SF := temp[OperandSize -1];
ZF := 0;
IF SRC = 0
   CF := 1;
ELSE
   CF := 0;
FI
DEST := temp;
```

**Flags Affected**

SF is updated based on the result. CF is set if the source if zero. ZF and OF flags are cleared. AF and PF flag are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

BLMSK unsigned __int32 _blsmsk_u32(unsigned __int32 src);
BLMSK unsigned __int64 _blsmsk_u64(unsigned __int64 src);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-29, “Type 13 Class Exception Conditions.”
BLSR—Reset Lowest Set Bit

**Opcode/Instruction** | **Op/En** | **64/32-bit Mode** | **CPUID Feature Flag** | **Description**
---|---|---|---|---
VEX.LZ.0F38.W0 F3 /1 | VM | V/V | BMI1 | Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
BLSR r32, r/m32 | | | |
VEX.LZ.0F38.W1 F3 /1 | VM | V/N.E. | BMI1 | Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.
BLSR r64, r/m64 | | | |

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**
Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**
\[
\text{temp} := (\text{SRC}-1) \text{ bitwiseAND} (\text{SRC});
\]
\[
\text{SF} := \text{temp}[\text{OperandSize}-1];
\]
\[
\text{ZF} := (\text{temp} = 0);
\]
\[
\text{IF SRC} = 0
\]
\[
\quad \text{CF} := 1;
\]
\[
\text{ELSE}
\]
\[
\quad \text{CF} := 0;
\]
\[
\text{FI}
\]
\[
\text{DEST} := \text{temp};
\]

**Flags Affected**
ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**
BLSR unsigned _int32 _blsr_u32(unsigned _int32 src);
BLSR unsigned _int64 _blsr_u64(unsigned _int64 src);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-29, “Type 13 Class Exception Conditions.”
BNDCL—Check Lower Bound

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 1A /r</td>
<td>RM</td>
<td>N.E./V</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.</td>
</tr>
<tr>
<td>BNDCL bnd, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 OF 1A /r</td>
<td>RM</td>
<td>V/N.E.</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.</td>
</tr>
<tr>
<td>BNDCL bnd, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

Operation

BNDCL BND, reg
IF reg < BND.LB Then
   BNDSTATUS := 01H;
   #BR;
Fi;

BNDCL BND, mem
TEMP := LEA(mem);
IF TEMP < BND.LB Then
   BNDSTATUS := 01H;
   #BR;
Fi;

Intel C/C++ Compiler Intrinsic Equivalent

BNDCL void _bnd_chk_ptr_lbounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR If lower bound check fails.
#UD If the LOCK prefix is used.
   If ModRM:r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 67H prefix is not used and CS.D=0.
   If 67H prefix is used and CS.D=1.
Real-Address Mode Exceptions
#BR If lower bound check fails.
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.

Virtual-8086 Mode Exceptions
#BR If lower bound check fails.
#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
Same exceptions as in protected mode.
BNDCU/BNDCN—Check Upper Bound

**Description**

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDCU perform 1’s complement operation on the upper bound of bnd first before proceeding with address comparison. BNDCN perform address comparison directly using the upper bound in bnd that is already reverted out of 1’s complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

**Operation**

**BNDCU** BND, reg

IF reg > NOT(BND.UB) Then
    BNDSTATUS := 01H;
    #BR;
Fi;

**BNDCU** BND, mem

TEMP := LEA(mem);
IF TEMP > NOT(BND.UB) Then
    BNDSTATUS := 01H;
    #BR;
Fi;

**BNDCN** BND, reg

IF reg > BND.UB Then
    BNDSTATUS := 01H;
    #BR;
Fi;

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 1A /r BNDCU bnd, r/m32</td>
<td>RM</td>
<td>N.E./V</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB in 1’s complement form).</td>
</tr>
<tr>
<td>F2 0F 1A /r BNDCU bnd, r/m64</td>
<td>RM</td>
<td>V/N.E.</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB in 1’s complement form).</td>
</tr>
<tr>
<td>F2 0F 1B /r BNDCN bnd, r/m32</td>
<td>RM</td>
<td>N.E./V</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB not in 1’s complement form).</td>
</tr>
<tr>
<td>F2 0F 1B /r BNDCN bnd, r/m64</td>
<td>RM</td>
<td>V/N.E.</td>
<td>MPX</td>
<td>Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB not in 1’s complement form).</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDCU perform 1’s complement operation on the upper bound of bnd first before proceeding with address comparison. BNDCN perform address comparison directly using the upper bound in bnd that is already reverted out of 1’s complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA
BNDCN BND, mem
TEMP := LEA(mem);
IF TEMP > BND.UB Then
   BNDSTATUS := 01H;
   #BR;
FI;

Intel C/C++ Compiler Intrinsic Equivalent
BNDCU .void _bnd_chk_ptr_ubounds(const void *q)

Flags Affected
None

Protected Mode Exceptions
#BR If upper bound check fails.
#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 67H prefix is not used and CS.D=0.
   If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions
#BR If upper bound check fails.
#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 16-bit addressing is used.

Virtual-8086 Mode Exceptions
#BR If upper bound check fails.
#UD If the LOCK prefix is used.
   If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
   If 16-bit addressing is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
Same exceptions as in protected mode.
BNLDX—Load Extended Bounds Using Address Translation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 1A /r BNLDX bnd, mib</td>
<td>RM</td>
<td>V/V</td>
<td>MPX</td>
<td>Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>SIB.base (r): Address of pointer SIB.index(r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

BNLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

**Operation**

base := mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value := mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit Mode

A_BT[31:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
  BNDSTATUS := A_BDE | 02H;
  #BR;
FI;
Temp_lb[31:0] := LoadFrom(A_BTE);
Temp_ub[31:0] := LoadFrom(A_BTE + 4);
Temp_ptr[31:0] := LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
  BND.LB := Temp_lb;
  BND.UB := Temp_ub;
ELSE
    BND.LB := 0;
    BND.UB := 0;
    FI;

In 64-bit Mode

A_BT[63:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS := A_BDE | 02H;
    #BR;
    FI;
Temp_lb[63:0] := LoadFrom(A_BTE);
Temp_ub[63:0] := LoadFrom(A_BTE + 8);
Temp_ptr[63:0] := LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB := Temp_lb;
    BND.UB := Temp_ub;
ELSE
    BND.LB := 0;
    BND.UB := 0;
    FI;

Intel C/C++ Compiler Intrinsic Equivalent

BN DLX: Generated by compiler as needed.

Flags Affected

None.

Protected Mode Exceptions

#BR If the bound directory entry is invalid.
#UD If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 67H prefix is not used and CS.D=0.
    If 67H prefix is used and CS.D=1.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.
    If DS register contains a NULL segment selector.
#PF(fault code) If a page fault occurs.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.
#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Appendix E.3.1 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.

#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

#PF(fault code) If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR If the bound directory entry is invalid.
#UD If ModRM is RIP relative.
    If the LOCK prefix is used.
    If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

#GP(0) If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code) If a page fault occurs.
BNDMK—Make Bounds

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 1B /r BNDMK bnd, m32</td>
<td>RM</td>
<td>N.E./V</td>
<td>MPX</td>
<td>Make lower and upper bounds from m32 and store them in bnd.</td>
</tr>
<tr>
<td>F3 0F 1B /r BNDMK bnd, m64</td>
<td>RM</td>
<td>V/N.E.</td>
<td>MPX</td>
<td>Make lower and upper bounds from m64 and store them in bnd.</td>
</tr>
</tbody>
</table>

Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

Operation

BNDL.B := SRCMEM.base;
IF 64-bit mode Then
   BNDL.UB := NOT(LEA.64_bits(SRCMEM));
ELSE
   BNDL.UB := Zero_Extend.64_bits(NOT(LEA.32_bits(SRCMEM)));
FI;

Intel C/C++ Compiler Intrinsic Equivalent

BNDMK void * _bnd_set_ptr_bounds(const void * q, size_t size);

Flags Affected
None.

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 67H prefix is not used and CS.D=0.
If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
If 16-bit addressing is used.
Virtual-8086 Mode Exceptions

#UD  If the LOCK prefix is used.
     If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
     If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD  If the LOCK prefix is used.
     If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
     If RIP-relative addressing is used.

#SS(0)  If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)   If the memory address is in a non-canonical form.

Same exceptions as in protected mode.
BNDMOV—Move Bounds

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 1A /r</td>
<td>RM</td>
<td>N.E./V</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2/m64 to bound register bnd1.</td>
</tr>
<tr>
<td>BNDMOV bnd1, bnd2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 1A /r</td>
<td>RM</td>
<td>V/N.E.</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2/m128 to bound register bnd1.</td>
</tr>
<tr>
<td>BNDMOV bnd1, bnd2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 1B /r</td>
<td>MR</td>
<td>N.E./V</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2 to bnd1/m64.</td>
</tr>
<tr>
<td>BNDMOV bnd1/m64, bnd2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 1B /r</td>
<td>MR</td>
<td>V/N.E.</td>
<td>MPX</td>
<td>Move lower and upper bound from bnd2 to bound register bnd1/m128.</td>
</tr>
<tr>
<td>BNDMOV bnd1/m128, bnd2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM.reg (w)</td>
<td>ModRM.r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM.r/m (w)</td>
<td>ModRM.reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 3-5.

![Figure 3-5. Memory Layout of BNDMOV to/from Memory](image)

This instruction does not change flags.

Operation

**BNDMOV register to register**

DEST.LB := SRC.LB;
DEST.UB := SRC.UB;
BNDMOV from memory
IF 64-bit mode THEN
    DEST.LB := LOAD_QWORD(SRC);
    DEST.UB := LOAD_QWORD(SRC+8);
ELSE
    DEST.LB := LOAD_DWORD_ZERO_EXT(SRC);
    DEST.UB := LOAD_DWORD_ZERO_EXT(SRC+4);
FI;

BNDMOV to memory
IF 64-bit mode THEN
    DEST[63:0] := SRC.LB;
    DEST[127:64] := SRC.UB;
ELSE
    DEST[31:0] := SRC.LB;
    DEST[63:32] := SRC.UB;
FI;

Intel C/C++ Compiler Intrinsic Equivalent
BNDMOV void * _bnd_copy_ptr_bounds(const void *q, const void *r)

Flags Affected
None.

Protected Mode Exceptions
#UD If the LOCK prefix is used but the destination is not a memory operand.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 67H prefix is not used and CS.D=0.
    If 67H prefix is used and CS.D=1.
#SS(0) If the memory operand effective address is outside the SS segment limit.
#GP(0) If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the destination operand points to a non-writable segment
    If the DS, ES, FS, or GS segment register contains a NULL segment selector.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code) If a page fault occurs.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used but the destination is not a memory operand.
    If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
    If 16-bit addressing is used.
#GP(0) If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If the memory operand effective address is outside the SS segment limit.
Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used but the destination is not a memory operand.
  If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
  If 16-bit addressing is used.
#GP(0) If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If the memory operand effective address is outside the SS segment limit.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code) If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used but the destination is not a memory operand.
  If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0) If the memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code) If a page fault occurs.
BNDSTX—Store Extended Bounds Using Address Translation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 1B /r BNDSTX mib, bnd</td>
<td>MR</td>
<td>V/V</td>
<td>MPX</td>
<td>Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>SIB.base (r): Address of pointer SIB.index(r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

### Operation

```plaintext
base := mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value := mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit Mode
A_BT[31:0] := LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
   BNDSTATUS := A_BDE | 02H;
   #BR;
FI;
A_DEST[8][31:0] := ptr_value;
A_DEST[0][31:0] := BND.LB;
A_DEST[4][31:0] := BND.UB;
```

---

3-120 Vol. 2A
In 64-bit Mode

\[
A_{\text{BDE}}[63:0] := (\text{Zero\_extend64}(\text{base}[47+\text{MAWA}:20] \ll 3) + (\text{BNDCFG}[63:12] \ll 12));^1
\]

\[
A_{\text{BT}}[63:0] := \text{LoadFrom}(A_{\text{BDE}});
\]

\[
\text{IF A}_{\text{BT}}[0] \text{ equal } 0 \text{ Then}
\]

\[
\begin{align*}
\text{BNDSTATUS} & := A_{\text{BDE}} | 02H; \\
\#\text{BR};
\end{align*}
\]

\[
\text{FI;}
\]

\[
A_{\text{DEST}}[63:0] := (\text{Zero\_extend64}(\text{base}[19:3] \ll 5) + (A_{\text{BT}}[63:3] \ll 3)); // address of Bound table entry
\]

\[
A_{\text{DEST}}[16][63:0] := \text{ptr\_value};
\]

\[
A_{\text{DEST}}[0][63:0] := \text{BND.LB};
\]

\[
A_{\text{DEST}}[8][63:0] := \text{BND.UB};
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\text{BNDSTX: } _\text{bnd\_store\_ptr\_bounds} (\text{const void **ptr\_addr, const void *ptr\_val});
\]

Flags Affected

None.

Protected Mode Exceptions

\#BR If the bound directory entry is invalid.

\#UD If the LOCK prefix is used.

If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.

If 67H prefix is not used and CS.D=0.

If 67H prefix is used and CS.D=1.

\#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

If DS register contains a NULL segment selector.

If the destination operand points to a non-writable segment.

\#PF(fault code) If a page fault occurs.

Real-Address Mode Exceptions

\#UD If the LOCK prefix is used.

If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.

If 16-bit addressing is used.

\#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

\#UD If the LOCK prefix is used.

If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.

If 16-bit addressing is used.

\#GP(0) If a destination effective address of the Bound Table entry is outside the DS segment limit.

\#PF(fault code) If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

---

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Appendix E.3.1 of Intel\textsuperscript{®} 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
64-Bit Mode Exceptions

#BR If the bound directory entry is invalid.

#UD If ModRM is RIP relative.
   If the LOCK prefix is used.
   If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

#GP(0) If the memory address (A_BDE or A_BTE) is in a non-canonical form.
   If the destination operand points to a non-writable segment

#PF(fault code) If a page fault occurs.
BOUND—Check Array Index Against Bounds

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>62 /r</td>
<td>BOUND r16, m16&amp;16</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Check if r16 (array index) is within bounds specified by m16&amp;16.</td>
</tr>
<tr>
<td>62 /r</td>
<td>BOUND r32, m32&amp;32</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Check if r32 (array index) is within bounds specified by m32&amp;32.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64bit Mode
    THEN
        #UD;
    ELSE
        IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound) THEN
            (* Below lower bound or above upper bound *)
            IF <equation for PL enabled> THEN BNDSTATUS := 0
                #BR;
            FI;
            FI;

Flags Affected

None.
Protected Mode Exceptions

#BR If the bounds test fails.
#UD If second operand is not a memory location.
If the LOCK prefix is used.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#BR If the bounds test fails.
#UD If second operand is not a memory location.
If the LOCK prefix is used.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#BR If the bounds test fails.
#UD If second operand is not a memory location.
If the LOCK prefix is used.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD If in 64-bit mode.
BSF—Bit Scan Forward

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F BC /r</td>
<td>BSF r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Bit scan forward on r/m16.</td>
</tr>
<tr>
<td>0F BC /r</td>
<td>BSF r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Bit scan forward on r/m32.</td>
</tr>
<tr>
<td>REX.W + 0F BC /r</td>
<td>BSF r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Bit scan forward on r/m64.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF SRC = 0
THEN
  ZF := 1;
  DEST is undefined;
ELSE
  ZF := 0;
  temp := 0;
  WHILE Bit(SRC, temp) = 0
  DO
    temp := temp + 1;
  OD;
  OD;
  DEST := temp;
FI;

**Flags Affected**

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
BSR—Bit Scan Reverse

**Description**

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF SRC = 0
    THEN
        ZF := 1;
        DEST is undefined;
    ELSE
        ZF := 0;
        temp := OperandSize - 1;
        WHILE Bit(SRC, temp) = 0
            temp := temp - 1;
        OD;
        OD;
        DEST := temp;
FI;

**Flags Affected**

The ZF flag is set to 1 if the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Instruction Set Reference, A-L**

Vol. 2A 3-127
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
BSWAP—Byte Swap

**Description**
Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**IA-32 Architecture Legacy Compatibility**
The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

**Operation**
\[
\text{TEMP} := \text{DEST} \\
\text{IF 64-bit mode AND OperandSize = 64} \\
\text{THEN} \\
\text{DEST}[7:0] := \text{TEMP}[63:56]; \\
\text{DEST}[15:8] := \text{TEMP}[55:48]; \\
\text{DEST}[23:16] := \text{TEMP}[47:40]; \\
\text{DEST}[31:24] := \text{TEMP}[39:32]; \\
\text{DEST}[39:32] := \text{TEMP}[31:24]; \\
\text{DEST}[47:40] := \text{TEMP}[23:16]; \\
\text{DEST}[55:48] := \text{TEMP}[15:8]; \\
\text{DEST}[63:56] := \text{TEMP}[7:0]; \\
\text{ELSE} \\
\text{DEST}[7:0] := \text{TEMP}[31:24]; \\
\text{DEST}[15:8] := \text{TEMP}[23:16]; \\
\text{DEST}[23:16] := \text{TEMP}[15:8]; \\
\text{DEST}[31:24] := \text{TEMP}[7:0]; \\
\text{FI}; \\
\]

**Flags Affected**
None.

**Exceptions (All Operating Modes)**
#UD If the LOCK prefix is used.
BT—Bit Test

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F A3 /r</td>
<td>BT r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>0F A3 /r</td>
<td>BT r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>REX.W + 0F A3 /r</td>
<td>BT r/m64, r64</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>0F BA /4 ib</td>
<td>BT r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>0F BA /4 ib</td>
<td>BT r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
<tr>
<td>REX.W + 0F BA /4 ib</td>
<td>BT r/m64, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Valid</td>
<td>Store selected bit in CF flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: `Bit(BitBase, BitOffset)` on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

\[
\text{Effective Address} + (4 \times (\text{BitOffset DIV 32}))
\]

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

\[
\text{Effective Address} + (2 \times (\text{BitOffset DIV 16}))
\]

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

\[
\text{CF} := \text{Bit(BitBase, BitOffset)};
\]
Flags Affected
The CF flag contains the value of the selected bit. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If the DS, ES, FS, or GS register contains a NULL segment selector.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
BTC—Bit Test and Complement

Description
Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: `Bit(BitBase, BitOffset)` on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
CF := Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) := NOT Bit(BitBase, BitOffset);

Flags Affected
The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.
Protected Mode Exceptions

#GP(0) If the destination operand points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
       current privilege level is 3.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

#UD If the LOCK prefix is used but the destination is not a memory operand.
BTR—Bit Test and Reset

**Description**
Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```
CF := Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) := 0;
```

**Flags Affected**
The CF flag contains the value of the selected bit before it is cleared. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.
**Protected Mode Exceptions**

- **#GP(0)**: If the destination operand points to a non-writable segment.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- **#GP**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS**: If a memory operand effective address is outside the SS segment limit.
- **#UD**: If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made.
- **#UD**: If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)**: If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)**: If the memory address is in a non-canonical form.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used but the destination is not a memory operand.
BTS—Bit Test and Set

Description
Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
\[
\begin{align*}
\text{CF} & := \text{Bit(BitBase, BitOffset)}; \\
\text{Bit(BitBase, BitOffset)} & := 1;
\end{align*}
\]

Flags Affected
The CF flag contains the value of the selected bit before it is set. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.
Protected Mode Exceptions

- **#GP(0)** If the destination operand points to a non-writable segment.
  - If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register contains a NULL segment selector.

- **#SS(0)** If a memory operand effective address is outside the SS segment limit.

- **#PF(fault-code)** If a page fault occurs.

- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

- **#SS** If a memory operand effective address is outside the SS segment limit.

- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

- **#SS** If a memory operand effective address is outside the SS segment limit.

- **#PF(fault-code)** If a page fault occurs.

- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.

- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.

- **#GP(0)** If the memory address is in a non-canonical form.

- **#PF(fault-code)** If a page fault occurs.

- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
BZHI—Zero High Bits Starting with Specified Bit Position

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.0F38.W0 F5 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Zero bits in r/m32 starting with the position in r32b, write result to r32a.</td>
</tr>
<tr>
<td>BZHI r32a, r/m32, r32b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LZ.0F38.W1 F5 /r</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Zero bits in r/m64 starting with the position in r64b, write result to r64a.</td>
</tr>
<tr>
<td>BZHI r64a, r/m64, r64b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize -1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize -1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

N := SRC2[7:0]  
DEST := SRC1  
IF (N < OperandSize)  
    DEST[OperandSize-1:N] := 0  
FI  
IF (N > OperandSize - 1)  
    CF := 1  
ELSE  
    CF := 0  
FI

Flags Affected

ZF and SF flags are updated based on the result. CF flag is set as specified in the Operation section. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BZHI unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);  
BZHI unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, "Type 13 Class Exception Conditions."
CALL—Call Procedure

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EB cw</td>
<td>CALL rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Call near, relative, displacement relative to next instruction.</td>
</tr>
<tr>
<td>EB cd</td>
<td>CALL rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.</td>
</tr>
<tr>
<td>FF /2</td>
<td>CALL r/m16</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Call near, absolute indirect, address given in r/m16.</td>
</tr>
<tr>
<td>FF /2</td>
<td>CALL r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Call near, absolute indirect, address given in r/m32.</td>
</tr>
<tr>
<td>FF /2</td>
<td>CALL r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Call near, absolute indirect, address given in r/m64.</td>
</tr>
<tr>
<td>9A cd</td>
<td>CALL ptr16:16</td>
<td>D</td>
<td>Invalid</td>
<td>Valid</td>
<td>Call far, absolute, address given in operand.</td>
</tr>
<tr>
<td>9A cp</td>
<td>CALL ptr16:32</td>
<td>D</td>
<td>Invalid</td>
<td>Valid</td>
<td>Call far, absolute, address given in operand.</td>
</tr>
<tr>
<td>FF /3</td>
<td>CALL m16:16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Call far, absolute indirect address given in m16:16. In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.</td>
</tr>
<tr>
<td>FF /3</td>
<td>CALL m16:32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.</td>
</tr>
<tr>
<td>REX.W FF /3</td>
<td>CALL m16:64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>In 64-bit mode: If selector points to a gate, then RIP = 64-bit offset from far pointer referenced in the instruction.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Offset</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional information on near, far, and inter-privilege-level calls. See Chapter 8, "Task Management," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information on performing task switches with the CALL instruction.
**Near Call.** When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16, r/m32, or r/m64). The operand-size attribute determines the size of the target operand (16, 32, or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (rel16 or rel32) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

**Far Calls in Real-Address or Virtual-8086 Mode.** When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Calls in Protected Mode.** When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)
On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack, an optional set of parameters from the calling procedure’s stack, and the segment selector and instruction pointer for the calling procedure’s code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task’s code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 8, “Task Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information on the mechanics of a task switch.

When you execute at task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS’s previous task link field is loaded with the old task’s TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See “Task Linking” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 22, “Mixing 16-Bit and 32-Bit Code,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for more information.

Far Calls in Compatibility Mode. When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target
operand can specify the call gate segment selector either directly with a pointer \((ptr16:16\) or \(ptr16:32\)) or indirectly with a memory location \((m16:16\) or \(m16:32\)). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack and the segment selector and instruction pointer for the calling procedure’s code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location \((m16:16, m16:32\) or \(m16:64\)). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location \((m16:16, m16:32\) or \(m16:64\)). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack and the segment selector and instruction pointer for the calling procedure’s code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.
Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 17, "Control-flow Enforcement Technology (CET)," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for CET details.

**Instruction ordering.** Instructions following a far call may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far call have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Instructions sequentially following a near indirect CALL instruction (i.e., those not at the target) may be executed speculatively. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an LFENCE instruction opcode can be placed after the near indirect CALL in order to block speculative execution.

**Operation**

IF near call
THEN IF near relative call
THEN
   IF OperandSize = 64
      THEN
         tempDEST := SignExtend(DEST); (* DEST is rel32 *)
         tempRIP := RIP + tempDEST;
         IF stack not large enough for a 8-byte return address
            THEN #SS(0); Fl;
            Push(RIP);
         IF ShadowStackEnabled(CPL) AND DEST != 0
            ShadowStackPush8B(RIP);
         Fl;
         RIP := tempRIP;
      FI;
   IF OperandSize = 32
      THEN
         tempEIP := EIP + DEST; (* DEST is rel32 *)
         IF tempEIP is not within code segment limit THEN #GP(0); Fl;
         IF stack not large enough for a 4-byte return address
            THEN #SS(0); Fl;
            Push(EIP);
         IF ShadowStackEnabled(CPL) AND DEST != 0
            ShadowStackPush4B(EIP);
         Fl;
         EIP := tempEIP;
      FI;
   IF OperandSize = 16
      THEN
         tempEIP := (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
         IF tempEIP is not within code segment limit THEN #GP(0); Fl;
         IF stack not large enough for a 2-byte return address
            THEN #SS(0); Fl;
            Push(IP);
         IF ShadowStackEnabled(CPL) AND DEST != 0
            (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
            ShadowStackPush4B(IP);
         Fl;
         EIP := tempEIP;
      FI;
   ELSE (* Near absolute call *)
IF OperandSize = 64 THEN
    tempRIP := DEST; (* DEST is r/m64 *)
    IF stack not large enough for a 8-byte return address
    THEN #SS(0); Fl;
    Push(RIP);
    IF ShadowStackEnabled(CPL)
    ShadowStackPush8B(RIP);
    FI;
    RIP := tempRIP;
FI;
IF OperandSize = 32 THEN
    tempEIP := DEST; (* DEST is r/m32 *)
    IF tempEIP is not within code segment limit THEN #GP(0); Fl;
    IF stack not large enough for a 4-byte return address
    THEN #SS(0); Fl;
    Push(EIP);
    IF ShadowStackEnabled(CPL)
    ShadowStackPush4B(EIP);
    FI;
    EIP := tempEIP;
FI;
IF OperandSize = 16 THEN
    tempEIP := DEST AND 0000FFFFH; (* DEST is r/m16 *)
    IF tempEIP is not within code segment limit THEN #GP(0); Fl;
    IF stack not large enough for a 2-byte return address
    THEN #SS(0); Fl;
    Push(IP);
    IF ShadowStackEnabled(CPL)
    (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
    ShadowStackPush4B(IP);
    FI;
    EIP := tempEIP;
FI;
IF (Call near indirect, absolute indirect)
    IF EndbranchEnabledAndNotSuppressed(CPL)
    IF CPL = 3 THEN
    IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
    THEN
        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
    FI;
    ELSE
    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
    THEN
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    FI;
    FI;
    FI; rel/abs
IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
THEN
  IF OperandSize = 32
  THEN
    IF stack not large enough for a 6-byte return address
      THEN #SS(0); FI;
    IF DEST[31:16] is not zero THEN #GP(0); FI;
    Push(CS); (* Padded with 16 high-order bits *)
    Push(EIP);
    CS := DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
    EIP := DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
  ELSE (* OperandSize = 16 *)
    IF stack not large enough for a 4-byte return address
      THEN #SS(0); FI;
    Push(CS);
    Push(IP);
    CS := DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
    EIP := DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
  FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
THEN
  IF segment selector in target operand NULL
    THEN #GP(0); FI;
  IF segment selector index not within descriptor table limits
    THEN #GP(new code segment selector); FI;
  Read type and access rights of selected segment descriptor;
  IF IA32_EFER.LMA = 0
    THEN
      IF segment type is not a conforming or nonconforming code segment, call
gate, task gate, or TSS
        THEN #GP(segment selector); FI;
    ELSE
      IF segment type is not a conforming or nonconforming code segment or
64-bit call gate,
        THEN #GP(segment selector); FI;
    FI;
  Depending on type and access rights:
  GO TO CONFORMING-CODE-SEGMENT;
  GO TO NONCONFORMING-CODE-SEGMENT;
  GO TO CALL-GATE;
  GO TO TASK-GATE;
  GO TO TASK-STATE-SEGMENT;
FI;

CONFORMING-CODE-SEGMENT:
  IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
  IF DPL > CPL
    THEN #GP(new code segment selector); FI;
  IF segment not present
    THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
THEN #SS(0); FI;
tempEIP := DEST(Offset);
IF target mode = Compatibility mode
THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
THEN
tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
THEN #GP(0); FI;
IF tempEIP is non-canonical
THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
IF OperandSize = 32
THEN
tempPushLIP = CSBASE + EIP;
ELSE
IF OperandSize = 16
THEN
tempPushLIP = CSBASE + IP;
ELSE (* OperandSize = 64 *)
tempPushLIP = RIP;
FI;
FI;
tempPushCS = CS;
FI;
IF OperandSize = 32
THEN
Push(CS); (* Padded with 16 high-order bits *)
Push(EIP);
CS := DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
ELSE
IF OperandSize = 16
THEN
Push(CS);
Push(IP);
CS := DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
ELSE (* OperandSize = 64 *)
Push(CS); (* Padded with 48 high-order bits *)
Push(RIP);
CS := DEST(CodeSegmentSelector);
(* Segment descriptor information also loaded *)
CS(RPL) := CPL;
RIP := tempEIP;
FI;
FI;
IF ShadowStackEnabled(CPL)
IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
(* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
IF (SSP & 0xFFFFFFFF00000000 != 0)
    THEN #GP(0); FI;
FI;
(* align to 8 byte boundary if not already aligned *)
tempSSP = SSP;
Shadow_stack_store 4 bytes of 0 to (SSP - 4)
SSP = SSP & 0xFFFFFFFFFFFFFF8H
ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(tempPushLIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled(CPL)
IF CPL = 3
    THEN
        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_U_CET.SUPPRESS = 0
    ELSE
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;
FI;
END;

NONCONFORMING-CODE-SEGMENT:
IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
IF (RPL > CPL) or (DPL ≠ CPL)
    THEN #GP(new code segment selector); FI;
IF segment not present
    THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
    THEN #SS(0); FI;
tempEIP := DEST(Offset);
IF target mode = Compatibility mode
    THEN tempEIP := tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
    THEN tempEIP := tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF ShadowStackEnabled(CPL)
    IF IA32_EFER.LMA & CS.L
        tempPushLIP = RIP
    ELSE
        tempPushLIP = CSBASE + EIP;
    FI;
    tempPushCS = CS;
FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
CALL—Call Procedure

IF ShadowStackEnabled(CPL)
  IF (IA32_EFER.LMA and DEST(CodeSegmentSelector).L) = 0
    (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
    IF (SSP & 0xFFFFFFFF00000000 != 0)
      THEN #GP(0); FI;
  FI;
  (* align to 8 byte boundary if not already aligned *)
  tempSSP = SSP;
  Shadow_stack_store 4 bytes of 0 to (SSP – 4)
  SSP = SSP & 0xFFFFFFFFFFFFFF8H
  ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
  ShadowStackPush8B(tempPushLIP); (* Padded 32 high-order bits of 0 for 32 bit LIP*)
  ShadowStackPush8B(tempSSP);
  FI;
  IF EndbranchEnabled(CPL)
    IF CPL = 3
      THEN
        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_U_CET.SUPPRESS = 0
      ELSE
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
      FI;
    FI;
  END;

CALL-GATE:
  IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); FI;
  IF call gate not present
    THEN #NP(call-gate selector); FI;
  IF call-gate code-segment selector is NULL
    THEN #NP(call-gate selector); FI;
THEN #GP(0); Fi;
IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); Fi;
Read call-gate code-segment descriptor;
IF call-gate code-segment descriptor does not indicate a code segment
    or call-gate code-segment descriptor DPL > CPL
    THEN #GP(call-gate code-segment selector); Fi;
IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
    not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
    THEN #GP(call-gate code-segment selector); Fi;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); Fi;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
Fi;
END;
MORE-PRIVILEGE:
IF current TSS is 32-bit
    THEN
        TSSstackAddress := (new code-segment DPL ∗ 8) + 4;
        IF (TSSstackAddress + 5) > current TSS limit
            THEN #TS(current TSS selector); Fi;
        NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 4);
        NewESP := 4 bytes loaded from (TSS base + TSSstackAddress);
    ELSE
        IF current TSS is 16-bit
            THEN
                TSSstackAddress := (new code-segment DPL ∗ 4) + 2
                IF (TSSstackAddress + 3) > current TSS limit
                    THEN #TS(current TSS selector); Fi;
                NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                NewESP := 2 bytes loaded from (TSS base + TSSstackAddress);
            ELSE (* current TSS is 64-bit *)
                TSSstackAddress := (new code-segment DPL ∗ 8) + 4;
                IF (TSSstackAddress + 7) > current TSS limit
                    THEN #TS(current TSS selector); Fi;
                NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
        END;
    END;
    FI;
IF IA32_EFER.LMA = 0 and NewSS is NULL
    THEN #TS(NewSS); Fi;
Read new stack-segment descriptor;
IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
    or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
    writable data segment)
    THEN #TS(NewSS); Fi;
IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); Fi;
IF CallGateSize = 32
    THEN
        IF new stack does not have room for parameters plus 16 bytes
THEN #SS(NewSS); Fi;
IF CallGate(InstructionPointer) not within new code-segment limit
    THEN #GP(0); Fi;
SS := newSS; (* Segment descriptor information also loaded *)
ESP := newESP;
CS:EIP := CallGate(CS:InstructionPointer);
(* Segment descriptor information also loaded *)
Push(oldSS:oldESP); (* From calling procedure *)
temp := parameter count from call gate, masked to 5 bits;
Push(parameters from calling procedure's stack, temp)
Push(oldCS:oldEIP); (* Return address to calling procedure *)
ELSE
    IF CallGateSize = 16
        THEN
            IF new stack does not have room for parameters plus 8 bytes
                THEN #SS(NewSS); Fi;
            IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                THEN #GP(0); Fi;
            SS := newSS; (* Segment descriptor information also loaded *)
            ESP := newESP;
            CS:IP := CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            temp := parameter count from call gate, masked to 5 bits;
            Push(parameters from calling procedure's stack, temp)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE (* CallGateSize = 64 *)
            IF pushing 32 bytes on the stack would use a non-canonical address
                THEN #SS(NewSS); Fi;
            IF (CallGate(InstructionPointer) is non-canonical)
                THEN #GP(0); Fi;
            SS := NewSS; (* NewSS is NULL) 
            ESP := NewESP;
            CS:IP := CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        FI;
    FI;
IF ShadowStackEnabled(CPL) AND CPL = 3
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
        ELSE (* adjust so bits 63:N get the value of bit N–1, where N is the CPU's maximum linear-address width *)
            IA32_PL3_SSP := LA_adjust(SSP);
        FI;
    FI;
CPL := CodeSegment(DPL)
CS(RPL) := CPL
IF ShadowStackEnabled(CPL)
    oldSSP := SSP
    SSP := IA32_PLi_SSP; (* where i is the CPL *)
IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then #GP *)
    THEN #GP(0); Fi;
(* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region*)

IF (SSP & ~0x1F) != ((SSP – 24) & ~0x1F)
    #GP(0); FI;

IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
    THEN #GP(0); FI;

expected_token_value = SSP (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT   (* Set the busy bit *)

IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
    THEN #GP(0); FI;

IF oldSS.DPL != 3
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(oldSSP);
    FI;

FI;

IF EndbranchEnabled (CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;

SAME-PRIVILEGE:

IF CallGateSize = 32
    THEN
        IF stack does not have room for 8 bytes
            THEN #SS(0); FI;
        IF CallGate(InstructionPointer) not within code segment limit
            THEN #GP(0); FI;
        CS:EIP := CallGate(CS:EIP) (* Segment descriptor information also loaded *)
        Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
        IF CallGateSize
            THEN
                IF stack does not have room for 4 bytes
                    THEN #SS(0); FI;
                IF CallGate(InstructionPointer) not within code segment limit
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:IP); (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64)
                IF pushing 16 bytes on the stack touches non-canonical addresses
                    THEN #SS(0); FI;
                IF RIP non-canonical
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:instruction pointer);
                (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 32)
                IF stack does not have room for 4 bytes
                    THEN #SS(0); FI;
                IF CallGate(InstructionPointer) not within code segment limit
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:IP); (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            END;
        FI;

    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;

END;

SAME-PRIVILEGE:

IF CallGateSize = 32
    THEN
        IF stack does not have room for 8 bytes
            THEN #SS(0); FI;
        IF CallGate(InstructionPointer) not within code segment limit
            THEN #GP(0); FI;
        CS:EIP := CallGate(CS:EIP) (* Segment descriptor information also loaded *)
        Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
        IF CallGateSize
            THEN
                IF stack does not have room for 4 bytes
                    THEN #SS(0); FI;
                IF CallGate(InstructionPointer) not within code segment limit
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:IP); (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64)
                IF pushing 16 bytes on the stack touches non-canonical addresses
                    THEN #SS(0); FI;
                IF RIP non-canonical
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:IP); (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64)
                IF pushing 16 bytes on the stack touches non-canonical addresses
                    THEN #SS(0); FI;
                IF RIP non-canonical
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:IP); (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 32)
                IF stack does not have room for 4 bytes
                    THEN #SS(0); FI;
                IF CallGate(InstructionPointer) not within code segment limit
                    THEN #GP(0); FI;
                CS:IP := CallGate(CS:IP); (* Segment descriptor information also loaded *)
                Push(oldCS:oldIP); (* Return address to calling procedure *)
            END;
        FI;

    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;

END;
Shadow_stack_store 4 bytes of 0 to (SSP - 4)
SSP = SSP & 0xFFFFFFFFFFFFFFF8H;
(* push cs:lip:ssp on shadow stack *)
ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
    FI;
  FI;
END;

TASK-GATE:
  IF task gate DPL < CPL or RPL
    THEN #GP(task gate selector); FI;
  IF task gate not present
    THEN #NP(task gate selector); FI;
  Read the TSS segment selector in the task-gate descriptor;
  IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    THEN #GP(TSS selector); FI;
  Access TSS descriptor in GDT;
  IF descriptor is not a TSS segment
    THEN #GP(TSS selector); FI;
  IF TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); FI;
  IF TSS not present
    THEN #NP(TSS selector); FI;
  SWtICH-TASKS (with nesting) to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
  IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); FI;
  IF TSS is not present
    THEN #NP(TSS selector); FI;
  SWtICH-TASKS (with nesting) to TSS;
  IF EIP not within code segment limit
    THEN #GP(0); FI;
END;

Flags Affected
All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.
Protected Mode Exceptions

#GP(0)  If the target offset in destination operand is beyond the new code segment limit.
If the segment selector in the destination operand is NULL.
If the code segment selector in the gate is NULL.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If target mode is compatibility mode and SSP is not in low 4GB.
If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.
If “supervisor Shadow Stack” token on new shadow stack is marked busy.
If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.
If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).

#GP(selector)  If a code segment or gate or TSS selector index is outside descriptor table limits.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS’s segment selector.
If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.
If the segment selector from a call gate is beyond the descriptor table limits.
If the DPL for a code-segment obtained from a call gate is greater than the CPL.
If the segment selector for a TSS has its local/global bit set for local.
If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0)  If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
If a memory operand effective address is outside the SS segment limit.
#SS(selector)  If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.
If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.
If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.

#NP(selector)  If a code segment, data segment, call gate, task gate, or TSS is not present.
#TS(selector) If the new stack segment selector and ESP are beyond the end of the TSS.
If the new stack segment selector is NULL.
If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code
segment being accessed.
If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the
code segment descriptor.
If the new stack segment is not a writable data segment.
If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the target offset is beyond the code segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the target offset is beyond the code segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
#GP(selector) If a memory address accessed by the selector is in non-canonical space.
#GP(0) If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0) If a memory address is non-canonical.
If target offset in destination operand is non-canonical.
If the segment selector in the destination operand is NULL.
If the code segment selector in the 64-bit gate is NULL.
If target mode is compatibility mode and SSP is not in low 4GB.
If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
If the token and the stack frame to be pushed on shadow stack are not contained in a naturally
aligned 32-byte region of the shadow stack.
If "supervisor Shadow Stack" token on new shadow stack is marked busy.
If destination mode is 32-bit mode or compatibility mode, but SSP address in "super-visor
shadow" stack token is beyond 4GB.
If SSP address in "supervisor shadow stack" token does not match SSP address in
IA32_PLi_SSP (where i is the new CPL).
#GP(selector)  If code segment or 64-bit call gate is outside descriptor table limits.
   If code segment or 64-bit call gate overlaps non-canonical space.
   If the segment descriptor pointed to by the segment selector in the destination operand is not
   for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.
   If the segment descriptor pointed to by the segment selector in the destination operand is a
   code segment and has both the D-bit and the L-bit set.
   If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the
   segment's segment selector is greater than the CPL.
   If the DPL for a conforming-code segment is greater than the CPL.
   If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
   If the upper type field of a 64-bit call gate is not 0x0.
   If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
   If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
   If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the
   L-bit set and the D-bit clear.
   If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it
   is a code segment.

#SS(0)     If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack
   segment when no stack switch occurs.
   If a memory operand effective address is outside the SS segment limit.
   If the stack address is in a non-canonical form.

#SS(selector)  If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error
   code onto the stack violates the canonical boundary when a stack switch occurs.

#NP(selector)   If a code segment or 64-bit call gate is not present.

#TS(selector)   If the load of the new RSP exceeds the limit of the TSS.

#UD  (64-bit mode only) If a far call is direct to an absolute address in memory.
   If the LOCK prefix is used.

#PF(fault-code) If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the
   current privilege level is 3.
CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>CBW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>AX := sign-extend of AL.</td>
</tr>
<tr>
<td>98</td>
<td>CWDE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>EAX := sign-extend of AX.</td>
</tr>
<tr>
<td>REX.W + 98</td>
<td>CDQE</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX := sign-extend of EAX.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register. CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

### Operation

IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX := SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX := SignExtend(AX); Fl;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX := SignExtend(EAX);
    Fl;

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
CLAC—Clear AC Flag in EFLAGS Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 CA CLAC</td>
<td>ZO</td>
<td>V/V</td>
<td>SMAP</td>
<td>Clear the AC flag in the EFLAGS register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

**Operation**

EFLAGS.AC := 0;

**Flags Affected**

AC cleared. Other flags are unaffected.

**Protected Mode Exceptions**

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

**Real-Address Mode Exceptions**

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

**Virtual-8086 Mode Exceptions**

#UD The CLAC instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

**64-Bit Mode Exceptions**

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.
CLC—Clear Carry Flag

**Description**
Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

**Operation**
CF := 0;

**Flags Affected**
The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**
#UD If the LOCK prefix is used.

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

### Opcode Instruction Op/En

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>CLC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Clear CF flag.</td>
</tr>
</tbody>
</table>
CLD—Clear Direction Flag

Opcode | Instruction | Op/En | 64-bit Mode | Compat/Leg Mode | Description
---|---|---|---|---|---
FC | CLD | ZO | Valid | Valid | Clear DF flag.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

### Operation

DF := 0;

### Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

#UD 
If the LOCK prefix is used.
CLDEMOTE—Cache Line Demote

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 1C /0 CLDEMOTE m8</td>
<td>A</td>
<td>V/V</td>
<td>CLDEMOTE</td>
<td>Hint to hardware to move the cache line containing m8 to a more distant level of the cache without writing back to memory.</td>
</tr>
</tbody>
</table>

Description

Hints to hardware that the cache line that contains the linear address specified with the memory operand should be moved ("demoted") from the cache(s) closest to the processor core to a level more distant from the processor core. This may accelerate subsequent accesses to the line by other cores in the same coherence domain, especially if the line was written by the core that demotes the line. Moving the line in such a manner is a performance optimization, i.e., it is a hint which does not modify architectural state. Hardware may choose which level in the cache hierarchy to retain the line (e.g., L3 in typical server designs). The source operand is a byte memory location.

The availability of the CLDEMOTE instruction is indicated by the presence of the CPUID feature flag CLDEMOTE (bit 25 of the ECX register in sub-leaf 07H, see "CPUID—CPU Identification"). On processors which do not support the CLDEMOTE instruction (including legacy hardware) the instruction will be treated as a NOP.

A CLDEMOTE instruction is ordered with respect to stores to the same cache line, but unordered with respect to other instructions including memory fences, CLDEMOTE, CLWB or CLFLUSHOPT instructions to a different cache line. Since CLDEMOTE will retire in order with respect to stores to the same cache line, software should ensure that after issuing CLDEMOTE the line is not accessed again immediately by the same core to avoid cache data movement penalties.

The effective memory type of the page containing the affected line determines the effect; cacheable types are likely to generate a data movement operation, while uncacheable types may cause the instruction to be ignored. Speculative fetching can occur at any time and is not tied to instruction execution. The CLDEMOTE instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms. That is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLDEMOTE instruction that references the cache line.

Unlike CLFLUSH, CLFLUSHOPT, and CLWB instructions, CLDEMOTE is not guaranteed to write back modified data to memory.

The CLDEMOTE instruction may be ignored by hardware in certain cases and is not a guarantee.

The CLDEMOTE instruction can be used at all privilege levels. In certain processor implementations the CLDEMOTE instruction may set the A bit but not the D bit in the page tables.

If the line is not found in the cache, the instruction will be treated as a NOP.

In some implementations, the CLDEMOTE instruction may always cause a transactional abort with Transactional Synchronization Extensions (TSX). However, programmers must not rely on CLDEMOTE instruction to force a transactional abort.

1. The Mod field of the ModR/M byte cannot have value 11B.
Operation
Cache_Line_Demote(m8);

Flags Affected
None.

C/C++ Compiler Intrinsic Equivalent
CLDEMote void _cldemote(const void*);

Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
CLFLUSH—Flush Cache Line

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE 77</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Flushes cache line containing m8.</td>
</tr>
<tr>
<td>CLFLUSH m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (CPUID.01H:EDX[bit 19]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, and fence instructions. They are not ordered with respect to executions of CLFLUSHOPT and CLWB. Software can use the SFENCE instruction to order an execution of CLFLUSH relative to one of those operations.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSH instruction may cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
Flush_Cache_Line(SRC);
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
CLFLUSH void _mm_clflush(void const *p)
```

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.
**Protected Mode Exceptions**

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF(fault-code) For a page fault.
#UD If CPUID.01H:EDX.CLFSH[bit 19] = 0.
If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD If CPUID.01H:EDX.CLFSH[bit 19] = 0.
If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#UD If CPUID.01H:EDX.CLFSH[bit 19] = 0.
If the LOCK prefix is used.
### CLFLUSHOPT—Flush Cache Line Optimized

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFx 66 0F AE 77</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Flushes cache line containing m8.</td>
</tr>
<tr>
<td>CLFLUSHOPT m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSHOPT is indicated by the presence of the CPUID feature flag CLFLUSHOPT (CPUID.(EAX=07H,ECX=0H):EBX[bit 23]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to older writes to the cache line being invalidated. They are not ordered with respect to other executions of CLFLUSHOPT, to executions of CLFLUSH and CLWB, or to younger writes to the cache line being invalidated. Software can use the SFENCE instruction to order an execution of CLFLUSHOPT relative to one of those operations.

The CLFLUSHOPT instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSHOPT instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSHOPT instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSHOPT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSHOPT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSHOPT instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

CLFLUSHOPT operation is the same in non-64-bit modes and 64-bit mode.

#### Operation

Flush_Cache_Line_Optimized(SRC);

#### Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSHOPT void _mm_clflushopt(void const *p)
Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF(fault-code) For a page fault.
#UD If CPUID.(EAX=07H,ECX=0H):EBX.CLFLUSHOPT[bit 23] = 0.
If the LOCK prefix is used.
If an instruction prefix F2H or F3H is used.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD If CPUID.(EAX=07H,ECX=0H):EBX.CLFLUSHOPT[bit 23] = 0.
If the LOCK prefix is used.
If an instruction prefix F2H or F3H is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.
#PF(fault-code) For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#UD If CPUID.(EAX=07H,ECX=0H):EBX.CLFLUSHOPT[bit 23] = 0.
If the LOCK prefix is used.
If an instruction prefix F2H or F3H is used.
CLI—Clear Interrupt Flag

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

### Instruction Set Reference, A-L

**CLI—Clear Interrupt Flag**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA</td>
<td>CLI</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Clear interrupt flag; interrupts disabled when interrupt flag cleared.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

### Table 3-7. Decision Table for CLI Results

<table>
<thead>
<tr>
<th>Mode</th>
<th>IOPL</th>
<th>CLI Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-address</td>
<td>X&lt;sup&gt;1&lt;/sup&gt;</td>
<td>IF = 0</td>
</tr>
<tr>
<td>Protected, not PVI&lt;sup&gt;2&lt;/sup&gt;</td>
<td>≥ CPL</td>
<td>IF = 0</td>
</tr>
<tr>
<td></td>
<td>&lt; CPL</td>
<td>#GP fault</td>
</tr>
<tr>
<td>Protected, PVI&lt;sup&gt;3&lt;/sup&gt;</td>
<td>3</td>
<td>IF = 0</td>
</tr>
<tr>
<td>Virtual-8086, not VME&lt;sup&gt;3&lt;/sup&gt;</td>
<td>0–2</td>
<td>VIF = 0</td>
</tr>
<tr>
<td>Virtual-8086, VME&lt;sup&gt;3&lt;/sup&gt;</td>
<td>3</td>
<td>IF = 0</td>
</tr>
<tr>
<td></td>
<td>0–2</td>
<td>#GP fault</td>
</tr>
</tbody>
</table>

**NOTES:**

1. X = This setting has no effect on instruction operation.
2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.
**Operation**

IF CR0.PE = 0
   THEN IF := 0; (* Reset Interrupt Flag *)
   ELSE
      IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
         THEN IF := 0; (* Reset Interrupt Flag *)
      ELSE
         IF VME mode OR PVI mode
            THEN VIF := 0; (* Reset Virtual Interrupt Flag *)
         ELSE #GP(0);
      FI;
   FI;
FI;

**Flags Affected**

Either the IF flag or the VIF flag is cleared to 0. Other flags are unaffected.

**Protected Mode Exceptions**

#GP(0)  If CPL is greater than IOPL and PVI mode is not active.
        If CPL is greater than IOPL and less than 3.
#UD    If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD    If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)  If IOPL is less than 3 and VME mode is not active.
#UD    If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
CLRSSBSY—Clear Busy Flag in a Supervisor Shadow Stack Token

**Description**
Clear busy flag in supervisor shadow stack token reference by m64. Subsequent to marking the shadow stack as not busy the SSP is loaded with value 0.

**Operation**
IF (CR4.CET = 0)
    THEN #UD; Fl;

IF (IA32_S.CET.SH_STK_EN = 0)
    THEN #UD; Fl;

IF CPL > 0
    THEN GP(0); Fl;

SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
    THEN #GP(0); Fl;

expected_token_value = SSP_LA | BUSY_BIT (* busy bit - bit position 0 - must be set *)
new_token_value = SSP_LA (* Clear the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP_LA, new_token_value, expected_token_value) != expected_token_value
    invalid_token := 1; Fl

(* Set the CF if invalid token was detected *)
RFLAGS.CF = (invalid_token == 1) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF := 0;
SSP := 0

**Flags Affected**
CF is set if an invalid token was detected, else it is cleared. ZF, PF, AF, OF, and SF are cleared.
Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
      IF IA32_S_CET.SH_STK_EN = 0.
#GP(0) If memory operand linear address not aligned to 8 bytes.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If destination is located in a non-writeable segment.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
      If CPL is not 0.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.

Real-Address Mode Exceptions
#UD The CLRSSBSY instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The CLRSSBSY instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD Same exceptions as in protected mode.
#GP(0) Same exceptions as in protected mode.
#PF(fault-code) If a page fault occurs.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
      IF IA32_S_CET.SH_STK_EN = 0.
#GP(0) If memory operand linear address not aligned to 8 bytes.
   If CPL is not 0.
   If the memory address is in a non-canonical form.
      If token is invalid.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
CLTS—Clear Task-Switched Flag in CR0

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 06</td>
<td>CLTS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Clears TS flag in CR0.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 26, “VMX Non-Root Operation,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation

CR0.TS[bit 3] := 0;

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) CLTS is not recognized in virtual-8086 mode.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If the CPL is greater than 0.
- #UD If the LOCK prefix is used.
CLUI—Clear User Interrupt Flag

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 01 EE CLUI</td>
<td>ZO</td>
<td>V/I</td>
<td>UINTR</td>
<td>Clear user interrupt flag; user interrupts blocked when user interrupt flag cleared.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

CLUI clears the user interrupt flag (UIF). Its effect takes place immediately: a user interrupt cannot be delivered on the instruction boundary following CLUI.

An execution of CLUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been caused due to an execution of CLI.

Operation

UIF := 0;

Flags Affected

None.

Protected Mode Exceptions

#UD The CLUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The CLUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The CLUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The CLUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
   If executed inside an enclave.
   If CR4.UINTR = 0.
   If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.
CLWB—Cache Line Write Back

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F AE /6 CLwb m8</td>
<td>M</td>
<td>V/V</td>
<td>CLWB</td>
<td>Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

 Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

 The availability of CLWB instruction is indicated by the presence of the CPUID feature flag CLWB (bit 24 of the EBX register, see “CPUID — CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

 The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLWB instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLWB instruction that references the cache line).

 Executions of the CLWB instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to older writes to the cache line being written back. They are not ordered with respect to other executions of CLWB, to executions of CLFLUSH and CLFLUSHOPT, or to younger writes to the cache line being written back. Software can use the SFENCE instruction to order an execution of CLWB relative to one of those operations.

 For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

 The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the accessed flag but not the dirty flag in the page tables.

 In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

### Operation

Cache Line_Write_Back(m8);

---

1. The Mod field of the ModR/M byte cannot have value 11B.
Flags Affected
None.

C/C++ Compiler Intrinsic Equivalent
CLWB void _mm_clwb(void const *p);

Protected Mode Exceptions
#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF(fault-code) For a page fault.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.
#PF(fault-code) For a page fault.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
    If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
CMC—Complement Carry Flag

**Description**
Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

EFLAGS.CF[bit 0] := NOT EFLAGS.CF[bit 0];

**Flags Affected**
The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD If the LOCK prefix is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>CMC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Complement CF flag.</td>
</tr>
</tbody>
</table>
## CMOVcc—Conditional Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 47 <code>/r</code></td>
<td>CMOVA r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 47 <code>/r</code></td>
<td>CMOVA r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 47 <code>/r</code></td>
<td>CMOVA r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 43 <code>/r</code></td>
<td>CMOVAE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 43 <code>/r</code></td>
<td>CMOVAE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if above or equal (CF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 43 <code>/r</code></td>
<td>CMOVAE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 42 <code>/r</code></td>
<td>CMOVB r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>0F 42 <code>/r</code></td>
<td>CMOVB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 42 <code>/r</code></td>
<td>CMOVB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>0F 46 <code>/r</code></td>
<td>CMOVBE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 46 <code>/r</code></td>
<td>CMOVBE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 46 <code>/r</code></td>
<td>CMOVBE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 43 <code>/r</code></td>
<td>CMOVC r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if carry (CF=1).</td>
</tr>
<tr>
<td>0F 43 <code>/r</code></td>
<td>CMOVC r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if carry (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 43 <code>/r</code></td>
<td>CMOVC r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if carry (CF=1).</td>
</tr>
<tr>
<td>0F 46 <code>/r</code></td>
<td>CMOVG r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 46 <code>/r</code></td>
<td>CMOVG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 46 <code>/r</code></td>
<td>CMOVG r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 4D <code>/r</code></td>
<td>CMOVGE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 4D <code>/r</code></td>
<td>CMOVGE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4D <code>/r</code></td>
<td>CMOVGE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 4C <code>/r</code></td>
<td>CMOVL r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less (SF=OF).</td>
</tr>
<tr>
<td>0F 4C <code>/r</code></td>
<td>CMOVL r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less (SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4C <code>/r</code></td>
<td>CMOVL r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if less (SF=OF).</td>
</tr>
<tr>
<td>0F 4E <code>/r</code></td>
<td>CMOVLE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>0F 4E <code>/r</code></td>
<td>CMOVLE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4E <code>/r</code></td>
<td>CMOVLE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>0F 42 <code>/r</code></td>
<td>CMOVNA r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 42 <code>/r</code></td>
<td>CMOVNA r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 42 <code>/r</code></td>
<td>CMOVNA r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 43 <code>/r</code></td>
<td>CMOVNAE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 43 <code>/r</code></td>
<td>CMOVNAE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 43 <code>/r</code></td>
<td>CMOVNAE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 47 <code>/r</code></td>
<td>CMOVNBE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Instruction</td>
<td>Op/En</td>
<td>64-Bit Mode</td>
<td>Compat/ Leg Mode</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>-------</td>
<td>-------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0F 47 /r</td>
<td>CMOVNB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 47 /r</td>
<td>CMOVNB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 43 /r</td>
<td>CMOVNC r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 43 /r</td>
<td>CMOVNC r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not carry (CF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 43 /r</td>
<td>CMOVNC r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 45 /r</td>
<td>CMOVNE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 45 /r</td>
<td>CMOVNE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 45 /r</td>
<td>CMOVNE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 4E /r</td>
<td>CMOVNG r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater (ZF=1 or SF≠OF).</td>
</tr>
<tr>
<td>0F 4E /r</td>
<td>CMOVNG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater (ZF=1 or SF≠OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4E /r</td>
<td>CMOVNG r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not greater (ZF=1 or SF≠OF).</td>
</tr>
<tr>
<td>0F 4C /r</td>
<td>CMOVNGE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater or equal (SF≠OF).</td>
</tr>
<tr>
<td>0F 4C /r</td>
<td>CMOVNGE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not greater or equal (SF≠OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4C /r</td>
<td>CMOVNGE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not greater or equal (SF≠OF).</td>
</tr>
<tr>
<td>0F 4D /r</td>
<td>CMOVNL r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 4D /r</td>
<td>CMOVNL r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less (SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4D /r</td>
<td>CMOVNL r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 4F /r</td>
<td>CMOVNLE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 4F /r</td>
<td>CMOVNLE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>REX.W + 0F 4F /r</td>
<td>CMOVNLE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 41 /r</td>
<td>CMOVNO r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not overflow (OF=0).</td>
</tr>
<tr>
<td>0F 41 /r</td>
<td>CMOVNO r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not overflow (OF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 41 /r</td>
<td>CMOVNO r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not overflow (OF=0).</td>
</tr>
<tr>
<td>0F 4B /r</td>
<td>CMOVNP r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not parity (PF=0).</td>
</tr>
<tr>
<td>0F 4B /r</td>
<td>CMOVNP r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not parity (PF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 4B /r</td>
<td>CMOVNP r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not parity (PF=0).</td>
</tr>
<tr>
<td>0F 49 /r</td>
<td>CMOVNS r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not sign (SF=0).</td>
</tr>
<tr>
<td>0F 49 /r</td>
<td>CMOVNS r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not sign (SF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 49 /r</td>
<td>CMOVNS r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not sign (SF=0).</td>
</tr>
<tr>
<td>0F 45 /r</td>
<td>CMOVNZ r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not zero (ZF=0).</td>
</tr>
<tr>
<td>0F 45 /r</td>
<td>CMOVNZ r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not zero (ZF=0).</td>
</tr>
<tr>
<td>REX.W + 0F 45 /r</td>
<td>CMOVNZ r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if not zero (ZF=0).</td>
</tr>
<tr>
<td>0F 40 /r</td>
<td>CMOVOR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if overflow (OF=1).</td>
</tr>
<tr>
<td>0F 40 /r</td>
<td>CMOVOR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if overflow (OF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 40 /r</td>
<td>CMOVOR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if overflow (OF=1).</td>
</tr>
<tr>
<td>0F 4A /r</td>
<td>CMOVPR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity (PF=1).</td>
</tr>
<tr>
<td>0F 4A /r</td>
<td>CMOVPR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity (PF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 4A /r</td>
<td>CMOVPR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if parity (PF=1).</td>
</tr>
<tr>
<td>0F 4A /r</td>
<td>CMOVPE r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity even (PF=1).</td>
</tr>
<tr>
<td>0F 4A /r</td>
<td>CMOVPE r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if parity even (PF=1).</td>
</tr>
<tr>
<td>REX.W + 0F 4A /r</td>
<td>CMOVPE r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move if parity even (PF=1).</td>
</tr>
</tbody>
</table>
Each of the CMOVcc instructions performs a move operation if the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) are in a specified state (or condition). A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

Specifically, CMOVcc loads data from its source operand into a temporary register unconditionally (regardless of the condition code and the status flags in the EFLAGS register). If the condition code associated with the instruction (cc) is satisfied, the data in the temporary register is then copied into the instruction’s destination operand.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOVcc mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOVcc instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```plaintext
temp := SRC
IF condition TRUE
    THEN DEST := temp;
ELSE IF (OperandSize = 32 and IA-32e mode active)
    THEN DEST[63:32] := 0;
FI;
```
Flags Affected
None.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
CMP—Compare Two Operands

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3C ib</td>
<td>CMP AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with AL.</td>
</tr>
<tr>
<td>3D iw</td>
<td>CMP AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm16 with AX.</td>
</tr>
<tr>
<td>3D id</td>
<td>CMP EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm32 with EAX.</td>
</tr>
<tr>
<td>REX.W + 3D id</td>
<td>CMP RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm32 sign-extended to 64-bits with RAX.</td>
</tr>
<tr>
<td>80 /7 lb</td>
<td>CMP r/m8,imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with r/m8.</td>
</tr>
<tr>
<td>REX + 80 /7 lb</td>
<td>CMP r/m8 , imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm8 with r/m8.</td>
</tr>
<tr>
<td>81 /7 iw</td>
<td>CMP r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm16 with r/m16.</td>
</tr>
<tr>
<td>81 /7 id</td>
<td>CMP r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm32 with r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /7 id</td>
<td>CMP r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare imm32 sign-extended to 64-bits with r/m64.</td>
</tr>
<tr>
<td>83 /7 lb</td>
<td>CMP r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with r/m16.</td>
</tr>
<tr>
<td>REX.W + 83 /7 lb</td>
<td>CMP r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare imm8 with r/m32.</td>
</tr>
<tr>
<td>3B /r</td>
<td>CMP r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r8 with r/m8.</td>
</tr>
<tr>
<td>REX + 3B /r</td>
<td>CMP r/m8 , r8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r8 with r/m8.</td>
</tr>
<tr>
<td>39 /r</td>
<td>CMP r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r16 with r/m16.</td>
</tr>
<tr>
<td>39 /r</td>
<td>CMP r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r32 with r/m32.</td>
</tr>
<tr>
<td>REX.W + 39 /r</td>
<td>CMP r/m64, imm8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r64 with r/m64.</td>
</tr>
<tr>
<td>3A /r</td>
<td>CMP r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r/m8 with r8.</td>
</tr>
<tr>
<td>REX + 3A /r</td>
<td>CMP r8 , r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r/m8 with r8.</td>
</tr>
<tr>
<td>3B /r</td>
<td>CMP r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r/m16 with r16.</td>
</tr>
<tr>
<td>3B /r</td>
<td>CMP r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare r/m32 with r32.</td>
</tr>
<tr>
<td>REX.W + 3B /r</td>
<td>CMP r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare r/m64 with r64.</td>
</tr>
</tbody>
</table>

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMsr/m (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMsr/m (r)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX (r)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the Jcc, CMOVcc, and SETcc instructions are based on the results of a CMP instruction. Appendix B, “EFLAGS Condition Codes,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, shows the relationship of the status flags and the condition codes.
In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[
\text{temp} := \text{SRC1} - \text{SignExt}(\text{SRC2});
\]

ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)

**Flags Affected**

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
CMPPD—Compare Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed double precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG C2 /r ib VCMPDD xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed double precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.256.66.0F:WIG C2 /r ib VCMPDD ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed double precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 C2 /r ib VCMPDD k1 (k2), xmm2, xmm3/m128/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed double precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 C2 /r ib VCMPDD k1 (k2), ymm2, ymm3/m256/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed double precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 C2 /r ib VCMPDD k1 (k2), zmm2, zmm3/m512/m64bcst{saes}, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed double precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).
VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

### Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions

<table>
<thead>
<tr>
<th>Predicate</th>
<th>imm8 Value</th>
<th>Description</th>
<th>Result: A &gt; B</th>
<th>A &lt; B</th>
<th>A = B</th>
<th>Unordered</th>
<th>Signals</th>
<th>#IA on QNAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ_OQ (EQ)</td>
<td>0H</td>
<td>Equal (ordered, non-signaling)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>LT_OQ (LT)</td>
<td>1H</td>
<td>Less-than (ordered, signaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>LE_OQ (LE)</td>
<td>2H</td>
<td>Less-than-or-equal (ordered, signaling)</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>UNORD_Q (UNORD)</td>
<td>3H</td>
<td>Unordered (non-signaling)</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>NEQ_UQ (NEQ)</td>
<td>4H</td>
<td>Not-equal (unordered, non-signaling)</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>NLT_US (NLT)</td>
<td>5H</td>
<td>Not-less-than (unordered, non-signaling)</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>NLE_US (NLE)</td>
<td>6H</td>
<td>Not-less-than-or-equal (unordered, signaling)</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>ORD_Q (ORD)</td>
<td>7H</td>
<td>Ordered (non-signaling)</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>EQ_UQ</td>
<td>8H</td>
<td>Equal (unordered, non-signaling)</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>NGE_US (NGE)</td>
<td>9H</td>
<td>Not-greater-than-or-equal (unordered, signaling)</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>NGT_US (NGT)</td>
<td>AH</td>
<td>Not-greater-than (unordered, signaling)</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>FALSE_EQ(FALSE)</td>
<td>8H</td>
<td>False (ordered, non-signaling)</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>NEQ_OQ</td>
<td>CH</td>
<td>Not-equal (ordered, non-signaling)</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>GE_OQ (GE)</td>
<td>DH</td>
<td>Greater-than-or-equal (ordered, signaling)</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>GT_OQ (GT)</td>
<td>EH</td>
<td>Greater-than (ordered, signaling)</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>TRUE_EQ(TRUE)</td>
<td>FH</td>
<td>True (unordered, non-signaling)</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>EQ_OQ</td>
<td>10H</td>
<td>Equal (ordered, signaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>LT_OQ</td>
<td>11H</td>
<td>Less-than (ordered, nonsignaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>LE_OQ</td>
<td>12H</td>
<td>Less-than-or-equal (ordered, nonsignaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>UNORD_S</td>
<td>13H</td>
<td>Unordered (signaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>NEQ_US</td>
<td>14H</td>
<td>Not-equal (unordered, signaling)</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>NLT_UQ</td>
<td>15H</td>
<td>Not-less-than (unordered, nonsignaling)</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>NLE_UQ</td>
<td>16H</td>
<td>Not-less-than-or-equal (unordered, nonsignaling)</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>ORD_S</td>
<td>17H</td>
<td>Ordered (signaling)</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>EQ_US</td>
<td>18H</td>
<td>Equal (unordered, signaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>NGE_UQ</td>
<td>19H</td>
<td>Not-greater-than-or-equal (unordered, nonsignaling)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>No</td>
</tr>
</tbody>
</table>
The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-2. The compiler should treat reserved imm8 values as illegal syntax.

### Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

<table>
<thead>
<tr>
<th>Predicate</th>
<th>imm8 Value</th>
<th>Description</th>
<th>Result: A is 1st Operand, B is 2nd Operand</th>
<th>Signals #IA on QNAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGT_UQ</td>
<td>1AH</td>
<td>Not-greater-than (unordered, nonsignaling)</td>
<td>False True True True True No</td>
<td></td>
</tr>
<tr>
<td>FALSE_OS</td>
<td>1BH</td>
<td>False (ordered, signaling)</td>
<td>False False False False False Yes</td>
<td></td>
</tr>
<tr>
<td>NEQ_OS</td>
<td>1CH</td>
<td>Not-equal (ordered, signaling)</td>
<td>True False False False False Yes</td>
<td></td>
</tr>
<tr>
<td>GE_OQ</td>
<td>1DH</td>
<td>Greater-than-or-equal (ordered, nonsignaling)</td>
<td>True False True False False Yes</td>
<td></td>
</tr>
<tr>
<td>GT_OQ</td>
<td>1EH</td>
<td>Greater-than (ordered, nonsignaling)</td>
<td>True False False False False No</td>
<td></td>
</tr>
<tr>
<td>TRUE_US</td>
<td>1FH</td>
<td>True (unordered, signaling)</td>
<td>True True True True True Yes</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. If either operand A or B is a NAN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with “CPUID.1H:ECX.AVX =0”. See Table 3-2. The compiler should treat reserved imm8 values as illegal syntax.

### Table 3-2. Pseudo-Op and CMPPD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLEPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNLEQPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLEPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDPD xmm1, xmm2</td>
<td>CMPPD xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with “CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-3, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as
illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMP**PD** instructions in a similar fashion by extending the syntax listed in Table 3-3.

**Table 3-3. Pseudo-Op and VCMP**PD** Implementation**

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPL<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLE<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPPUNORD<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPPNEQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPPNL<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPPNLE<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPPORD<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPEQ_UQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPPNEQ_UQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPNGT_UQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMFFALSE_UQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 0BH</td>
</tr>
<tr>
<td>VCMPEQ_OQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 10H</td>
</tr>
<tr>
<td>VCMPLT_OQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 11H</td>
</tr>
<tr>
<td>VCMPLTE_OQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCMPPUNORD_SP<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCMPPNEQ_US<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCMPPNLT_US<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMPPNLE_US<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCMPPORD_SP<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPEQ_US<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCMPLT_OQ<em>PD</em> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCMPLT_OQ<em>PD</em> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPLT_OQ<em>PD</em> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMPPGE_OQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMPPGE_OQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 1EH</td>
</tr>
<tr>
<td>VCMPPGE_OQ<strong>PD</strong> reg1, reg2, reg3</td>
<td>VCMP<strong>PD</strong> reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>
Operation

CASE (COMPARISON PREDICATE) OF

0: OP3 := EQ_Q; OP5 := EQ_Q;

1: OP3 := LT_OS; OP5 := LT_OS;

2: OP3 := LE_OS; OP5 := LE_OS;

3: OP3 := UNORD_Q; OP5 := UNORD_Q;

4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;


7: OP3 := ORD_Q; OP5 := ORD_Q;

8: OP5 := EQ_UQ;

9: OP5 := NGE_US;

10: OP5 := NGT_US;

11: OP5 := FALSE_OQ;

12: OP5 := NEQ_OQ;

13: OP5 := GE_OS;

14: OP5 := GT_OS;

15: OP5 := TRUE_UQ;

16: OP5 := EQ_OS;

17: OP5 := LT_OQ;

18: OP5 := LE_OQ;

19: OP5 := UNORD_S;

20: OP5 := NEQ_US;

21: OP5 := NLT_UQ;

22: OP5 := NLE_UQ;

23: OP5 := ORD_S;

24: OP5 := EQ_US;

25: OP5 := NGE_UQ;

26: OP5 := NGT_UQ;

27: OP5 := FALSE_OS;

28: OP5 := NEQ_OS;

29: OP5 := GE_OQ;

30: OP5 := GT_OQ;

31: OP5 := TRUE_US;

DEFAULT: Reserved;

ESAC;
VCMPPD (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
i := j * 64
IF k2[j] OR *no writemask*
THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
CMP := SRC1[i+63:i] OP5 SRC2[63:0]
ELSE
CMP := SRC1[i+63:i] OP5 SRC2[i+63:i]
FI;
IF CMP = TRUE
THEN DEST[j] := 1;
ELSE DEST[j] := 0; FI;
ELSE DEST[j] := 0 ; zeroing-masking only
FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VCMPPD (VEX.256 Encoded Version)

CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
CMP3 := SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] := 0000000000000000H; FI;
IF CMP2 = TRUE
ELSE DEST[191:128] := 0000000000000000H; FI;
IF CMP3 = TRUE
THEN DEST[255:192] := FFFFFFFFFFFFFFFFH;
ELSE DEST[255:192] := 0000000000000000H; FI;
DEST[MAXVL-1:256] := 0

VCMPPD (VEX.128 Encoded Version)

CMP0 := SRC1[63:0] OP5 SRC2[63:0];
CMP1 := SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] := 0
CMPPD (128-bit Legacy SSE Version)
CMPPD := SRC1[63:0] OP3 SRC2[63:0];
CMPPD := SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] := FFFFFFFF00000000H;
    ELSE DEST[63:0] := 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] := FFFFFFFF00000000H;
    ELSE DEST[127:64] := 0000000000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCMPD __m512 _mm512_cmp_pd_mask( __m512d a, __m512d b, int imm);
VCMPD __m512 _mm512_cmp_round_pd_mask( __m512d a, __m512d b, int imm, int sae);
VCMPD __m512 _mm512_mask_cmp_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm);
VCMPD __m512 _mm512_mask_cmp_round_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPD __mm256 _mm256_cmp_pd_mask( __m256d a, __m256d b, int imm);
VCMPD __mm256 _mm256_mask_cmp_pd_mask( __mmask8 k1, __m256d a, __m256d b, int imm);
VCMPD __mm128 _mm128_cmp_pd_mask( __m128d a, __m128d b, int imm);
(V)CMPPD __m128 _mm128_cmp_pd(__m128d a, __m128d b, int imm)

SIMD Floating-Point Exceptions
Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
CMPPS—Compare Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C2 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare packed single precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.128.0F.W1 C2 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed single precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>VEX.256.0F.W1 C2 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed single precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed single precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed single precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed single precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

**EVEX encoded versions:** The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

**VEX.256 encoded version:** The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

**128-bit Legacy SSE version:** The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).
VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instructions encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater-than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-4. The compiler should treat reserved imm8 values as illegal syntax.

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLEPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNEQPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLEPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDPS xmm1, xmm2</td>
<td>CMPPS xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1” implement the full complement of 32 predicates shown in Table 3-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.
<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPPS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPLTPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPUNORDPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPEQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPLTPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPLEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPORDPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPNGEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPNGTPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMPCFALSEPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0BH</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0CH</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0DH</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 0FH</td>
</tr>
<tr>
<td>VCMPEQ_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 10H</td>
</tr>
<tr>
<td>VCMPLT_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 11H</td>
</tr>
<tr>
<td>VCMPLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCMRORDER_SPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCMPEQ_USPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCMPLT_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCMRORDER_SPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPEQ_USPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCMPNG_UQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCMPNGTPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPCFALSE_USPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMNLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMNLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1DH</td>
</tr>
<tr>
<td>VCMNLE_OQPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1EH</td>
</tr>
<tr>
<td>VCMTRUE_USPS reg1, reg2, reg3</td>
<td>VCMPPS reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP3 := EQ_OQ; OP5 := EQ_OQ;
  1: OP3 := LT_OQ; OP5 := LT_OQ;
  2: OP3 := LE_OQ; OP5 := LE_OQ;
  3: OP3 := UNORD_Q; OP5 := UNORD_Q;
  4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
  7: OP3 := ORD_Q; OP5 := ORD_Q;
  8: OP5 := EQ_UQ;
  9: OP5 := NGE_US;
 10: OP5 := NGT_US;
 11: OP5 := FALSE_OQ;
 12: OP5 := NEQ_OQ;
 13: OP5 := GE_OQ;
 14: OP5 := GT_OQ;
 15: OP5 := TRUE_UQ;
 16: OP5 := EQ_OQ;
 17: OP5 := LT_OQ;
 18: OP5 := LE_OQ;
 19: OP5 := UNORD_S;
 20: OP5 := NEQ_US;
 21: OP5 := NLT_UQ;
 22: OP5 := NLE_UQ;
 23: OP5 := ORD_S;
 24: OP5 := EQ_US;
 25: OP5 := NGE_UQ;
 26: OP5 := NGT_UQ;
 27: OP5 := FALSE_OS;
 28: OP5 := NEQ_OS;
 29: OP5 := GE_OQ;
 30: OP5 := GT_OQ;
 31: OP5 := TRUE_US;
DEFAULT: Reserved

ESAC;
VCMPPS (EVEX Encoded Versions)

(\(KL, VL\)) = (4, 128), (8, 256), (16, 512)

FOR \(j := 0\) TO \(KL-1\)
  \(i := j \times 32\)
  IF \(k2[j]\) OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          CMP := SRC1\([i+31:i]\) OP5 SRC2\([31:0]\)
        ELSE
          CMP := SRC1\([i+31:i]\) OP5 SRC2\([i+31:i]\)
        FI;
        IF CMP = TRUE
          THEN DEST\([j]\) := 1;
          ELSE DEST\([j]\) := 0; FI;
        ELSE DEST\([j]\) := 0 ; zeroing-masking onlyFI;
    FI;
  ENDFOR

DEST\([\text{MAX}_KL-1:KL]\) := 0

VCMPPS (VEX.256 Encoded Version)

CMP0 := SRC1\([31:0]\) OP5 SRC2\([31:0]\);
CMP1 := SRC1\([63:32]\) OP5 SRC2\([63:32]\);
CMP2 := SRC1\([95:64]\) OP5 SRC2\([95:64]\);
CMP3 := SRC1\([127:96]\) OP5 SRC2\([127:96]\);
CMP4 := SRC1\([159:128]\) OP5 SRC2\([159:128]\);
CMP5 := SRC1\([191:160]\) OP5 SRC2\([191:160]\);
CMP6 := SRC1\([223:192]\) OP5 SRC2\([223:192]\);
CMP7 := SRC1\([255:224]\) OP5 SRC2\([255:224]\);

IF CMP0 = TRUE
  THEN DEST\([31:0]\) := FFFFFFFFH;
  ELSE DEST\([31:0]\) := 000000000H; FI;
IF CMP1 = TRUE
  THEN DEST\([63:32]\) := FFFFFFFFH;
  ELSE DEST\([63:32]\) := 000000000H; FI;
IF CMP2 = TRUE
  THEN DEST\([95:64]\) := FFFFFFFFH;
  ELSE DEST\([95:64]\) := 000000000H; FI;
IF CMP3 = TRUE
  THEN DEST\([127:96]\) := FFFFFFFFH;
  ELSE DEST\([127:96]\) := 000000000H; FI;
IF CMP4 = TRUE
  THEN DEST\([159:128]\) := FFFFFFFFH;
  ELSE DEST\([159:128]\) := 000000000H; FI;
IF CMP5 = TRUE
  THEN DEST\([191:160]\) := FFFFFFFFH;
  ELSE DEST\([191:160]\) := 000000000H; FI;
IF CMP6 = TRUE
  THEN DEST\([223:192]\) := FFFFFFFFH;
  ELSE DEST\([223:192]\) := 000000000H; FI;
IF CMP7 = TRUE
  THEN DEST\([255:224]\) := FFFFFFFFH;
  ELSE DEST\([255:224]\) := 000000000H; FI;
DEST\([\text{MAXVL}-1:256]\) := 0
VCMPPS (VEX.128 Encoded Version)
CMPO := SRC1[31:0] OP5 SRC2[31:0];
CMPI := SRC1[63:32] OP5 SRC2[63:32];
CMPI := SRC1[95:64] OP5 SRC2[95:64];
CMPI := SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
   THEN DEST[31:0] := FFFFFFFFH;
   ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
   THEN DEST[63:32] := FFFFFFFFH;
   ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
   THEN DEST[95:64] := FFFFFFFFH;
   ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
   THEN DEST[127:96] := FFFFFFFFH;
   ELSE DEST[127:96] := 000000000H; FI;
DEST[MAXVL-1:128] := 0

CMPPS (128-bit Legacy SSE Version)
CMPO := SRC1[31:0] OP3 SRC2[31:0];
CMPI := SRC1[95:64] OP3 SRC2[95:64];
CMPI := SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
   THEN DEST[31:0] := FFFFFFFFH;
   ELSE DEST[31:0] := 000000000H; FI;
IF CMP1 = TRUE
   THEN DEST[63:32] := FFFFFFFFH;
   ELSE DEST[63:32] := 000000000H; FI;
IF CMP2 = TRUE
   THEN DEST[95:64] := FFFFFFFFH;
   ELSE DEST[95:64] := 000000000H; FI;
IF CMP3 = TRUE
   THEN DEST[127:96] := FFFFFFFFH;
   ELSE DEST[127:96] := 000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCMPPS __m512 _mm512_cmp_ps( __m512 a, __m512 b, int imm);
VCMPPS __m512 _mm512_cmp_round_ps( __m512 a, __m512 b, int imm, int sae);
VCMPPS __m512 _mm512_mask_cmp_ps( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __m512 _mm512_mask_cmp_round_ps( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPS __m256 _mm256_cmp_ps( __m256 a, __m256 b, int imm);
VCMPPS __m256 _mm256_mask_cmp_ps( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __m256 _mm256_mask_cmp_round_ps( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __m128 _mm128_cmp_ps( __m128 a, __m128 b, int imm);
VCMPPS __m128 _mm128_mask_cmp_ps( __mmask8 k1, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions
Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1, Denormal.
Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
**CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6</td>
<td>CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPS m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPS m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R</td>
</tr>
<tr>
<td>REX.W + A7</td>
<td>CMPS m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compares quadword at address (R</td>
</tr>
<tr>
<td>A6</td>
<td>CMPSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R</td>
</tr>
<tr>
<td>A7</td>
<td>CMPSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R</td>
</tr>
<tr>
<td>REX.W + A7</td>
<td>CMPSQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compares quadword at address (R</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct loca-
Location of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement. The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPZ/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```plaintext
temp := SRC1 - SRC2;
SetStatusFlags(temp);
IF (64-Bit Mode) THEN
  IF (Byte comparison) THEN IF DF = 0 THEN
      (R|E)SI := (R|E)SI + 1;
      (R|E)DI := (R|E)DI + 1;
    ELSE
      (R|E)SI := (R|E)SI - 1;
      (R|E)DI := (R|E)DI - 1;
    FI;
  ELSE IF (Word comparison) THEN IF DF = 0 THEN
      (R|E)SI := (R|E)SI + 2;
      (R|E)DI := (R|E)DI + 2;
    ELSE
      (R|E)SI := (R|E)SI - 2;
      (R|E)DI := (R|E)DI - 2;
    FI;
  ELSE IF (Doubleword comparison) THEN IF DF = 0 THEN
      (R|E)SI := (R|E)SI + 4;
      (R|E)DI := (R|E)DI + 4;
    ELSE
      (R|E)SI := (R|E)SI - 4;
      (R|E)DI := (R|E)DI - 4;
    FI;
```
ELSE (* Quadword comparison *)
  THEN IF DF = 0
    (R|E)SI := (R|E)SI + 8;
    (R|E)DI := (R|E)DI + 8;
  ELSE
    (R|E)SI := (R|E)SI - 8;
    (R|E)DI := (R|E)DI - 8;
  FI;
FL;
ELSE (* Non-64-bit Mode *)
  IF (byte comparison)
    THEN IF DF = 0
      (E)SI := (E)SI + 1;
      (E)DI := (E)DI + 1;
    ELSE
      (E)SI := (E)SI - 1;
      (E)DI := (E)DI - 1;
    FI;
  ELSE IF (Word comparison)
    THEN IF DF = 0
      (E)SI := (E)SI + 2;
      (E)DI := (E)DI + 2;
    ELSE
      (E)SI := (E)SI - 2;
      (E)DI := (E)DI - 2;
    FI;
  ELSE (* Doubleword comparison *)
    THEN IF DF = 0
      (E)SI := (E)SI + 4;
      (E)DI := (E)DI + 4;
    ELSE
      (E)SI := (E)SI - 4;
      (E)DI := (E)DI - 4;
    FI;
FL;

Flags Affected
The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**CMPSD—Compare Scalar Double Precision Floating-Point Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare low double precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 C2 /r ib VCMPSD k1 (k2), xmm2, xmm3/m64{sa}, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low double precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Compares the low double precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX =0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater-than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either
by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-6. The compiler should treat reserved imm8 values as illegal syntax.

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

### Table 3-6. Pseudo-Op and CMPSD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLESD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNEGQSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTSQD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLEGSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDSD xmm1, xmm2</td>
<td>CMPSD xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

### Table 3-7. Pseudo-Op and VCMPSD Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMPLQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPNLEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPLTSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMPLNSEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMPLNLEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMPLNLEQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPEQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMPNLEQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMPLNLEQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMPLT_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, OBH</td>
</tr>
<tr>
<td>VCMPLEQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, OCH</td>
</tr>
<tr>
<td>VCMPLQ_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, ODH</td>
</tr>
</tbody>
</table>
Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Table 3-7. Pseudo-Op and VCMPSD Implementation**

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSD Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPGTS_D reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPTRU_ESD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 0FH</td>
</tr>
<tr>
<td>VCMPEQ_OSSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 10H</td>
</tr>
<tr>
<td>VCMPLT_OQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 11H</td>
</tr>
<tr>
<td>VCMPLE_OQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCMPU_NORD_SSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCMPNLE_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCMPNLT_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMPNLE_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCMPORD_SSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPEQ_USSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCMPNGE_QSSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCMPNGT_UQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPFALSE_OSSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMPEQ_OSSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMPGT_OQSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1DH</td>
</tr>
<tr>
<td>VCMPTFALSE_USSD reg1, reg2, reg3</td>
<td>VCMPSD reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>

**Operation**

CASE (COMPARISON PREDICATE) OF
  0: OP3 := EQ_OQ; OP5 := EQ_OQ;
  1: OP3 := LT_OS; OP5 := LT_OS;
  2: OP3 := LE_OS; OP5 := LE_OS;
  3: OP3 := UNORD_Q; OP5 := UNORD_Q;
  4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
  7: OP3 := ORD_Q; OP5 := ORD_Q;
  8: OP5 := EQ_UQ;
  9: OP5 := NGE_US;
 10: OP5 := NGT_US;
 11: OP5 := FALSE_OQ;
 12: OP5 := NEQ_OQ;
 13: OP5 := GE_OS;
 14: OP5 := GT_OQ;
 15: OP5 := TRUE_UQ;
 16: OP5 := EQ_OS;
 17: OP5 := LT_OQ;
 18: OP5 := LE_OQ;
 19: OP5 := UNORD_S;
 20: OP5 := NEQ_US;
 21: OP5 := NLT_UQ;
22: OP5 := NLE_UQ;
23: OP5 := ORD_S;
24: OP5 := EQ_US;
25: OP5 := NGE_UQ;
26: OP5 := NGT_UQ;
27: OP5 := FALSE_OS;
28: OP5 := NEQ_OS;
29: OP5 := GE_OQ;
30: OP5 := GT_OQ;
31: OP5 := TRUE_US;
DEFAULT: Reserved

ESAC;

VCMPSD (EVEX Encoded Version)
CMPO := SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN IF CMPO = TRUE
        THEN DEST[0] := 1;
        ELSE DEST[0] := 0; FI;
    ELSE DEST[0] := 0 ; zeroing-masking only
    FI;
DEST[MAX_KL-1:1] := 0

CMPSD (128-bit Legacy SSE Version)
CMPO := DEST[63:0] OP3 SRC[63:0];
IF CMPO = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[MAXVL-1:64] (Unmodified)

VCMPSD (VEX.128 Encoded Version)
CMPO := SRC1[63:0] OP5 SRC2[63:0];
IF CMPO = TRUE
THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0000000000000000H; FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCMPSD __mmask8 __m128d __mm_cmp_sd_mask( __m128d a, __m128d b, int imm);
VCMPSD __mmask8 __m128d __mm_cmp_round_sd_mask( __m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 __m128d __mm_mask_cmp_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 __m128d __mm_mask_cmp_round_sd_mask( __mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d _mm_cmp_sd(__m128d a, __m128d b, const int imm)

SIMD Floating-Point Exceptions
Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
CMPSS—Compare Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF C2 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare low single precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>VEX.LI.G3.0F.WIG C2 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.</td>
</tr>
<tr>
<td>EVEX.LI.G3.0F.W0 C2 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low single precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

Compares the low single precision floating-point values in the second source operand and the first source operand and returns the result of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or a 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 127:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.
Note that processors with “CPUID.1H:ECX.AVX = 0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater-than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with “CPUID.1H:ECX.AVX = 0”. See Table 3-8. The compiler should treat reserved imm8 values as illegal syntax.

### Table 3-8. Pseudo-Op and CMPSS Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPEQSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 0</td>
</tr>
<tr>
<td>CMPLTSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 1</td>
</tr>
<tr>
<td>CMPLESS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 2</td>
</tr>
<tr>
<td>CMPUNORDSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 3</td>
</tr>
<tr>
<td>CMPNEQSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 4</td>
</tr>
<tr>
<td>CMPNLTSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 5</td>
</tr>
<tr>
<td>CMPNLESS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 6</td>
</tr>
<tr>
<td>CMPORDSS xmm1, xmm2</td>
<td>CMPSS xmm1, xmm2, 7</td>
</tr>
</tbody>
</table>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 3-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. The compiler should treat reserved imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 3-9.

### Table 3-9. Pseudo-Op and VCMPSS Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>VCMPSS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VCMLTSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VCMPLESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VCMPUNORDSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 3</td>
</tr>
<tr>
<td>VCMPEQ_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VCMNLEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VCMNLESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 6</td>
</tr>
<tr>
<td>VCMORDSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 7</td>
</tr>
<tr>
<td>VCMPEQ_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 8</td>
</tr>
<tr>
<td>VCMNLEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 9</td>
</tr>
<tr>
<td>VCMNNGESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0AH</td>
</tr>
<tr>
<td>VCMNNGTSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0BH</td>
</tr>
</tbody>
</table>

CMPSS—Compare Scalar Single Precision Floating-Point Value
Table 3-9. Pseudo-Op and VCMPSS Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>CMPSS Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCMPNEQ_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, OCH</td>
</tr>
<tr>
<td>VCMPGESS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0DH</td>
</tr>
<tr>
<td>VCPGTEQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0EH</td>
</tr>
<tr>
<td>VCMPTRUESSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 0FH</td>
</tr>
<tr>
<td>VCMPGT_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 10H</td>
</tr>
<tr>
<td>VCMPLT_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 11H</td>
</tr>
<tr>
<td>VCMPEQ_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 12H</td>
</tr>
<tr>
<td>VCPMUNORD_SSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 13H</td>
</tr>
<tr>
<td>VCPMNEQ_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 14H</td>
</tr>
<tr>
<td>VCPMLT_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 15H</td>
</tr>
<tr>
<td>VCMNLE_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 16H</td>
</tr>
<tr>
<td>VCPMORD_SSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 17H</td>
</tr>
<tr>
<td>VCMPEQ_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 18H</td>
</tr>
<tr>
<td>VCPMGT_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 19H</td>
</tr>
<tr>
<td>VCPMGT_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1AH</td>
</tr>
<tr>
<td>VCMPEQ_USSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1BH</td>
</tr>
<tr>
<td>VCMPEQ_OQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1CH</td>
</tr>
<tr>
<td>VCMPPG_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1DH</td>
</tr>
<tr>
<td>VCMPPGT_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1EH</td>
</tr>
<tr>
<td>VCMPPG_UQSS reg1, reg2, reg3</td>
<td>VCMPSS reg1, reg2, reg3, 1FH</td>
</tr>
</tbody>
</table>

Software should ensure VCMPSS is encoded with VEX.L=0. Encoding VCMPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

CASE (COMPARISON PREDICATE) OF
0: OP3 := EQ_OQ; OP5 := EQ_OQ;
1: OP3 := LT_OQ; OP5 := LT_OQ;
2: OP3 := LE_OQ; OP5 := LE_OQ;
3: OP3 := UNORD_OQ; OP5 := UNORD_OQ;
4: OP3 := NEQ_UQ; OP5 := NEQ_UQ;
7: OP3 := ORD_OQ; OP5 := ORD_OQ;
8: OP5 := EQ_UQ;
9: OP5 := NGE_US;
10: OP5 := NGT_US;
11: OP5 := FALSE_OQ;
12: OP5 := NEQ_OQ;
13: OP5 := GE_OQ;
14: OP5 := GT_OQ;
15: OP5 := TRUE_UQ;
16: OP5 := EQ_OS;
17: OP5 := LT_OQ;
18: OP5 := LE_OQ;
19: OP5 := UNORD_S;
20: OP5 := NEQ_US;
21: OP5 := NLT_UQ;
22: OP5 := NLE_UQ;
23: OP5 := ORD_S;
24: OP5 := EQ_US;
25: OP5 := NGE_UQ;
26: OP5 := NGT_UQ;
27: OP5 := FALSE_OS;
28: OP5 := NEQ_OS;
29: OP5 := GE_OQ;
30: OP5 := GT_OQ;
31: OP5 := TRUE_US;
DEFAULT: Reserved

ESAC;

VCMPSS (EVEX Encoded Version)
CMP0 := SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
    THEN IF CMP0 = TRUE
        THEN DEST[0] := 1;
        ELSE DEST[0] := 0; Fl;
    ELSE DEST[0] := 0 ; zeroing-masking only
Fl;
DEST[MAX_KL-1:1] := 0

CMPSS (128-bit Legacy SSE Version)
CMP0 := DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; Fl;
DEST[MAXVL-1:32] (Unmodified)

VCMPSS (VEX.128 Encoded Version)
CMP0 := SRC1[31:0] OP5 SRC2[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 00000000H; Fl;
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, int imm);
VCMPSS __m128 _mm_cmp_round_ss(__m128 a, __m128 b, int imm, int sae);
VCMPSS __m128 _mm_mask_cmp_ss(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __m128 _mm_mask_cmp_round_ss(__mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions
Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.
Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
CMPXCHG—Compare and Exchange

**Description**

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor’s bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**IA-32 Architecture Compatibility**

This instruction is not supported on Intel processors earlier than the Intel486 processors.

**Operation**

(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed *)

```
TEMP := DEST
IF accumulator = TEMP
    THEN
        ZF := 1;
        DEST := SRC;
    ELSE
        ZF := 0;
        accumulator := TEMP;
        DEST := TEMP;
FI;
```
Flags Affected
The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

### Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64 bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor’s bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-243.

### IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.
Operation
IF (64-Bit Mode and OperandSize = 64)
  THEN
    TEMP128 := DEST
    IF (RDX:RAX = TEMP128)
      THEN
        ZF := 1;
        DEST := RCX:RBX;
      ELSE
        ZF := 0;
        RDX:RAX := TEMP128;
        DEST := TEMP128;
      FI;
    FI
  ELSE
    TEMP64 := DEST;
    IF (EDX:EAX = TEMP64)
      THEN
        ZF := 1;
        DEST := ECX:EBX;
      ELSE
        ZF := 0;
        EDX:EAX := TEMP64;
        DEST := TEMP64;
      FI;
    FI;
FI;

Flags Affected
The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected Mode Exceptions
#UD If the destination is not a memory operand.
#GP(0) If the destination is located in a non-writable segment.
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions
#UD If the destination operand is not a memory location.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
Virtual-8086 Mode Exceptions

- #UD If the destination operand is not a memory location.
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
  If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary.
  If CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0.
- #UD If the destination operand is not a memory location.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
### COMISD—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F.WIG 2F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F.W1 2F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Compares the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

#### Operation

COMISD (All Versions)

RESULT := OrderedCompare([DEST][63:0] <> [SRC][63:0]) { (* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF := 111;
  GREATER_THAN: ZF,PF,CF := 000;
  LESS_THAN: ZF,PF,CF := 001;
  EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0;}
Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd(__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd(__m128d a, __m128d b)
VCOMISD int _mm_comile_sd(__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd(__m128d a, __m128d b)
VCOMISD int _mm_comige_sd(__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd(__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions
Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 111B.
COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2F /r COMISS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>VEX.LIG.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>EVEX.LIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Compares the single precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid operation exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

COMISS (All Versions)

```plaintext
RESULT := OrderedCompare(Dest[31:0] <> Src[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF := 111;
  GREATER_THAN: ZF,PF,CF := 000;
  LESS_THAN: ZF,PF,CF := 001;
  EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
```
**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comile_ss (__m128 a, __m128 b)
VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comige_ss (__m128 a, __m128 b)
VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)

**SIMD Floating-Point Exceptions**

Invalid (if SNaN or QNaN operands), Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
**CPUID—CPU Identification**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F A2</td>
<td>CPUID</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The instruction’s output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using some Intel processors, the following is true:

- CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
- CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
- CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
- CPUID.EAX = 1FH (* Returns V2 Extended Topology Enumeration leaf. *)
- CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
- CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

**See also:**

"Serializing Instructions” in Chapter 9, “Multiple-Processor Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

"Caching Translation Information” in Chapter 4, “Paging,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

---

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.
2. CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of CPUID leaf 1FH before using leaf 0BH.
### Table 3-8. Information Returned by CPUID Instruction

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic CPUID Information</strong></td>
<td></td>
</tr>
<tr>
<td>0H EAX</td>
<td>Maximum Input Value for Basic CPUID Information.</td>
</tr>
<tr>
<td>EBX</td>
<td>“Genu”</td>
</tr>
<tr>
<td>ECX</td>
<td>“Intel”</td>
</tr>
<tr>
<td>EDX</td>
<td>“Intel”</td>
</tr>
<tr>
<td>01H EAX</td>
<td>Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 07-00: Brand Index.</td>
</tr>
<tr>
<td></td>
<td>Bits 15-08: CLFLUSH line size (Value $\times 8 = \text{cache line size in bytes}$; used also by CLFLUSHOPT).</td>
</tr>
<tr>
<td></td>
<td>Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-24: Initial APIC ID**.</td>
</tr>
<tr>
<td>ECX</td>
<td>Feature Information (see Figure 3-7 and Table 3-10).</td>
</tr>
<tr>
<td>EDX</td>
<td>Feature Information (see Figure 3-8 and Table 3-11).</td>
</tr>
<tr>
<td><strong>NOTES:</strong></td>
<td></td>
</tr>
<tr>
<td>* The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.</td>
<td></td>
</tr>
<tr>
<td>** The 8-bit initial APIC ID in EBX[31:24] is replaced by the 32-bit x2APIC ID, available in Leaf 0BH and Leaf 1FH.</td>
<td></td>
</tr>
<tr>
<td>02H EAX</td>
<td>Cache and TLB Information (see Table 3-12).</td>
</tr>
<tr>
<td>EBX</td>
<td>Cache and TLB Information.</td>
</tr>
<tr>
<td>ECX</td>
<td>Cache and TLB Information.</td>
</tr>
<tr>
<td>EDX</td>
<td>Cache and TLB Information.</td>
</tr>
<tr>
<td>03H EAX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 00-31 of 96-bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 32-63 of 96-bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)</td>
</tr>
<tr>
<td><strong>NOTES:</strong></td>
<td></td>
</tr>
<tr>
<td>Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.</td>
<td></td>
</tr>
<tr>
<td>CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.</td>
<td></td>
</tr>
</tbody>
</table>

### Deterministic Cache Parameters Leaf (Initial EAX Value = 04H)

<table>
<thead>
<tr>
<th>EAX</th>
<th>Notes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf 04H output depends on the initial value in ECX.*</td>
<td></td>
</tr>
<tr>
<td>See also: &quot;INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level&quot; on page 251.</td>
<td></td>
</tr>
<tr>
<td>Bits 04-00: Cache Type Field.</td>
<td></td>
</tr>
<tr>
<td>0 = Null - No more caches.</td>
<td></td>
</tr>
<tr>
<td>1 = Data Cache.</td>
<td></td>
</tr>
<tr>
<td>2 = Instruction Cache.</td>
<td></td>
</tr>
<tr>
<td>3 = Unified Cache.</td>
<td></td>
</tr>
<tr>
<td>4-31 = Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| 05H               | Bits 07-05: Cache Level (starts at 1).  
|                   | Bit 08: Self Initializing cache level (does not need SW initialization). 
|                   | Bit 09: Fully Associative cache.  
|                   | Bits 13-10: Reserved. 
|                   | Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache**.  
|                   | Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package**.  
|                   | ** Add one to the return value to get the result.  
|                   | *** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache.  
|                   | **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.  
|                   | ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.  
|                   | NOTES: 
|                   | * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0.  
|                   | ** Add one to the return value to get the result.  
|                   | *** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache.  
|                   | **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.  
|                   | ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.  

<table>
<thead>
<tr>
<th>MONITOR/MWait Leaf (Initial EAX Value = 05H)</th>
</tr>
</thead>
</table>
| 05H                                        | EAX | Bits 15-00: Smallest monitor-line size in bytes (default is processor’s monitor granularity). 
|                                             |     | Bits 31-16: Reserved = 0.  
|                                             | EBX | Bits 15-00: Largest monitor-line size in bytes (default is processor’s monitor granularity). 
|                                             |     | Bits 31-16: Reserved = 0.  
|                                             | ECX | Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported. 
|                                             |     | Bit 01: Supports treating interrupts as break-event for MWait, even when interrupts disabled. 
|                                             |     | Bits 31-02: Reserved.  
|                                             | EDX | Bits 03-00: Number of C0* sub C-states supported using MWait. 
|                                             |     | Bits 07-04: Number of C1* sub C-states supported using MWait. 
|                                             |     | Bits 11-08: Number of C2* sub C-states supported using MWait. 
|                                             |     | Bits 15-12: Number of C3* sub C-states supported using MWait. 
|                                             |     | Bits 19-16: Number of C4* sub C-states supported using MWait. 
|                                             |     | Bits 23-20: Number of C5* sub C-states supported using MWait. 
|                                             |     | Bits 27-24: Number of C6* sub C-states supported using MWait. 
|                                             |     | Bits 31-28: Number of C7* sub C-states supported using MWait.  

CPUD—CPU Identification
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTE:</strong></td>
<td>* The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.</td>
</tr>
<tr>
<td><strong>EBX</strong></td>
<td>Bits 03-00: Number of Interrupt Thresholds in Digital Thermal Sensor.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-04: Reserved.</td>
</tr>
<tr>
<td><strong>ECX</strong></td>
<td>Bits 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency.</td>
</tr>
<tr>
<td></td>
<td>Bits 02-01: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>Bits 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H).</td>
</tr>
<tr>
<td></td>
<td>Bits 07-04: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>Bits 15-08: Number of Intel® Thread Director classes supported by the processor. Information for that many classes is written into the Intel Thread Director Table by the hardware.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-16: Reserved = 0.</td>
</tr>
</tbody>
</table>

**THERMAL AND POWER MANAGEMENT LEAF (INITIAL EAX VALUE = 06H)**

06H  EAX  Bit 00: Digital temperature sensor is supported if set.  
      Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]).  
      Bit 02: ARAT. APIC-Timer-always-running feature is supported if set.  
      Bit 03: Reserved.  
      Bit 04: PLN. Power limit notification controls are supported if set.  
      Bit 05: ECMD. Clock modulation duty cycle extension is supported if set.  
      Bit 06: PTM. Package thermal management is supported if set.  
      Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set.  
      Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set.  
      Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set.  
      Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set.  
      Bit 12: Reserved.  
      Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREADSTALL MSRs are supported if set.  
      Bit 14: Intel® Turbo Boost Max Technology 3.0 available.  
      Bit 15: HWP Capabilities. Highest Performance change is supported if set.  
      Bit 16: HWP PECI override is supported if set.  
      Bit 17: Flexible Hwp is supported if set.  
      Bit 18: Fast access mode for the IA32_HWP_REQUEST MSR is supported if set.  
      Bit 19: Hw_FEEDBACK, IA32_Hw_FEEDBACK_PTR MSR, IA32_Hw_FEEDBACK_CONFIG MSR, IA32_PACKAGE_THERM_STATUS MSR bit 26, and IA32_PACKAGE_THERM_INTERRUPT MSR bit 25 are supported if set.  
      Bit 20: Ignoring Idle Logical Processor Hwp request is supported if set.  
      Bits 22-21: Reserved.  
      Bit 23: Intel® Thread Director supported if set. IA32_Hw_FEEDBACK_CHAR and IA32_Hw_FEEDBACK_THREAD_CONFIG MSRs are supported if set.  
      Bit 24: IA32_THERM_INTERRUPT MSR bit 25 is supported if set.  
      Bits 31-25: Reserved.  

**EBX Bits 03-00:** Number of Interrupt Thresholds in Digital Thermal Sensor.

**ECX Bit 00:** Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency.

**ECX Bits 02-01:** Reserved = 0.

**ECX Bit 03:** The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H).

**ECX Bits 07-04:** Reserved = 0.

**ECX Bits 15-08:** Number of Intel® Thread Director classes supported by the processor. Information for that many classes is written into the Intel Thread Director Table by the hardware.

**ECX Bits 31-16:** Reserved = 0.
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| EDX               | Bits 07-00: Bitmap of supported hardware feedback interface capabilities.  
|                   | 0 = When set to 1, indicates support for performance capability reporting.  
|                   | 1 = When set to 1, indicates support for energy efficiency capability reporting.  
|                   | 2-7 = Reserved  
|                   | Bits 11-08: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages; add one to the return value to get the result.  
|                   | Bits 31-16: Index (starting at 0) of this logical processor’s row in the hardware feedback interface structure. Note that on some parts the index may be same for multiple logical processors. On some parts the indices may not be contiguous, i.e., there may be unused rows in the hardware feedback interface structure.  
|                   | **NOTE:**  
|                   | Bits 0 and 1 will always be set together. |

Structured Extended Feature Flags Enumeration Leaf (Initial EAX Value = 07H, ECX = 0)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| 07H               | Bits 31-00: Reports the maximum input value for supported leaf 7 sub-leaves.  
| EBX               | Bit 00: FSGSBASE. Supports RDWSFBASE/RDGSBASE/WRFWSBASE/WRGWSBASE if 1.  
|                   | Bit 01: IA32_TSC_ADJUST MSR is supported if 1.  
|                   | Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1.  
|                   | Bit 03: BMI1.  
|                   | Bit 04: HLE.  
|                   | Bit 05: AVX2. Supports Intel® Advanced Vector Extensions 2 (Intel® AVX2) if 1.  
|                   | Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1.  
|                   | Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1.  
|                   | Bit 08: BMI2.  
|                   | Bit 09: Supports Enhanced REP MOVSB/STOSB if 1.  
|                   | Bit 10: INVPID. If 1, supports INVPID instruction for system software that manages process-context identifiers.  
|                   | Bit 11: RTM.  
|                   | Bit 13: Deprecated FPU CS and FPU DS values if 1.  
|                   | Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1.  
|                   | Bit 16: AVX512F.  
|                   | Bit 17: AVX512DQ.  
|                   | Bit 18: RDSEED.  
|                   | Bit 19: ADX.  
|                   | Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1.  
|                   | Bit 21: AVX512_IFMA.  
|                   | Bit 22: Reserved.  
|                   | Bit 23: CLFLUSHOPT.  
|                   | Bit 24: CLWB.  
|                   | Bit 25: Intel Processor Trace.  
|                   | Bit 26: AVX512PF. (Intel® Xeon Phi™ only.)  
|                   | Bit 27: AVX512ER. (Intel® Xeon Phi™ only.)  
|                   | Bit 28: AVX512CD.  
|                   | Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1.  
|                   | Bit 30: AVX512BW.  
<p>|                   | Bit 31: AVX512VL.  |</p>
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| ECX               | Bit 00: PREFETCHWT1. (Intel® Xeon Phi™ only.)
|                   | Bit 01: AVX512_VBMI.
|                   | Bit 02: UMIP. Supports user-mode instruction prevention if 1.
|                   | Bit 03: PKU. Supports protection keys for user-mode pages if 1.
|                   | Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/wRPKRU instructions).
|                   | Bit 05: WAITPKG.
|                   | Bit 06: AVX512_VBMI2.
|                   | Bit 07: CET_SS. Supports CET shadow stack features if 1. Processors that set this bit define bits 1:0 of the IA32_U_CET and IA32_S_CET MSRs. Enumerates support for the following MSRs: IA32_INTERRUPT_SPP_TABLE_ADDR, IA32_PL3_SSP, IA32_PL2_SSP, IA32_PL1_SSP, and IA32_PL0_SSP.
|                   | Bit 08: GFNI.
|                   | Bit 09: VAES.
|                   | Bit 10: VPCLMULQDQ.
|                   | Bit 11: AVX512_VNNI.
|                   | Bit 12: AVX512_BITALG.
|                   | Bits 13: TME_EN. If 1, the following MSRs are supported: IA32_TME_CAPABILITY, IA32_TME_ACTIVATE, IA32_TME_EXCLUDE_MASK, and IA32_TME_EXCLUDE_BASE.
|                   | Bit 14: AVX512_VPOPCNTDQ.
|                   | Bit 15: Reserved.
|                   | Bit 16: IA57. Supports 57-bit linear addresses and five-level paging if 1.
|                   | Bits 21-17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode.
|                   | Bit 22: RDPID and IA32_TSC_AUX are available if 1.
|                   | Bit 23: KL. Supports Key Locker if 1.
|                   | Bit 24: BUS_LOCK_DETECT. If 1, indicates support for OS bus-lock detection.
|                   | Bit 25: CLDEMOTE. Supports cache line demote if 1.
|                   | Bit 26: Reserved.
|                   | Bit 27: MOVDIRI. Supports MOVDIRI if 1.
|                   | Bit 28: MOVDIR64B. Supports MOVDIR64B if 1.
|                   | Bit 29: ENQCMD. Supports Enqueue Stores if 1.
|                   | Bit 30: SGX_LC. Supports SGX Launch Configuration if 1.
|                   | Bit 31: PKS. Supports protection keys for supervisor-mode pages if 1.
| EDX               | Bit 00: Reserved.
|                   | Bit 01: SGX-KEYS. If 1, Attestation Services for Intel® SGX is supported.
|                   | Bit 02: AVX512_4VNNIw. (Intel® Xeon Phi™ only.)
|                   | Bit 03: AVX512_4FMAPS. (Intel® Xeon Phi™ only.)
|                   | Bit 04: Fast Short REP MOV.
|                   | Bit 05: UNINTR. If 1, the processor supports user interrupts.
|                   | Bits 07-06: Reserved.
|                   | Bit 08: AVX512_2P2INTERSECT.
|                   | Bit 09: SRBDS_CTRL. If 1, enumerates support for the IA32_MCU_OPT_CTRL MSR and indicates its bit 0 (RNGDS_MITG_DIS) is also supported.
|                   | Bit 10: MD_CLEAR supported.
|                   | Bit 11: RTM_ALWAYS_ABORT. If set, any execution of XBEGIN immediately aborts and transitions to the specified fallback address.
|                   | Bit 12: Reserved.
|                   | Bit 13: If 1, RTM_FORCE_ABORT supported. Processors that set this bit support the IA32_TSC_FORCE_ABORT MSR. They allow software to set IA32_TSC_FORCE_ABORT[0] (RTM_FORCE_ABORT).
|                   | Bit 14: SERIALIZE.
|                   | Bit 15: Hybrid. If 1, the processor is identified as a hybrid part. If CPUID.0.MAXLEAF ≥ 1AH and CPUID.1A.EAX ≠ 0, then the Native Model ID Enumeration Leaf 1AH exists.
|                   | Bit 16: TSXLDTRK. If 1, the processor supports Intel TSX suspend/resume of load address tracking.
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 17: Reserved.</td>
<td>Bit 18: PCONFIG. Supports PCONFIG if 1.</td>
</tr>
<tr>
<td>Bit 19: Architectural LBRs. If 1, indicates support for architectural LBRs.</td>
<td>Bit 22: AMX-BF16. If 1, the processor supports tile computational operations on bfloat16 numbers.</td>
</tr>
<tr>
<td>Bit 20: CET_IBT. Supports CET indirect branch tracking features if 1. Processors that set this bit define bits 5:2 and bits 63:10 of the IA32_U_CET and IA32_S_CET MSRs.</td>
<td>Bit 23: AVX512_FP16.</td>
</tr>
<tr>
<td>Bit 21: Reserved.</td>
<td>Bit 24: AMX-TILE. If 1, the processor supports tile architecture.</td>
</tr>
<tr>
<td>Bit 22: AMX-BF16. If 1, the processor supports tile computational operations on bfloat16 numbers.</td>
<td>Bits 25: AMX-INT8. If 1, the processor supports tile computational operations on 8-bit integers.</td>
</tr>
<tr>
<td>Bit 23: AVX512_FP16.</td>
<td>Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB).</td>
</tr>
<tr>
<td>Bit 24: AMX-TILE. If 1, the processor supports tile architecture.</td>
<td>Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP).</td>
</tr>
<tr>
<td>Bits 25: AMX-INT8. If 1, the processor supports tile computational operations on 8-bit integers.</td>
<td>Bit 28: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH).</td>
</tr>
<tr>
<td>Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB).</td>
<td>Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR.</td>
</tr>
<tr>
<td>Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP).</td>
<td>Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR.</td>
</tr>
<tr>
<td>Bit 28: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH).</td>
<td>IA32_CORE_CAPABILITIES is an architectural MSR that enumerates model-specific features. A bit being set in this MSR indicates that a model specific feature is supported; software must still consult CPUID family/model/stepping to determine the behavior of the enumerated feature as features enumerated in IA32_CORE_CAPABILITIES may have different behavior on different processor models. Some of these features may have behavior that is consistent across processor models (and for which consultation of CPUID family/model/stepping is not necessary); such features are identified explicitly where they are documented in this manual.</td>
</tr>
<tr>
<td>Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR.</td>
<td>Bit 31: Enumerates support for Speculative Store Bypass Disable (SSBD). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[2] (SSBD).</td>
</tr>
</tbody>
</table>

**NOTE:**
* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.

### Structured Extended Feature Enumeration Sub-leaf (Initial EAX Value = 07H, ECX = 1)

<table>
<thead>
<tr>
<th>EAX</th>
<th>NOTES:</th>
</tr>
</thead>
<tbody>
<tr>
<td>07H</td>
<td>Leaf 07H output depends on the initial value in ECX.</td>
</tr>
<tr>
<td></td>
<td>If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.</td>
</tr>
<tr>
<td></td>
<td>This field reports 0 if the sub-leaf index, 1, is invalid.</td>
</tr>
<tr>
<td></td>
<td>Bits 03-00: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 04: AVX-VNNI. AVX (VEX-encoded) versions of the Vector Neural Network Instructions.</td>
</tr>
<tr>
<td></td>
<td>Bit 05: AVX512_BF16. Vector Neural Network Instructions supporting BFLOAT16 inputs and conversion instructions from IEEE single precision.</td>
</tr>
<tr>
<td></td>
<td>Bits 09-06: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 10: If 1, supports fast zero-length REP MOVSB.</td>
</tr>
<tr>
<td></td>
<td>Bit 11: If 1, supports fast short REP STOSB.</td>
</tr>
<tr>
<td></td>
<td>Bit 12: If 1, supports fast short REP CMPSB, REP SCASB.</td>
</tr>
<tr>
<td></td>
<td>Bits 21-13: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 22: HRESET. If 1, supports history reset via the HRESET instruction and the IA32_HRESET_ENABLE MSR. When set, indicates that the Processor History Reset Leaf (EAX = 20H) is valid.</td>
</tr>
<tr>
<td></td>
<td>Bits 29-23: Reserved.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>Bit 30: INVD_DISABLE_POST_BIOS_DONE. If 1, supports INVD execution prevention after BIOS Done. Bit 31: Reserved. This field reports 0 if the sub-leaf index, 1, is invalid. Bit 00: Enumerates the presence of the IA32_PPIN and IA32_PPIN_CTL MSRs. If 1, these MSRs are supported. Bits 31-01: Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>This field reports 0 if the sub-leaf index, 1, is invalid; otherwise it is reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>This field reports 0 if the sub-leaf index, 1, is invalid. Bits 17-00: Reserved. Bit 18: CET_SSS. If 1, indicates that an operating system can enable supervisor shadow stacks as long as it ensures that a supervisor shadow stack cannot become prematurely busy due to page faults (see Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). When emulating the CPUID instruction, a virtual-machine monitor (VMM) should return this bit as 1 only if it ensures that VM exits cannot cause a guest supervisor shadow stack to appear to be prematurely busy. Such a VMM could set the “prematurely busy shadow stack” VM-exit control and use the additional information that it provides. Bits 31-19: Reserved.</td>
</tr>
</tbody>
</table>

#### Structured Extended Feature Enumeration Sub-leaf (Initial EAX Value = 07H, ECX = 2)

<table>
<thead>
<tr>
<th>07H</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>EDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTES:</td>
<td>Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved. This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved. This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved. This field reports 0 if the sub-leaf index, 2, is invalid; otherwise it is reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 00: PSFD. If 1, indicates bit 7 of the IA32_SPEC_CTRL MSR is supported. Bit 7 of this MSR disables Fast Store Forwarding Predictor without disabling Speculative Store Bypass. Bit 01: IPRED_CTRL. If 1, indicates bits 3 and 4 of the IA32_SPEC_CTRL MSR are supported. Bit 3 of this MSR enables IPRED_DIS control for CPL3. Bit 4 of this MSR enables IPRED_DIS control for CPL0/1/2. Bit 02: RRSBA_CTRL. If 1, indicates bits 5 and 6 of the IA32_SPEC_CTRL MSR are supported. Bit 5 of this MSR disables RRSBA behavior for CPL3. Bit 6 of this MSR disables RRSBA behavior for CPL0/1/2. Bit 03: DDPD_U. If 1, indicates bit 8 of the IA32_SPEC_CTRL MSR is supported. Bit 8 of this MSR disables Data Dependent Prefetcher. Bit 04: BHI_CTRL. If 1, indicates bit 10 of the IA32_SPEC_CTRL MSR is supported. Bit 10 of this MSR enables BHI_DIS_S behavior. Bit 05: MCDT_NO. Processors that enumerate this bit as 1 do not exhibit MXCSR Configuration Dependent Timing (MCDT) behavior and do not need to be mitigated to avoid data-dependent behavior for certain instructions. Bits 31-06: Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Direct Cache Access Information Leaf (Initial EAX Value = 09H)

<table>
<thead>
<tr>
<th>09H</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>EDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H). Reserved. Reserved. Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architectural Performance Monitoring Leaf (Initial EAX Value = 0AH)</strong></td>
<td></td>
</tr>
<tr>
<td>0AH EAX Bits 07-00: Version ID of architectural performance monitoring. Bits 15-08: Number of general-purpose performance monitoring counter per logical processor. Bits 23-16: Bit width of general-purpose, performance monitoring counter. Bits 31-24: Length of EBX bit vector to enumerate architectural performance monitoring events. Architectural event ( x ) is supported if ( EBX[x]=0 ) &amp;&amp; ( EAX[31:24]&gt;x ).</td>
<td></td>
</tr>
<tr>
<td>EBX Bit 00: Core cycle event not available if 1 or if ( EAX[31:24]&lt;1 ). Bit 01: Instruction retired event not available if 1 or if ( EAX[31:24]&lt;2 ). Bit 02: Reference cycles event not available if 1 or if ( EAX[31:24]&lt;3 ). Bit 03: Last-level cache reference event not available if 1 or if ( EAX[31:24]&lt;4 ). Bit 04: Last-level cache misses event not available if 1 or if ( EAX[31:24]&lt;5 ). Bit 05: Branch instruction retired event not available if 1 or if ( EAX[31:24]&lt;6 ). Bit 06: Branch mispredict retired event not available if 1 or if ( EAX[31:24]&lt;7 ). Bit 07: Top-down slots event not available if 1 or if ( EAX[31:24]&lt;8 ). Bits 31-08: Reserved = 0.</td>
<td></td>
</tr>
<tr>
<td>ECX Bits 31-00: Supported fixed counters bit mask. Fixed-function performance counter ( i ) is supported if bit ( i ) is 1 (first counter index starts at zero). It is recommended to use the following logic to determine if a Fixed Counter is supported: ( FxCtr[i] _is _supported := ECX[i] | (EDX[4:0] &gt; i) ).</td>
<td></td>
</tr>
<tr>
<td>EDX Bits 04-00: Number of contiguous fixed-function performance counters starting from 0 (if Version ID &gt; 1). Bits 12-05: Bit width of fixed-function performance counters (if Version ID &gt; 1). Bits 14-13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31-16: Reserved = 0.</td>
<td></td>
</tr>
<tr>
<td><strong>Extended Topology Enumeration Leaf (Initial EAX Value = 0BH)</strong></td>
<td></td>
</tr>
<tr>
<td>0BH NOTES: CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH. The sub-leaves of CPUID leaf 0BH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket). The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated. For two valid sub-leaves ( N ) and ( N+1 ), sub-leaf ( N+1 ) represents the next immediate higher-scoped domain with respect to the domain of sub-leaf ( N ) for the given processor. If sub-leaf index &quot;N&quot; returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than &quot;N&quot; shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.</td>
<td></td>
</tr>
<tr>
<td>EAX Bits 04-00: The number of bits that the x2APIC ID must be shifted to the right to address instances of the next higher-scoped domain. When logical processor is not supported by the processor, the value of this field at the Logical Processor domain sub-leaf may be returned as either 0 (no allocated bits in the x2APIC ID) or 1 (one allocated bit in the x2APIC ID); software should plan accordingly. Bits 31-05: Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EBX</strong></td>
<td>Bits 15-00: The number of logical processors across all instances of this domain within the next higher-scoped domain. (For example, in a processor socket/package comprising “M” dies of “N” cores each, where each core has “L” logical processors, the “die” domain sub-leaf value of this field would be M<em>N</em>L.) This number reflects configuration as shipped by Intel. Note, software must not use this field to enumerate processor topology*. Bits 31-16: Reserved.</td>
</tr>
<tr>
<td><strong>ECX</strong></td>
<td>Bits 07-00: The input ECX sub-leaf index. Bits 15-08: Domain Type. This field provides an identification value which indicates the domain as shown below. Although domains are ordered, their assigned identification values are not and software should not depend on it.</td>
</tr>
<tr>
<td><strong>EDX</strong></td>
<td>Bits 31-00: x2APIC ID of the current logical processor.</td>
</tr>
</tbody>
</table>

**NOTES:**
* Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.

Processor Extended State Enumeration Main Leaf (Initial EAX Value = ODH, ECX = 0)

<table>
<thead>
<tr>
<th>ODH</th>
<th>NOTES:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf ODH main leaf (ECX = 0).</td>
<td></td>
</tr>
</tbody>
</table>

| EAX | Bits 31-00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04-03: MPX state. Bits 07-05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 16-10: Used for IA32_XSS. Bit 17: TILECFG state. Bit 18: TILEDATA state. Bits 31-19: Reserved. |

| EBX | Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled. |

| ECX | Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0. |

| EDX | Bit 31-00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31-00: Reserved. |
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor Extended State Enumeration Sub-leaf (Initial EAX Value = ODH, ECX = 1)</strong></td>
<td></td>
</tr>
</tbody>
</table>
| ODH | EAX | Bit 00: XSAVEOPT is available.  
Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set.  
Bit 02: Supports XGETBV with ECX = 1 if set.  
Bit 03: Supports XSAVE/EXRSTORS and IA32_XSS if set.  
Bit 04: Supports extended feature disable (XFD) if set.  
Bits 31-05: Reserved. |
| | EBX | Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCRO | IA32_XSS.  
NOTES:  
If EAX[3] is enumerated as 0 and EAX[1] is enumerated as 1, EBX enumerates the size of the XSAVE area containing all states enabled by XCRO. If EAX[1] and EAX[3] are both enumerated as 0, EBX enumerates zero. |
| | ECX | Bits 31-00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1.  
Bits 07-00: Used for XCRO.  
Bit 08: PT state.  
Bit 09: Used for XCRO.  
Bit 10: PASID state.  
Bit 11: CET user state.  
Bit 12: CET supervisor state.  
Bit 13: HDC state.  
Bit 14: UINTR state.  
Bit 15: LBR state (only for the architectural LBR feature).  
Bit 16: HWP state.  
Bits 18-17: Used for XCRO.  
Bits 31-19: Reserved. |
| | EDX | Bits 31-00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1.  
Bits 31-00: Reserved. |
| **Processor Extended State Enumeration Sub-leaves (Initial EAX Value = ODH, ECX = n, n > 1)** |
| ODH | NOTES:  
Leaf ODH output depends on the initial value in ECX.  
Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCRO register or the IA32_XSS MSR.  
* If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 ≤ n ≤ 31) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (32 ≤ n ≤ 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32]. |
| | EAX | Bits 31-00: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, n. |
| | EBX | Bits 31-00: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area.  
This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCRO register*. |
| | ECX | Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCRO.  
Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component).  
Bits 31-02 are reserved.  
This field reports 0 if the sub-leaf index, n, is invalid*. |
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>This field reports 0 if the sub-leaf index, ( n ), is invalid; otherwise it is reserved.</td>
</tr>
</tbody>
</table>

**Intel® Resource Director Technology (Intel® RDT) Monitoring Enumeration Sub-leaf (Initial EAX Value = 0FH, ECX = 0)**

<table>
<thead>
<tr>
<th>0FH</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leaf 0FH output depends on the initial value in ECX.</td>
</tr>
<tr>
<td></td>
<td>Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.</td>
</tr>
<tr>
<td>EAX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 31-00: Maximum range (zero-based) of RMID within this physical processor of all types.</td>
</tr>
<tr>
<td>ECX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bit 00: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 01: Supports L3 Cache Intel RDT Monitoring if 1.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-02: Reserved.</td>
</tr>
</tbody>
</table>

**L3 Cache Intel® RDT Monitoring Capability Enumeration Sub-leaf (Initial EAX Value = 0FH, ECX = 1)**

<table>
<thead>
<tr>
<th>0FH</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leaf 0FH output depends on the initial value in ECX.</td>
</tr>
<tr>
<td></td>
<td>Bits 07-00: The counter width is encoded as an offset from 24b. A value of zero in this field indicates that 24-bit counters are supported. A value of 8 in this field indicates that 32-bit counters are supported.</td>
</tr>
<tr>
<td></td>
<td>Bit 08: If 1, indicates the presence of an overflow bit in the IA32_QM_CTR MSR (bit 61).</td>
</tr>
<tr>
<td></td>
<td>Bit 09: If 1, indicates the presence of non-CPU agent Intel RDT CMT support.</td>
</tr>
<tr>
<td></td>
<td>Bit 10: If 1, indicates the presence of non-CPU agent Intel RDT MBM support.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-11: Reserved.</td>
</tr>
<tr>
<td>EAX</td>
<td>Bits 31-00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes) and Memory Bandwidth Monitoring (MBM) metrics.</td>
</tr>
<tr>
<td>ECX</td>
<td>Maximum range (zero-based) of RMID of this resource type.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bit 00: Supports L3 occupancy monitoring if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 01: Supports L3 Total Bandwidth monitoring if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 02: Supports L3 Local Bandwidth monitoring if 1.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-03: Reserved.</td>
</tr>
</tbody>
</table>

**Intel® Resource Director Technology (Intel® RDT) Allocation Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = 0)**

<table>
<thead>
<tr>
<th>10H</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leaf 10H output depends on the initial value in ECX.</td>
</tr>
<tr>
<td></td>
<td>Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.</td>
</tr>
<tr>
<td>EAX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bit 00: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bit 01: Supports L3 Cache Allocation Technology if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 02: Supports L2 Cache Allocation Technology if 1.</td>
</tr>
<tr>
<td></td>
<td>Bit 03: Supports Memory Bandwidth Allocation if 1.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-04: Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

**L3 Cache Allocation Technology Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = ResID =1)**

<table>
<thead>
<tr>
<th>10H</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leaf 10H output depends on the initial value in ECX.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Bits 04-00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31-05: Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 31-00: Bit-granular map of isolation/contention of allocation units.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 00: Reserved. Bit 01: If 1, indicates L3 CAT for non-CPU agents is supported. Bit 02: If 1, indicates L3 Code and Data Prioritization Technology is supported. Bit 03: If 1, indicates non-contiguous capacity bitmask is supported. The bits that are set in the various IA32_L3_MASK_n registers do not have to be contiguous. Bits 31-04: Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 15-00: Highest Class of Service (COS) number supported for this ResID. Bits 31-16: Reserved.</td>
</tr>
</tbody>
</table>

**L2 Cache Allocation Technology Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = ResID =2)**

<table>
<thead>
<tr>
<th>10H</th>
<th><strong>NOTES:</strong> Leaf 10H output depends on the initial value in ECX.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Bits 04-00: Length of the capacity bit mask for the corresponding ResID. Add one to the return value to get the result. Bits 31-05: Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 31-00: Bit-granular map of isolation/contention of allocation units.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bit 01: If 1, indicates L2 CAT for non-CPU agents is supported. Bit 02: If 1, indicates L2 Code and Data Prioritization Technology is supported. Bit 03: If 1, indicates non-contiguous capacity bitmask is supported. The bits that are set in the various IA32_L2_MASK_n registers do not have to be contiguous. Bits 31-04: Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved.</td>
</tr>
</tbody>
</table>

**Memory Bandwidth Allocation Enumeration Sub-leaf (Initial EAX Value = 10H, ECX = ResID =3)**

<table>
<thead>
<tr>
<th>10H</th>
<th><strong>NOTES:</strong> Leaf 10H output depends on the initial value in ECX.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Bits 11-00: Reports the maximum MBA throttling value supported for the corresponding ResID. Add one to the return value to get the result. Bits 31-12: Reserved.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 31-00: Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 01-00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31-03: Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved.</td>
</tr>
</tbody>
</table>

**Intel® SGX Capability Enumeration Leaf, Sub-leaf 0 (Initial EAX Value = 12H, ECX = 0)**

| 12H               | **NOTES:** Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID(EAX=07H, ECX=0H); EBX[SGX] = 1. |

---

CPUID—CPU Identification
<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| EAX               | Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions.  
|                   | Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions.  
|                   | Bits 04-02: Reserved.  
|                   | Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVIRTCHILD, EDECVIRTCHILD, and ESETOCONTEXT.  
|                   | Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC.  
|                   | Bit 07: If 1, indicates Intel SGX supports ENCLU instruction leaf EVERIFYREPORT2.  
|                   | Bits 09-08: Reserved.  
|                   | Bit 10: If 1, indicates Intel SGX supports ENCLS instruction leaf EUPDATESVN.  
|                   | Bit 11: If 1, indicates Intel SGX supports ENCLU instruction leaf EDECCSSA.  
|                   | Bits 31-12: Reserved.  
| EBX               | Bits 31-00: MISCESELECT. Bit vector of supported extended SGX features.  
| ECX               | Bits 31-00: Reserved.  
| EDX               | Bits 07-00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is $2^{(EDX[7:0])}$.  
|                   | Bits 15-08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is $2^{(EDX[15:8])}$.  
|                   | Bits 31-16: Reserved.  

**Intel SGX Attributes Enumeration Leaf, Sub-leaf 1 (Initial EAX Value = 12H, ECX = 1)**

12H

**NOTES:**

Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.

- **EAX**
  - Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.

- **EBX**
  - Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.

- **ECX**
  - Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.

- **EDX**
  - Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.

**Intel® SGX EPC Enumeration Leaf, Sub-leaves (Initial EAX Value = 12H, ECX = 2 or higher)**

12H

**NOTES:**

Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.

- **EAX**
  - Bit 03-00: Sub-leaf Type
    - 0000b: This sub-leaf is invalid.
    - 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section.
    - All other type encodings are reserved.

- **Type**
  - 0000b. This sub-leaf is invalid.
  - EDX:ECX:EBX:EAX return 0.
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows.</td>
</tr>
<tr>
<td></td>
<td>EAX[11:04]: Reserved (enumerate 0).</td>
</tr>
<tr>
<td></td>
<td>EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section.</td>
</tr>
<tr>
<td></td>
<td>EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section.</td>
</tr>
<tr>
<td></td>
<td>EBX[31:20]: Reserved.</td>
</tr>
<tr>
<td></td>
<td>ECX[03:00]: EPC section property encoding defined as follows:</td>
</tr>
<tr>
<td></td>
<td>If ECX[3:0] = 0000b, then all bits of the EDX:ECX pair are enumerated as 0.</td>
</tr>
<tr>
<td></td>
<td>If ECX[3:0] = 0001b, then this section has confidentiality and integrity protection.</td>
</tr>
<tr>
<td></td>
<td>If ECX[3:0] = 0010b, then this section has confidentiality protection only.</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
<tr>
<td></td>
<td>ECX[11:04]: Reserved (enumerate 0).</td>
</tr>
<tr>
<td></td>
<td>ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.</td>
</tr>
<tr>
<td></td>
<td>EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory.</td>
</tr>
<tr>
<td></td>
<td>EDX[31:20]: Reserved.</td>
</tr>
</tbody>
</table>

Intel® Processor Trace Enumeration Main Leaf (Initial EAX Value = 14H, ECX = 0)

14H  

NOTES:  
Leaf 14H main leaf (ECX = 0).

EAX  
Bits 31-00: Reports the maximum sub-leaf supported in leaf 14H.

EBX  
Bit 00: If 1, indicates that IA32_RTIT_CTL.CTRL3Filter can be set to 1, and that IA32_RTIT_CTL3_MATCH MSR can be accessed.
Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode.
Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset.
Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets.
Bit 04: If 1, indicates support of PTRYTE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets.
Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation.
Bit 06: If 1, indicates support for PSB and PMI preservation. Writes can set IA32_RTIT_CTL[56] (PendTopaPmi) and/or IA32_RTIT_CTL[6] (PendPsb) in order to preserve ToPA PMIs and/or PSBs otherwise lost due to Intel PT disable. Writes can also set PendTopaPm and PendPsb.
Bit 07: If 1, writes can set IA32_RTIT_CTL[31] (EventEn), enabling Event Trace packet generation.
Bit 08: If 1, writes can set IA32_RTIT_CTL[55] (DisTNT), disabling TNT packet generation.
Bit 31-09: Reserved.

ECX  
Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPa = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed.
Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS.
Bit 02: If 1, indicates support of Single-Range Output scheme.
Bit 03: If 1, indicates support of output to Trace Transport subsystem.
Bit 30-04: Reserved.
Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.

EDX  
Bits 31-00: Reserved.
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>14H</td>
<td><strong>Intel® Processor Trace Enumeration Sub-leaf (Initial EAX Value = 14H, ECX = 1)</strong></td>
</tr>
<tr>
<td>EAX Bits 02-00</td>
<td>Number of configurable Address Ranges for filtering.</td>
</tr>
<tr>
<td></td>
<td>Bits 15-03: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 31-16: Bitmap of supported MTC period encodings.</td>
</tr>
<tr>
<td>EBX Bits 15-00</td>
<td>Bitmap of supported Cycle Threshold value encodings.</td>
</tr>
<tr>
<td></td>
<td>Bit 31-16: Bitmap of supported Configurable PSB frequency encodings.</td>
</tr>
<tr>
<td>ECX Bits 31-00</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EDX Bits 31-00</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

**Time Stamp Counter and Nominal Core Crystal Clock Information Leaf (Initial EAX Value = 15H)**

<table>
<thead>
<tr>
<th>15H</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If EBX[31:0] is 0, the TSC/&quot;core crystal clock&quot; ratio is not enumerated.</td>
</tr>
<tr>
<td></td>
<td>EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency.</td>
</tr>
<tr>
<td></td>
<td>If ECX is 0, the nominal core crystal clock frequency is not enumerated.</td>
</tr>
<tr>
<td></td>
<td>“TSC frequency&quot; = &quot;core crystal clock frequency&quot; * EBX/EAX.</td>
</tr>
<tr>
<td></td>
<td>The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</td>
</tr>
<tr>
<td>EAX Bits 31-00</td>
<td>An unsigned integer which is the denominator of the TSC/&quot;core crystal clock&quot; ratio.</td>
</tr>
<tr>
<td>EBX Bits 31-00</td>
<td>An unsigned integer which is the numerator of the TSC/&quot;core crystal clock&quot; ratio.</td>
</tr>
<tr>
<td>ECX Bits 31-00</td>
<td>An unsigned integer which is the nominal frequency of the core crystal clock in Hz.</td>
</tr>
<tr>
<td>EDX Bits 31-00</td>
<td>Reserved = 0.</td>
</tr>
</tbody>
</table>

**Processor Frequency Information Leaf (Initial EAX Value = 16H)**

<table>
<thead>
<tr>
<th>16H</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX Bits 15-00</td>
<td>Processor Base Frequency (in MHz).</td>
</tr>
<tr>
<td></td>
<td>Bits 31-16: Reserved = 0.</td>
</tr>
<tr>
<td>EBX Bits 15-00</td>
<td>Maximum Frequency (in MHz).</td>
</tr>
<tr>
<td></td>
<td>Bits 31-16: Reserved = 0.</td>
</tr>
<tr>
<td>ECX Bits 15-00</td>
<td>Bus (Reference) Frequency (in MHz).</td>
</tr>
<tr>
<td></td>
<td>Bits 31-16: Reserved = 0.</td>
</tr>
<tr>
<td>EDX Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* Data is returned from this interface in accordance with the processor’s specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.

While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.

**System-On-Chip Vendor Attribute Enumeration Main Leaf (Initial EAX Value = 17H, ECX = 0)**

<table>
<thead>
<tr>
<th>17H</th>
<th><strong>NOTES:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Leaf 17H main leaf (ECX = 0).</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H output depends on the initial value in ECX.</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String.</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H is valid if MaxSOCID_Index &gt;= 3.</td>
</tr>
<tr>
<td></td>
<td>Leaf 17H sub-leaves 4 and above are reserved.</td>
</tr>
</tbody>
</table>
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Bits 31-00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.</td>
</tr>
<tr>
<td>EBX</td>
<td>Bits 15-00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31-17: Reserved = 0.</td>
</tr>
<tr>
<td>ECX</td>
<td>Bits 31-00: Project ID. A unique number an SOC vendor assigns to its SOC projects.</td>
</tr>
<tr>
<td>EDX</td>
<td>Bits 31-00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.</td>
</tr>
</tbody>
</table>

**System-On-Chip Vendor Attribute Enumeration Sub-leaf (Initial EAX Value = 17H, ECX = 1..3)**

<table>
<thead>
<tr>
<th>17H</th>
<th>EBX Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ECX Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
<tr>
<td></td>
<td>EDX Bit 31-00: SOC Vendor Brand String. UTF-8 encoded string.</td>
</tr>
</tbody>
</table>

**NOTES:**
- Leaf 17H output depends on the initial value in ECX.
- SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H.
- The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.

**System-On-Chip Vendor Attribute Enumeration Sub-leaves (Initial EAX Value = 17H, ECX > MaxSOCID_Index)**

<table>
<thead>
<tr>
<th>17H</th>
<th>NOTES: Leaf 17H output depends on the initial value in ECX.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EAX Bits 31-00: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>EBX Bits 31-00: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>ECX Bits 31-00: Reserved = 0.</td>
</tr>
<tr>
<td></td>
<td>EDX Bits 31-00: Reserved = 0.</td>
</tr>
</tbody>
</table>

**Deterministic Address Translation Parameters Main Leaf (Initial EAX Value = 18H, ECX = 0)**

<table>
<thead>
<tr>
<th>18H</th>
<th>NOTES: Each sub-leaf enumerates a different address translation structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product. ** Add one to the return value to get the result.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EAX Bits 31-00: Reports the maximum input value of supported sub-leaf in leaf 18H.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| EBX               | Bit 00: 4K page size entries supported by this structure.  
|                   | Bit 01: 2MB page size entries supported by this structure.  
|                   | Bit 02: 4MB page size entries supported by this structure.  
|                   | Bit 03: 1 GB page size entries supported by this structure.  
|                   | Bits 07-04: Reserved.  
|                   | Bits 10-08: Partitioning (0: Soft partitioning between the logical processors sharing this structure).  
|                   | Bits 15-11: Reserved.  
|                   | Bits 31-16: W = Ways of associativity.  
| ECX               | Bits 31-00: S = Number of Sets.  
| EDX               | Bits 04-00: Translation cache type field.  
|                   | 00000b: Null (indicates this sub-leaf is not valid).  
|                   | 00001b: Data TLB.  
|                   | 00010b: Instruction TLB.  
|                   | 00011b: Unified TLB*.  
|                   | 00100b: Load Only TLB. Hit on loads; fills on both loads and stores.  
|                   | 00101b: Store Only TLB. Hit on stores; fill on stores.  
|                   | All other encodings are reserved.  
|                   | Bits 07-05: Translation cache level (starts at 1).  
|                   | Bit 08: Fully associative structure.  
|                   | Bits 13-09: Reserved.  
|                   | Bits 25-14: Maximum number of addressable IDs for logical processors sharing this translation cache.**  
|                   | Bits 31-26: Reserved.  

#### Deterministic Address Translation Parameters Sub-leaf (Initial EAX Value = 18H, ECX ≥ 1)

- **18H**
  - **NOTES:**
    - Each sub-leaf enumerates a different address translation structure.
    - If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure.
    - * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch. See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product.
    - ** Add one to the return value to get the result.
  - EAX | Bits 31-00: Reserved.
  - EBX | Bit 00: 4K page size entries supported by this structure.  
  |       | Bit 01: 2MB page size entries supported by this structure.  
  |       | Bit 02: 4MB page size entries supported by this structure.  
  |       | Bit 03: 1 GB page size entries supported by this structure.  
  |       | Bits 07-04: Reserved.  
  |       | Bits 10-08: Partitioning (0: Soft partitioning between the logical processors sharing this structure).  
  |       | Bits 15-11: Reserved.  
  |       | Bits 31-16: W = Ways of associativity.  
  - ECX | Bits 31-00: S = Number of Sets.
Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>Bits 04-00: Translation cache type field.</td>
</tr>
<tr>
<td></td>
<td>0000b: Null (indicates this sub-leaf is not valid).</td>
</tr>
<tr>
<td></td>
<td>0001b: Data TLB.</td>
</tr>
<tr>
<td></td>
<td>0010b: Instruction TLB.</td>
</tr>
<tr>
<td></td>
<td>0011b: Unified TLB*.</td>
</tr>
<tr>
<td></td>
<td>All other encodings are reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 07-05: Translation cache level (starts at 1).</td>
</tr>
<tr>
<td></td>
<td>Bit 08: Fully associative structure.</td>
</tr>
<tr>
<td></td>
<td>Bits 13-09: Reserved.</td>
</tr>
<tr>
<td></td>
<td>Bits 25-14: Maximum number of addressable IDs for logical processors sharing this translation cache**</td>
</tr>
<tr>
<td></td>
<td>Bits 31-26: Reserved.</td>
</tr>
</tbody>
</table>

| EAX               | Bit 00: Key Locker restriction of CPL0-only supported. |
|                   | Bit 01: Key Locker restriction of no-encrypt supported. |
|                   | Bit 02: Key Locker restriction of no-decrypt supported. |
|                   | Bits 31-03: Reserved. |

| EBX               | Bit 00: AESKLE. If 1, the AES Key Locker instructions are fully enabled. |
|                   | Bit 01: Reserved. |
|                   | Bit 02: If 1, the AES wide Key Locker instructions are supported. |
|                   | Bit 03: Reserved. |
|                   | Bit 04: If 1, the platform supports the Key Locker MSRs (IA32_COPY_LOCAL_TO_PLATFORM, IA23_COPY_PLATFORM_TO_LOCAL, IA32_COPY_STATUS, and IA32_IWKEYBACKUP_STATUS) and backing up the internal wrapping key. |
|                   | Bits 31-05: Reserved. |

| ECX               | Bit 00: If 1, the NoBackup parameter to LOADIWKEY is supported. |
|                   | Bit 01: If 1, KeySource encoding of 1 (randomization of the internal wrapping key) is supported. |
|                   | Bits 31-02: Reserved. |
| EDX               | Reserved. |

**Native Model ID Enumeration Leaf (Initial EAX Value = 1AH, ECX = 0)**

| 1AH               | NOTES: |
|                   | This leaf exists on all hybrid parts, however this leaf is not only available on hybrid parts. The following algorithm is used for detection of this leaf: |
|                   | If CPUID.0.MAXLEAF ≥ 1AH and CPUID.1A.EAX ≠ 0, then the leaf exists. |
| EAX               | Enumerates the native model ID and core type. |
|                   | Bits 31-24: Core type* |
|                   | 10H: Reserved |
|                   | 20H: Intel Atom* |
|                   | 30H: Reserved |
|                   | 40H: Intel® Core™ |
|                   | Bits 23-00: Native model ID of the core. The core-type and native model ID can be used to uniquely identify the microarchitecture of the core. This native model ID is not unique across core types, and not related to the model ID reported in CPUID leaf 01H, and does not identify the SOC. |
|                   | * The core type may only be used as an identification of the microarchitecture for this logical processor and its numeric value has no significance, neither large nor small. This field neither implies nor expresses any other attribute to this logical processor and software should not assume any. |

| EBX               | Reserved. |
| ECX               | Reserved. |
| EDX               | Reserved. |
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PCONFIG Information Sub-leaf (Initial EAX Value = 1BH, ECX ≥ 0)</strong></td>
<td></td>
</tr>
</tbody>
</table>
| 1BH | For details on this sub-leaf, see "INPUT EAX = 1BH: Returns PCONFIG Information" on page 3-253.  
**NOTE:** Leaf 1BH is supported if CPUID.(EAX=07H, ECX=0H):EDX[18] = 1. |

<table>
<thead>
<tr>
<th><strong>Last Branch Records Information Leaf (Initial EAX Value = 1CH)</strong></th>
<th></th>
</tr>
</thead>
</table>
| 1CH | **NOTE:** This leaf pertains to the architectural feature.  
EAX | Bits 07-00: Supported LBR Depth Values. For each bit n set in this field, the IA32_LBR_DEPTH.DEPTH value 8*(n+1) is supported.  
Bit 30: Deep C-state Reset. If set, indicates that LBRs may be cleared on an MWAIT that requests a C-state numerically greater than C1.  
Bit 31: IP Values Contain LIP. If set, LBR IP values contain LIP. If clear, IP values contain Effective IP.  
EBX | Bit 00: CPL Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[2:1] to non-zero value.  
Bit 01: Branch Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[22:16] to non-zero value.  
Bit 02: Call-stack Mode Supported. If set, the processor supports setting IA32_LBR_CTL[3] to 1.  
Bits 31-03: Reserved.  
ECX | Bit 00: Mispredict Bit Supported. IA32_LBR_x_INFO[63] holds indication of branch misprediction (MISPRED).  
Bit 01: Timed LBRs Supported. IA32_LBR_x_INFO[15:0] holds CPU cycles since last LBR entry (CYC_CNT), and IA32_LBR_x_INFO[60] holds an indication of whether the value held there is valid (CYC_CNT_VALID).  
Bit 02: Branch Type Field Supported. IA32_LBR_INFO_x[59:56] holds indication of the recorded operation's branch type (BR_TYPE).  
Bits 31-03: Reserved.  
EDX | Bits 31-00: Reserved. |

<table>
<thead>
<tr>
<th><strong>Tile Information Main Leaf (Initial EAX Value = 1DH, ECX = 0)</strong></th>
<th></th>
</tr>
</thead>
</table>
| 1DH | **NOTES:** For sub-leaves of 1DH, they are indexed by the palette id.  
Leaf 1DH sub-leaves 2 and above are reserved.  
EAX | Bits 31-00: max_palette. Highest numbered palette sub-leaf. Value = 1.  
EBX | Bits 31-00: Reserved = 0.  
ECX | Bits 31-00: Reserved = 0.  
EDX | Bits 31-00: Reserved = 0. |

<table>
<thead>
<tr>
<th><strong>Tile Palette 1 Sub-leaf (Initial EAX Value = 1DH, ECX = 1)</strong></th>
<th></th>
</tr>
</thead>
</table>
| 1DH | EAX | Bits 15-00: Palette 1 total_tile_bytes. Value = 8192.  
Bits 31-16: Palette 1 bytes_per_tile. Value = 1024.  
EBX | Bits 15-00: Palette 1 bytes_per_row. Value = 64.  
Bits 31-16: Palette 1 max_names (number of tile registers). Value = 8.  
ECX | Bits 15-00: Palette 1 max_rows. Value = 16.  
Bits 31-16: Reserved = 0.  
EDX | Bits 31-00: Reserved = 0. |
**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TMUL Information Main Leaf (Initial EAX Value = 1EH, ECX = 0)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1EH</strong></td>
<td>NOTE:</td>
</tr>
<tr>
<td></td>
<td>Leaf 1EH sub-leaves 1 and above are reserved.</td>
</tr>
<tr>
<td></td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECX</td>
</tr>
<tr>
<td></td>
<td>EDX</td>
</tr>
<tr>
<td><strong>V2 Extended Topology Enumeration Leaf (Initial EAX Value = 1FH)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1FH</strong></td>
<td>NOTES:</td>
</tr>
<tr>
<td></td>
<td>CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends using leaf 1FH when available rather than leaf 0BH and ensuring that any leaf 0BH algorithms are updated to support leaf 1FH.</td>
</tr>
<tr>
<td></td>
<td>The sub-leaves of CPUID leaf 1FH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket).</td>
</tr>
<tr>
<td></td>
<td>The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated. As an example, a processor may report an ordered hierarchy consisting only of “Logical Processor,” “Core,” and “Die.” For two valid sub-leaves N and N+1, sub-leaf N+1 represents the next immediate higher-scoped domain with respect to the domain of sub-leaf N for the given processor.</td>
</tr>
<tr>
<td></td>
<td>If sub-leaf index “N” returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than “N” shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.</td>
</tr>
<tr>
<td></td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
</table>
| ECX               | Bits 07-00: The input ECX sub-leaf index.  
|                   | Bits 15-08: Domain Type. This field provides an identification value which indicates the domain as shown below. Although domains are ordered, as also shown below, their assigned identification values are not and software should not depend on it. (For example, if a new domain between core and module is specified, it will have an identification value higher than 5.)  
|                   | Hierarchy | Domain | Domain Type Identification Value |
|                   | Lowest | Logical Processor | 1 |
|                   | ...    | Core | 2 |
|                   | ...    | Module | 3 |
|                   | ...    | Tile | 4 |
|                   | ...    | Die | 5 |
|                   | ...    | DieGrp | 6 |
|                   | Highest | Package/Socket | (implied) |
|                   | (Note that enumeration values of 0 and 7-255 are reserved.) |
| EDX               | Bits 31-00: x2APIC ID of the current logical processor. It is always valid and does not vary with the sub-leaf index in ECX. |

**NOTES:**
- Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations.

### Processor History Reset Sub-leaf (Initial EAX Value = 20H, ECX = 0)

<table>
<thead>
<tr>
<th>EAX</th>
<th>Reports the maximum number of sub-leaves that are supported in leaf 20H.</th>
</tr>
</thead>
</table>
| EBX | Indicates which bits may be set in the IA32_HRESET_ENABLE MSR to enable reset of different components of hardware-maintained history.  
|     | Bit 00: Indicates support for both HRESET's EAX[0] parameter, and IA32_HRESET_ENABLE[0] set by the OS to enable reset of Intel® Thread Director history.  
|     | Bits 31-01: Reserved = 0. |
| ECX | Reserved. |
| EDX | Reserved. |

**Unimplemented CPUID Leaf Functions**

| 21H | Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is 21H. If the value returned by CPUID.0:EAX (the maximum input value for basic CPUID information) is at least 21H, 0 is returned in the registers EAX, EBX, ECX, and EDX. Otherwise, the data for the highest basic information leaf is returned. |
| 400000000H — 4FFFFFFFH | Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH. |

**Extended Function CPUID Information**

<table>
<thead>
<tr>
<th>80000000H</th>
<th>Maximum Input Value for Extended Function CPUID Information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>Reserved.</td>
</tr>
<tr>
<td>EDX</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
### Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000001H</td>
<td><strong>EAX</strong> Extended Processor Signature and Feature Bits.</td>
</tr>
<tr>
<td></td>
<td><strong>EBX</strong> Reserved.</td>
</tr>
</tbody>
</table>
|                    | **ECX** Bit 00: LAHF/SAHF available in 64-bit mode.*
|                    | Bits 04-01: Reserved.
|                    | Bit 05: LZCNT.
|                    | Bits 07-06: Reserved.
|                    | Bit 08: PREFETCHW.
|                    | Bits 31-09: Reserved.  |
|                    | **EDX** Bits 10-00: Reserved.
|                    | Bit 11: SYSCALL/SYSRET.**
|                    | Bits 19-12: Reserved = 0.
|                    | Bit 20: Execute Disable Bit available.
|                    | Bits 25-21: Reserved = 0.
|                    | Bit 26: 1-GByte pages are available if 1.
|                    | Bit 27: RDTSCP and IA32_TSC_AUX are available if 1.
|                    | Bit 28: Reserved = 0.
|                    | Bit 29: Intel® 64 Architecture available if 1.
|                    | Bits 31-30: Reserved = 0.  |
|                    | **NOTES:**  |
|                    | * LAHF and SAHF are always available in other modes, regardless of the enumeration of this feature flag.  |
|                    | ** Intel processors support SYSCALL and SYSRET only in 64-bit mode. This feature flag is always enumerated as 0 outside 64-bit mode.  |
| 80000002H          | **EAX** Processor Brand String.  |
|                    | **EBX** Processor Brand String Continued.  |
|                    | **ECX** Processor Brand String Continued.  |
|                    | **EDX** Processor Brand String Continued.  |
| 80000003H          | **EAX** Processor Brand String Continued.  |
|                    | **EBX** Processor Brand String Continued.  |
|                    | **ECX** Processor Brand String Continued.  |
|                    | **EDX** Processor Brand String Continued.  |
| 80000004H          | **EAX** Processor Brand String Continued.  |
|                    | **EBX** Processor Brand String Continued.  |
|                    | **ECX** Processor Brand String Continued.  |
|                    | **EDX** Processor Brand String Continued.  |
| 80000005H          | **EAX** Reserved = 0.  |
|                    | **EBX** Reserved = 0.  |
|                    | **ECX** Reserved = 0.  |
|                    | **EDX** Reserved = 0.  |
| 80000006H          | **EAX** Reserved = 0.  |
|                    | **EBX** Reserved = 0.  |
|                    | **ECX** Bits 07-00: Cache Line size in bytes.
|                    | Bits 11-08: Reserved.
|                    | Bits 15-12: L2 Associativity field *.
|                    | Bits 31-16: Cache size in 1K units.  |
|                    | **EDX** Reserved = 0.  |
## Table 3-8. Information Returned by CPUID Instruction (Contd.)

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTES:</strong></td>
<td></td>
</tr>
<tr>
<td>* L2 associativity field encodings:</td>
<td></td>
</tr>
<tr>
<td>00H - Disabled</td>
<td>08H - 16 ways</td>
</tr>
<tr>
<td>01H - 1 way (direct mapped)</td>
<td>09H - Reserved</td>
</tr>
<tr>
<td>02H - 2 ways</td>
<td>0AH - 32 ways</td>
</tr>
<tr>
<td>03H - Reserved</td>
<td>08H - 48 ways</td>
</tr>
<tr>
<td>04H - 4 ways</td>
<td>0CH - 64 ways</td>
</tr>
<tr>
<td>05H - Reserved</td>
<td>0DH - 96 ways</td>
</tr>
<tr>
<td>06H - 8 ways</td>
<td>0EH - 128 ways</td>
</tr>
<tr>
<td>07H - See CPUID leaf 04H, sub-leaf 2**</td>
<td>0FH - Fully associative</td>
</tr>
</tbody>
</table>

**CPUID leaf 04H provides details of deterministic cache parameters, including the L2 cache in sub-leaf 2.**

### INPUT EAX = 0: Returns CPUID’s Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is “GenuineIntel” and is expressed:

- **EBX** := 756e6547h (*“Genu”, with G in the low eight bits of BL*)
- **EDX** := 49656e69h (*“intel”, with i in the low eight bits of DL*)
- **ECX** := 6c65746eh (*“ntel”, with n in the low eight bits of CL*)

### INPUT EAX = 80000000H: Returns CPUID’s Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

### IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.

![Figure 3-6. Version Information Returned by CPUID in EAX](image)

Table 3-9. Processor Type Field

<table>
<thead>
<tr>
<th>Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original OEM Processor</td>
<td>00B</td>
</tr>
<tr>
<td>Intel OverDrive Processor</td>
<td>01B</td>
</tr>
<tr>
<td>Dual processor (not applicable to Intel486 processors)</td>
<td>10B</td>
</tr>
<tr>
<td>Intel reserved</td>
<td>11B</td>
</tr>
</tbody>
</table>

NOTE

See Chapter 20 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN DisplayFamily = Family_ID;
    ELSE DisplayFamily = Extended_Family_ID + Family_ID;
FI;
(* Show DisplayFamily as HEX field. *)
```
The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN DisplayModel = (Extended_Model_ID << 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field.*)
```

**INPUT EAX = 01H: Returns Additional Information in EBX**

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- **Brand index (low byte of EBX)** — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- **CLFLUSH instruction cache line size (second byte of EBX)** — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- **Local APIC ID (high byte of EBX)** — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

**INPUT EAX = 01H: Returns Feature Information in ECX and EDX**

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

**NOTE**

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.
### Table 3-10. Feature Information Returned in the ECX Register

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SSE3</td>
<td>Streaming SIMD Extensions 3 (SSE3). A value of 1 indicates the processor supports this technology.</td>
</tr>
<tr>
<td>1</td>
<td>PCLMULQDQ</td>
<td>PCLMULQDQ. A value of 1 indicates the processor supports the PCLMULQDQ instruction.</td>
</tr>
<tr>
<td>2</td>
<td>DTES64</td>
<td>64-bit DS Area. A value of 1 indicates the processor supports DS area using 64-bit layout.</td>
</tr>
<tr>
<td>3</td>
<td>MONITOR</td>
<td>MONITOR/MWAIT. A value of 1 indicates the processor supports this feature.</td>
</tr>
<tr>
<td>4</td>
<td>DS-CPL</td>
<td>CPL Qualified Debug Store. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.</td>
</tr>
<tr>
<td>5</td>
<td>VMX</td>
<td>Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.</td>
</tr>
<tr>
<td>6</td>
<td>SMX</td>
<td>Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 7, “Safer Mode Extensions Reference.”</td>
</tr>
<tr>
<td>7</td>
<td>EIST</td>
<td>Enhanced Intel SpeedStep® technology. A value of 1 indicates that the processor supports this technology.</td>
</tr>
<tr>
<td>8</td>
<td>TM2</td>
<td>Thermal Monitor 2. A value of 1 indicates whether the processor supports this technology.</td>
</tr>
<tr>
<td>9</td>
<td>SSSE3</td>
<td>A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.</td>
</tr>
</tbody>
</table>

- **Figure 3-7.** Feature Information Returned in the ECX Register
<table>
<thead>
<tr>
<th>Bit #</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CNXT-ID</td>
<td><strong>L1 Context ID.</strong> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.</td>
</tr>
<tr>
<td>11</td>
<td>SDBG</td>
<td>A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.</td>
</tr>
<tr>
<td>12</td>
<td>FMA</td>
<td>A value of 1 indicates the processor supports FMA extensions using YMM state.</td>
</tr>
<tr>
<td>13</td>
<td>CMPXCHG16B</td>
<td><strong>CMPXCHG16B Available.</strong> A value of 1 indicates that the feature is available. See the &quot;CMPXCHG16B—Compare and Exchange Bytes&quot; section in this chapter for a description.</td>
</tr>
<tr>
<td>14</td>
<td>xTPR Update</td>
<td><strong>xTPR Update Control.</strong> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>PDCM</td>
<td><strong>Perfmon and Debug Capability:</strong> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>17</td>
<td>PCID</td>
<td><strong>Process-context identifiers.</strong> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.</td>
</tr>
<tr>
<td>18</td>
<td>DCA</td>
<td>A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.</td>
</tr>
<tr>
<td>19</td>
<td>SSE4_1</td>
<td>A value of 1 indicates that the processor supports SSE4.1.</td>
</tr>
<tr>
<td>20</td>
<td>SSE4_2</td>
<td>A value of 1 indicates that the processor supports SSE4.2.</td>
</tr>
<tr>
<td>21</td>
<td>x2APIC</td>
<td>A value of 1 indicates that the processor supports x2APIC feature.</td>
</tr>
<tr>
<td>22</td>
<td>MOVBE</td>
<td>A value of 1 indicates that the processor supports MOVBE instruction.</td>
</tr>
<tr>
<td>23</td>
<td>POPCNT</td>
<td>A value of 1 indicates that the processor supports the POPCNT instruction.</td>
</tr>
<tr>
<td>24</td>
<td>TSC-Deadline</td>
<td>A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.</td>
</tr>
<tr>
<td>25</td>
<td>AESNI</td>
<td>A value of 1 indicates that the processor supports the AESNI instruction extensions.</td>
</tr>
<tr>
<td>26</td>
<td>XSAVE</td>
<td>A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.</td>
</tr>
<tr>
<td>27</td>
<td>OSXSAVE</td>
<td>A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.</td>
</tr>
<tr>
<td>28</td>
<td>AVX</td>
<td>A value of 1 indicates the processor supports the AVX instruction extensions.</td>
</tr>
<tr>
<td>29</td>
<td>F16C</td>
<td>A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.</td>
</tr>
<tr>
<td>30</td>
<td>RDRAND</td>
<td>A value of 1 indicates that processor supports RDRAND instruction.</td>
</tr>
<tr>
<td>31</td>
<td>Not Used</td>
<td>Always returns 0.</td>
</tr>
</tbody>
</table>
Figure 3-8. Feature Information Returned in the EDX Register
<table>
<thead>
<tr>
<th>Bit #</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FPU</td>
<td>Floating-Point Unit On-Chip. The processor contains an x87 FPU.</td>
</tr>
<tr>
<td>1</td>
<td>VME</td>
<td>Virtual 8086 Mode Enhancements. Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirect expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.</td>
</tr>
<tr>
<td>2</td>
<td>DE</td>
<td>Debugging Extensions. Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.</td>
</tr>
<tr>
<td>3</td>
<td>PSE</td>
<td>Page Size Extension. Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.</td>
</tr>
<tr>
<td>4</td>
<td>TSC</td>
<td>Time Stamp Counter. The RDTSC instruction is supported, including CR4.TSD for controlling privilege.</td>
</tr>
<tr>
<td>5</td>
<td>MSR</td>
<td>Model Specific Registers RDMSR and WRMSR Instructions. The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.</td>
</tr>
<tr>
<td>6</td>
<td>PAE</td>
<td>Physical Address Extension. Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 MByte pages if PAE bit is 1.</td>
</tr>
<tr>
<td>7</td>
<td>MCE</td>
<td>Machine Check Exception. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.</td>
</tr>
<tr>
<td>8</td>
<td>CX8</td>
<td>CMPXCHG8B Instruction. The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).</td>
</tr>
<tr>
<td>9</td>
<td>APIC</td>
<td>APIC On-Chip. The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).</td>
</tr>
<tr>
<td>10</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>SEP</td>
<td>SYSENTER and SYSEXIT Instructions. The SYSENTER and SYSEXIT and associated MSRs are supported.</td>
</tr>
<tr>
<td>12</td>
<td>MTRR</td>
<td>Memory Type Range Registers. MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.</td>
</tr>
<tr>
<td>13</td>
<td>PGE</td>
<td>Page Global Bit. The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.</td>
</tr>
<tr>
<td>14</td>
<td>MCA</td>
<td>Machine Check Architecture. A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.</td>
</tr>
<tr>
<td>15</td>
<td>CMOV</td>
<td>Conditional Move Instructions. The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCMI and FCMOV instructions are supported.</td>
</tr>
<tr>
<td>16</td>
<td>PAT</td>
<td>Page Attribute Table. Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.</td>
</tr>
<tr>
<td>17</td>
<td>PSE-36</td>
<td>36-Bit Page Size Extension. 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHVADDR and may be up to 40 bits in size.</td>
</tr>
<tr>
<td>18</td>
<td>PSN</td>
<td>Processor Serial Number. The processor supports the 96-bit processor identification number feature and the feature is enabled.</td>
</tr>
<tr>
<td>19</td>
<td>CLFSH</td>
<td>CLFLUSH Instruction. CLFLUSH Instruction is supported.</td>
</tr>
<tr>
<td>20</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
When CPUID executes with EAX set to 02H, the processor returns information about the processor’s internal TLBs, cache, and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of “cache type” via CPUID leaf 2.
### Table 3-12. Encoding of CPUID Leaf 2 Descriptors

<table>
<thead>
<tr>
<th>Descriptor Value</th>
<th>Type</th>
<th>Cache or TLB Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>General</td>
<td>Null descriptor, this byte contains no information.</td>
</tr>
<tr>
<td>01H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries.</td>
</tr>
<tr>
<td>02H</td>
<td>TLB</td>
<td>Instruction TLB: 4 MByte pages, fully associative, 2 entries.</td>
</tr>
<tr>
<td>03H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way set associative, 64 entries.</td>
</tr>
<tr>
<td>04H</td>
<td>TLB</td>
<td>Data TLB: 4 MByte pages, 4-way set associative, 8 entries.</td>
</tr>
<tr>
<td>05H</td>
<td>TLB</td>
<td>Data TLB1: 4 MByte pages, 4-way set associative, 32 entries.</td>
</tr>
<tr>
<td>06H</td>
<td>Cache</td>
<td>1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>08H</td>
<td>Cache</td>
<td>1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>09H</td>
<td>Cache</td>
<td>1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>0AH</td>
<td>Cache</td>
<td>1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>0BH</td>
<td>TLB</td>
<td>Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries.</td>
</tr>
<tr>
<td>0CH</td>
<td>Cache</td>
<td>1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>0DH</td>
<td>Cache</td>
<td>1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>0EH</td>
<td>Cache</td>
<td>1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>1DH</td>
<td>Cache</td>
<td>2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>21H</td>
<td>Cache</td>
<td>2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>22H</td>
<td>Cache</td>
<td>3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>23H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>24H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>25H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>29H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>2CH</td>
<td>Cache</td>
<td>1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>30H</td>
<td>Cache</td>
<td>1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>40H</td>
<td>Cache</td>
<td>No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache.</td>
</tr>
<tr>
<td>41H</td>
<td>Cache</td>
<td>2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>42H</td>
<td>Cache</td>
<td>2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>43H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>44H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>45H</td>
<td>Cache</td>
<td>2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>46H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>47H</td>
<td>Cache</td>
<td>3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>48H</td>
<td>Cache</td>
<td>2nd-level cache: 3MByte, 12-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>49H</td>
<td>Cache</td>
<td>3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>4AH</td>
<td>Cache</td>
<td>3rd-level cache: 6MByte, 12-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>4BH</td>
<td>Cache</td>
<td>3rd-level cache: 8MByte, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>4CH</td>
<td>Cache</td>
<td>3rd-level cache: 12MByte, 12-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>4DH</td>
<td>Cache</td>
<td>3rd-level cache: 16MByte, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>4EH</td>
<td>Cache</td>
<td>2nd-level cache: 6MByte, 24-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>4FH</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, 32 entries.</td>
</tr>
<tr>
<td>Descriptor Value</td>
<td>Type</td>
<td>Cache or TLB Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>50H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries.</td>
</tr>
<tr>
<td>51H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries.</td>
</tr>
<tr>
<td>52H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries.</td>
</tr>
<tr>
<td>55H</td>
<td>TLB</td>
<td>Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries.</td>
</tr>
<tr>
<td>56H</td>
<td>TLB</td>
<td>Data TLB0: 4 MByte pages, 4-way set associative, 16 entries.</td>
</tr>
<tr>
<td>57H</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte pages, 4-way associative, 16 entries.</td>
</tr>
<tr>
<td>59H</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte pages, fully associative, 16 entries.</td>
</tr>
<tr>
<td>5AH</td>
<td>TLB</td>
<td>Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries.</td>
</tr>
<tr>
<td>58H</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte and 4 MByte pages, 64 entries.</td>
</tr>
<tr>
<td>5CH</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte and 4 MByte pages, 128 entries.</td>
</tr>
<tr>
<td>5DH</td>
<td>TLB</td>
<td>Data TLB0: 4 KByte and 4 MByte pages, 256 entries.</td>
</tr>
<tr>
<td>60H</td>
<td>Cache</td>
<td>1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>61H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, fully associative, 48 entries.</td>
</tr>
<tr>
<td>63H</td>
<td>TLB</td>
<td>Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries.</td>
</tr>
<tr>
<td>64H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way set associative, 512 entries.</td>
</tr>
<tr>
<td>66H</td>
<td>Cache</td>
<td>1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>67H</td>
<td>Cache</td>
<td>1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>68H</td>
<td>Cache</td>
<td>1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>6AH</td>
<td>Cache</td>
<td>uTLB: 4 KByte pages, 8-way set associative, 64 entries.</td>
</tr>
<tr>
<td>6BH</td>
<td>Cache</td>
<td>DTLB: 4 KByte pages, 8-way set associative, 256 entries.</td>
</tr>
<tr>
<td>6CH</td>
<td>Cache</td>
<td>DTLB: 2M/4M pages, 8-way set associative, 128 entries.</td>
</tr>
<tr>
<td>6DH</td>
<td>Cache</td>
<td>DTLB: 1 GByte pages, fully associative, 16 entries.</td>
</tr>
<tr>
<td>70H</td>
<td>Cache</td>
<td>Trace cache: 12 K-μop, 8-way set associative.</td>
</tr>
<tr>
<td>71H</td>
<td>Cache</td>
<td>Trace cache: 16 K-μop, 8-way set associative.</td>
</tr>
<tr>
<td>72H</td>
<td>Cache</td>
<td>Trace cache: 32 K-μop, 8-way set associative.</td>
</tr>
<tr>
<td>76H</td>
<td>TLB</td>
<td>Instruction TLB: 2M/4M pages, fully associative, 8 entries.</td>
</tr>
<tr>
<td>78H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 4-way set associative, 64byte line size.</td>
</tr>
<tr>
<td>79H</td>
<td>Cache</td>
<td>2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>7AH</td>
<td>Cache</td>
<td>2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>7BH</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>7CH</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector.</td>
</tr>
<tr>
<td>7DH</td>
<td>Cache</td>
<td>2nd-level cache: 2 MByte, 8-way set associative, 64byte line size.</td>
</tr>
<tr>
<td>7FH</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size.</td>
</tr>
<tr>
<td>80H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size.</td>
</tr>
<tr>
<td>82H</td>
<td>Cache</td>
<td>2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>83H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>84H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>85H</td>
<td>Cache</td>
<td>2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size.</td>
</tr>
<tr>
<td>86H</td>
<td>Cache</td>
<td>2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>87H</td>
<td>Cache</td>
<td>2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>Descriptor Value</td>
<td>Type</td>
<td>Cache or TLB Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>A0H</td>
<td>DTLB</td>
<td>DTLB: 4k pages, fully associative, 32 entries.</td>
</tr>
<tr>
<td>B0H</td>
<td>TLB</td>
<td>Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries.</td>
</tr>
<tr>
<td>B1H</td>
<td>TLB</td>
<td>Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries.</td>
</tr>
<tr>
<td>B2H</td>
<td>TLB</td>
<td>Instruction TLB: 4KByte pages, 4-way set associative, 64 entries.</td>
</tr>
<tr>
<td>B3H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way set associative, 128 entries.</td>
</tr>
<tr>
<td>B4H</td>
<td>TLB</td>
<td>Data TLB1: 4 KByte pages, 4-way associative, 256 entries.</td>
</tr>
<tr>
<td>B5H</td>
<td>TLB</td>
<td>Instruction TLB: 4KByte pages, 8-way set associative, 64 entries.</td>
</tr>
<tr>
<td>B6H</td>
<td>TLB</td>
<td>Instruction TLB: 4KByte pages, 8-way set associative, 128 entries.</td>
</tr>
<tr>
<td>BAH</td>
<td>TLB</td>
<td>Data TLB: 4 KByte pages, 4-way associative, 64 entries.</td>
</tr>
<tr>
<td>C0H</td>
<td>TLB</td>
<td>Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries.</td>
</tr>
<tr>
<td>C1H</td>
<td>STLB</td>
<td>Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries.</td>
</tr>
<tr>
<td>C2H</td>
<td>DTLB</td>
<td>DTLB: 4KByte/2 MByte pages, 4-way associative, 16 entries.</td>
</tr>
<tr>
<td>C3H</td>
<td>STLB</td>
<td>Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GByte pages, 4-way, 16 entries.</td>
</tr>
<tr>
<td>C4H</td>
<td>DTLB</td>
<td>DTLB: 2M/4M Byte pages, 4-way associative, 32 entries.</td>
</tr>
<tr>
<td>CAH</td>
<td>STLB</td>
<td>Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries.</td>
</tr>
<tr>
<td>D0H</td>
<td>Cache</td>
<td>3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>D1H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>D2H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>D6H</td>
<td>Cache</td>
<td>3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>D7H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>D8H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>DCH</td>
<td>Cache</td>
<td>3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>DDH</td>
<td>Cache</td>
<td>3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>D6H</td>
<td>Cache</td>
<td>3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>E2H</td>
<td>Cache</td>
<td>3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>E3H</td>
<td>Cache</td>
<td>3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>E4H</td>
<td>Cache</td>
<td>3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>EAH</td>
<td>Cache</td>
<td>3rd-level cache: 12MByte, 24-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>EBH</td>
<td>Cache</td>
<td>3rd-level cache: 18MByte, 24-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>ECH</td>
<td>Cache</td>
<td>3rd-level cache: 24MByte, 24-way set associative, 64 byte line size.</td>
</tr>
<tr>
<td>F0H</td>
<td>Prefetch</td>
<td>64-Byte prefetching.</td>
</tr>
<tr>
<td>F1H</td>
<td>Prefetch</td>
<td>128-Byte prefetching.</td>
</tr>
<tr>
<td>FEH</td>
<td>General</td>
<td>CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.</td>
</tr>
<tr>
<td>FFH</td>
<td>General</td>
<td>CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters.</td>
</tr>
</tbody>
</table>
Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

\[
\begin{align*}
EAX &= 66 5B 50 01H \\
EBX &= 0H \\
ECX &= 0H \\
EDX &= 00 7A 70 00H
\end{align*}
\]

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-μop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

\[
\text{This Cache Size in Bytes} = (Ways + 1) \times (Partitions + 1) \times (Line\_Size + 1) \times (Sets + 1)
\]

\[
= (EBX[31:22] + 1) \times (EBX[21:12] + 1) \times (EBX[11:0] + 1) \times (ECX + 1)
\]

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 9, "Multiple-Processor Management," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.
**INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information**
When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.
When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

**INPUT EAX = 09H: Returns Direct Cache Access Information**
When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

**INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features**
When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.
For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 24, “Introduction to Virtual Machine Extensions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

**INPUT EAX = 0BH: Returns Extended Topology Information**
CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.
When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

**INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information**
When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.
When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:
For i = 2 to 62 // sub-leaf 1 is reserved
  IF (CPUID.(EAX=0DH, ECX=0H):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
    Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
  FI;

**INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information**
When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.
When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.
INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information
When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 12H: Returns Intel SGX Enumeration Information
When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information
When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information
When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

INPUT EAX = 16H: Returns Processor Frequency Information
When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

INPUT EAX = 17H: Returns System-On-Chip Information
When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information
When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

INPUT EAX = 19H: Returns Key Locker Information
When CPUID executes with EAX set to 19H, the processor returns information about Key Locker. See Table 3-8.

INPUT EAX = 1AH: Returns Native Model ID Information
When CPUID executes with EAX set to 1AH, the processor returns information about Native Model Identification. See Table 3-8.

INPUT EAX = 1BH: Returns PCONFIG Information
When CPUID executes with EAX set to 1BH, the processor returns information about PCONFIG capabilities. This information is enumerated in sub-leaves selected by the value of ECX (starting with 0).
Each sub-leaf of CPUID function 1BH enumerates its **sub-leaf type** in EAX. If a sub-leaf type is 0, the sub-leaf is invalid and zero is returned in EBX, ECX, and EDX. In this case, all subsequent sub-leaves (selected by larger input values of ECX) are also invalid.

The only valid sub-type currently defined is 1, indicating that the sub-leaf enumerates target identifiers for the PCONFIG instruction. Any non-zero value returned in EBX, ECX, or EDX indicates a valid target identifier of the PCONFIG instruction (any value of zero should be ignored). The only target identifier currently defined is 1, indicating TME-MK. See the “PCONFIG—Platform Configuration” instruction in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for more information.

**INPUT EAX = 1CH: Returns Last Branch Record Information**

When CPUID executes with EAX set to 1CH, the processor returns information about LBRs (the architectural feature). See Table 3-8.

**INPUT EAX = 1DH: Returns Tile Information**

When CPUID executes with EAX set to 1DH and ECX = 0H, the processor returns information about tile architecture. See Table 3-8.

When CPUID executes with EAX set to 1DH and ECX = 1H, the processor returns information about tile palette 1. See Table 3-8.

**INPUT EAX = 1EH: Returns TMUL Information**

When CPUID executes with EAX set to 1EH and ECX = 0H, the processor returns information about TMUL capabilities. See Table 3-8.

**INPUT EAX = 1FH: Returns V2 Extended Topology Information**

When CPUID executes with EAX set to 1FH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 1FH by verifying (a) the highest leaf index supported by CPUID is >= 1FH, and (b) CPUID.1FH:EBX[15:0] reports a non-zero value. See Table 3-8.

**INPUT EAX = 20H: Returns History Reset Information**

When CPUID executes with EAX set to 20H, the processor returns information about History Reset. See Table 3-8.

**METHODS FOR RETURNING BRANDING INFORMATION**

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 20 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

**The Processor Brand String Method**

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.
How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 80000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-13. Processor Brand String Returned with Pentium 4 Processor

<table>
<thead>
<tr>
<th>EAX Input Value</th>
<th>Return Values</th>
<th>ASCII Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000002H</td>
<td>EAX = 20200200H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EBX = 20200200H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECX = 20200200H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDX = 6E492020H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;nI  &quot;</td>
<td></td>
</tr>
<tr>
<td>80000003H</td>
<td>EAX = 286C6574H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EBX = 50202952H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECX = 69746E65H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDX = 52286D75H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;(let&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;P )R&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;itne&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;R(mu&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

<table>
<thead>
<tr>
<th>EAX Input Value</th>
<th>Return Values</th>
<th>ASCII Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000004H</td>
<td>EAX = 20342029H</td>
<td>“ 4 )”</td>
</tr>
<tr>
<td></td>
<td>EBX = 20555043H</td>
<td>“ UPC”</td>
</tr>
<tr>
<td></td>
<td>ECX = 30303531H</td>
<td>“0051”</td>
</tr>
<tr>
<td></td>
<td>EDX = 007A484DH</td>
<td>“0zHM”</td>
</tr>
</tbody>
</table>

Table 3-14 shows brand indices that have identification strings associated with them.

**The Processor Brand Index Method**

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-14 shows brand indices that have identification strings associated with them.

---

**Figure 3-10. Algorithm for Extracting Processor Frequency**

**Table 3-13. Processor Brand String Returned with Pentium 4 Processor (Contd.)**

<table>
<thead>
<tr>
<th>EAX Input Value</th>
<th>Return Values</th>
<th>ASCII Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF Substring Matched</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>Determine “Freq” and “Multiplier”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF “zHM”</td>
<td></td>
<td>Multiplier = 1 x 10^12</td>
</tr>
<tr>
<td>IF “zHG”</td>
<td></td>
<td>Multiplier = 1 x 10^9</td>
</tr>
<tr>
<td>IF “zHT”</td>
<td></td>
<td>Multiplier = 1 x 10^6</td>
</tr>
<tr>
<td>Report Error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan Digits Until Blank In Reverse Order</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse Digits To Decimal Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Freq</em> = X.YZ if Digits = “ZY.X”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

Operation

IA32_BIOS_SIGN_ID MSR := Update with installed microcode revision number;

CASE (EAX) OF
  EAX = 0:
    EAX := Highest basic function input value understood by CPUID;
    EBX := Vendor identification string;
    EDX := Vendor identification string;
    ECX := Vendor identification string;
   BREAK;
  EAX = 1H:
    EAX[3:0] := Stepping ID;
    EAX[7:4] := Model;

Table 3-14. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

<table>
<thead>
<tr>
<th>Brand Index</th>
<th>Brand String</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>This processor does not support the brand identification feature</td>
</tr>
<tr>
<td>01H</td>
<td>Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>02H</td>
<td>Intel(R) Pentium(R) III processor¹</td>
</tr>
<tr>
<td>03H</td>
<td>Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor</td>
</tr>
<tr>
<td>04H</td>
<td>Intel(R) Pentium(R) III processor</td>
</tr>
<tr>
<td>06H</td>
<td>Mobile Intel(R) Pentium(R) III processor-M</td>
</tr>
<tr>
<td>07H</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>08H</td>
<td>Intel(R) Pentium(R) 4 processor</td>
</tr>
<tr>
<td>09H</td>
<td>Intel(R) Pentium(R) 4 processor</td>
</tr>
<tr>
<td>0AH</td>
<td>Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>0BH</td>
<td>Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP</td>
</tr>
<tr>
<td>0CH</td>
<td>Intel(R) Xeon(R) processor MP</td>
</tr>
<tr>
<td>0EH</td>
<td>Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor</td>
</tr>
<tr>
<td>0FH</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>11H</td>
<td>Mobile Genuine Intel(R) processor</td>
</tr>
<tr>
<td>12H</td>
<td>Intel(R) Celeron(R) M processor</td>
</tr>
<tr>
<td>13H</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>14H</td>
<td>Intel(R) Celeron(R) processor</td>
</tr>
<tr>
<td>15H</td>
<td>Mobile Genuine Intel(R) processor</td>
</tr>
<tr>
<td>16H</td>
<td>Intel(R) Pentium(R) M processor</td>
</tr>
<tr>
<td>17H</td>
<td>Mobile Intel(R) Celeron(R) processor¹</td>
</tr>
<tr>
<td>18H - 0FFH</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

NOTES:
1. Indicates versions of these processors that were introduced after the Pentium III
EAX[13:12] := Processor type;
EAX[15:14] := Reserved;
EAX[19:16] := Extended Model;
EAX[27:20] := Extended Family;
EAX[31:28] := Reserved;
EBX[7:0] := Brand Index; (* Reserved if the value is zero. *)
EBX[15:8] := CLFLUSH Line Size;
EBX[16:23] := Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
EBX[24:31] := Initial APIC ID;
ECX := Feature flags; (* See Figure 3-7. *)
EDX := Feature flags; (* See Figure 3-8. *)

BREAK;

EAX = 2H:
EAX := Cache and TLB information;
EBX := Cache and TLB information;
ECX := Cache and TLB information;
EDX := Cache and TLB information;

BREAK;

EAX = 3H:
EAX := Reserved;
EBX := Reserved;
ECX := ProcessorSerialNumber[31:0];
(* Pentium III processors only, otherwise reserved. *)
EDX := ProcessorSerialNumber[63:32];
(* Pentium III processors only, otherwise reserved. *)

BREAK

EAX = 4H:
EAX := Deterministic Cache Parameters Leaf; (* See Table 3-8. *)
EBX := Deterministic Cache Parameters Leaf;
ECX := Deterministic Cache Parameters Leaf;
EDX := Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:
EAX := MONITOR/MWAIT Leaf; (* See Table 3-8. *)
EBX := MONITOR/MWAIT Leaf;
ECX := MONITOR/MWAIT Leaf;
EDX := MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:
EAX := Thermal and Power Management Leaf; (* See Table 3-8. *)
EBX := Thermal and Power Management Leaf;
ECX := Thermal and Power Management Leaf;
EDX := Thermal and Power Management Leaf;

BREAK;

EAX = 7H:
EAX := Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-8. *)
EBX := Structured Extended Feature Flags Enumeration Leaf;
ECX := Structured Extended Feature Flags Enumeration Leaf;
EDX := Structured Extended Feature Flags Enumeration Leaf;

BREAK;

EAX = 8H:
EAX := Reserved = 0;
EBX := Reserved = 0;
ECX := Reserved = 0;
EDX := Reserved = 0;
BREAK;

EAX = 9H:
    EAX := Direct Cache Access Information Leaf; (* See Table 3-8. *)
    EBX := Direct Cache Access Information Leaf;
    ECX := Direct Cache Access Information Leaf;
    EDX := Direct Cache Access Information Leaf;
    BREAK;

EAX = AH:
    EAX := Architectural Performance Monitoring Leaf; (* See Table 3-8. *)
    EBX := Architectural Performance Monitoring Leaf;
    ECX := Architectural Performance Monitoring Leaf;
    EDX := Architectural Performance Monitoring Leaf;
    BREAK;

EAX = BH:
    EAX := Extended Topology Enumeration Leaf; (* See Table 3-8. *)
    EBX := Extended Topology Enumeration Leaf;
    ECX := Extended Topology Enumeration Leaf;
    EDX := Extended Topology Enumeration Leaf;
    BREAK;

EAX = CH:
    EAX := Reserved = 0;
    EBX := Reserved = 0;
    ECX := Reserved = 0;
    EDX := Reserved = 0;
    BREAK;

EAX = DH:
    EAX := Processor Extended State Enumeration Leaf; (* See Table 3-8. *)
    EBX := Processor Extended State Enumeration Leaf;
    ECX := Processor Extended State Enumeration Leaf;
    EDX := Processor Extended State Enumeration Leaf;
    BREAK;

EAX = EH:
    EAX := Reserved = 0;
    EBX := Reserved = 0;
    ECX := Reserved = 0;
    EDX := Reserved = 0;
    BREAK;

EAX = FH:
    EAX := Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    ECX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    EDX := Intel Resource Director Technology Monitoring Enumeration Leaf;
    BREAK;

EAX = 10H:
    EAX := Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel Resource Director Technology Allocation Enumeration Leaf;
    ECX := Intel Resource Director Technology Allocation Enumeration Leaf;
    EDX := Intel Resource Director Technology Allocation Enumeration Leaf;
    BREAK;

EAX = 12H:
    EAX := Intel SGX Enumeration Leaf; (* See Table 3-8. *)
    EBX := Intel SGX Enumeration Leaf;
    ECX := Intel SGX Enumeration Leaf;
    EDX := Intel SGX Enumeration Leaf;
EDX := Intel SGX Enumeration Leaf;
BREAK;
EAX = 14H:
   EAX := Intel Processor Trace Enumeration Leaf; (* See Table 3-8. *)
   EBX := Intel Processor Trace Enumeration Leaf;
   ECX := Intel Processor Trace Enumeration Leaf;
   EDX := Intel Processor Trace Enumeration Leaf;
BREAK;
EAX = 15H:
   EAX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (* See Table 3-8. *)
   EBX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
   ECX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
   EDX := Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
BREAK;
EAX = 16H:
   EAX := Processor Frequency Information Enumeration Leaf; (* See Table 3-8. *)
   EBX := Processor Frequency Information Enumeration Leaf;
   ECX := Processor Frequency Information Enumeration Leaf;
   EDX := Processor Frequency Information Enumeration Leaf;
BREAK;
EAX = 17H:
   EAX := System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 3-8. *)
   EBX := System-On-Chip Vendor Attribute Enumeration Leaf;
   ECX := System-On-Chip Vendor Attribute Enumeration Leaf;
   EDX := System-On-Chip Vendor Attribute Enumeration Leaf;
BREAK;
EAX = 18H:
   EAX := Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 3-8. *)
   EBX := Deterministic Address Translation Parameters Enumeration Leaf;
   ECX := Deterministic Address Translation Parameters Enumeration Leaf;
   EDX := Deterministic Address Translation Parameters Enumeration Leaf;
BREAK;
EAX = 19H:
   EAX := Key Locker Enumeration Leaf; (* See Table 3-8. *)
   EBX := Key Locker Enumeration Leaf;
   ECX := Key Locker Enumeration Leaf;
   EDX := Key Locker Enumeration Leaf;
BREAK;
EAX = 1AH:
   EAX := Native Model ID Enumeration Leaf; (* See Table 3-8. *)
   EBX := Native Model ID Enumeration Leaf;
   ECX := Native Model ID Enumeration Leaf;
   EDX := Native Model ID Enumeration Leaf;
BREAK;
EAX = 1BH:
   EAX := PCONFIG Information Enumeration Leaf; (* See “INPUT EAX = 1BH: Returns PCONFIG Information” on page 3-253. *)
   EBX := PCONFIG Information Enumeration Leaf;
   ECX := PCONFIG Information Enumeration Leaf;
   EDX := PCONFIG Information Enumeration Leaf;
BREAK;
EAX = 1CH:
   EAX := Last Branch Record Information Enumeration Leaf; (* See Table 3-8. *)
   EBX := Last Branch Record Information Enumeration Leaf;
   ECX := Last Branch Record Information Enumeration Leaf;
EDX := Last Branch Record Information Enumeration Leaf;
BREAK;

EAX = 1DH:
   EAX := Tile Information Enumeration Leaf; (* See Table 3-8. *)
   EBX := Tile Information Enumeration Leaf;
   ECX := Tile Information Enumeration Leaf;
   EDX := Tile Information Enumeration Leaf;
BREAK;

EAX = 1EH:
   EAX := TMUL Information Enumeration Leaf; (* See Table 3-8. *)
   EBX := TMUL Information Enumeration Leaf;
   ECX := TMUL Information Enumeration Leaf;
   EDX := TMUL Information Enumeration Leaf;
BREAK;

EAX = 1FH:
   EAX := V2 Extended Topology Enumeration Leaf; (* See Table 3-8. *)
   EBX := V2 Extended Topology Enumeration Leaf;
   ECX := V2 Extended Topology Enumeration Leaf;
   EDX := V2 Extended Topology Enumeration Leaf;
BREAK;

EAX = 20H:
   EAX := Processor History Reset Sub-leaf; (* See Table 3-8. *)
   EBX := Processor History Reset Sub-leaf;
   ECX := Processor History Reset Sub-leaf;
   EDX := Processor History Reset Sub-leaf;
BREAK;

EAX = 80000000H:
   EAX := Highest extended function input value understood by CPUID;
   EBX := Reserved;
   ECX := Reserved;
   EDX := Reserved;
BREAK;

EAX = 80000001H:
   EAX := Reserved;
   EBX := Reserved;
   ECX := Extended Feature Bits (* See Table 3-8. *);
   EDX := Extended Feature Bits (* See Table 3-8. *);
BREAK;

EAX = 80000002H:
   EAX := Processor Brand String;
   EBX := Processor Brand String, continued;
   ECX := Processor Brand String, continued;
   EDX := Processor Brand String, continued;
BREAK;

EAX = 80000003H:
   EAX := Processor Brand String, continued;
   EBX := Processor Brand String, continued;
   ECX := Processor Brand String, continued;
   EDX := Processor Brand String, continued;
BREAK;

EAX = 80000004H:
   EAX := Processor Brand String, continued;
   EBX := Processor Brand String, continued;
   ECX := Processor Brand String, continued;
EDX := Processor Brand String, continued;
BREAK;

EAX = 80000005H:
  EAX := Reserved = 0;
  EBX := Reserved = 0;
  ECX := Reserved = 0;
  EDX := Reserved = 0;
BREAK;

EAX = 80000006H:
  EAX := Reserved = 0;
  EBX := Reserved = 0;
  ECX := Cache information;
  EDX := Reserved = 0;
BREAK;

EAX = 80000007H:
  EAX := Reserved = 0;
  EBX := Reserved = 0;
  ECX := Reserved = 0;
  EDX := Reserved = Misc Feature Flags;
BREAK;

EAX = 80000008H:
  EAX := Address Size Information;
  EBX := Misc Feature Flags;
  ECX := Reserved = 0;
  EDX := Reserved = 0;
BREAK;

EAX >= 40000000H and EAX <= 4FFFFFFFH:
DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
(* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
  EAX := Reserved; (* Information returned for highest basic information leaf. *)
  EBX := Reserved; (* Information returned for highest basic information leaf. *)
  ECX := Reserved; (* Information returned for highest basic information leaf. *)
  EDX := Reserved; (* Information returned for highest basic information leaf. *)

BREAK;
ESAC;

Flags Affected
None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.
CRC32—Accumulate CRC32 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 OF 38 F0 /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Accumulate CRC32 on r/m8.</td>
</tr>
<tr>
<td>CRC32 r32, r/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX OF 38 F0 /r</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Accumulate CRC32 on r/m8.</td>
</tr>
<tr>
<td>CRC32 r32, r/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 OF 38 F1 /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Accumulate CRC32 on r/m16.</td>
</tr>
<tr>
<td>CRC32 r32, r/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 OF 38 F1 /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Accumulate CRC32 on r/m32.</td>
</tr>
<tr>
<td>CRC32 r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX.W OF 38 F0 /r</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Accumulate CRC32 on r/m8.</td>
</tr>
<tr>
<td>CRC32 r64, r/m8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX.W OF 38 F1 /r</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Accumulate CRC32 on r/m64.</td>
</tr>
<tr>
<td>CRC32 r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

Operation

Notes:

- BIT_REFLECT64: DST[63-0] = SRC[0-63]
- BIT_REFLECT32: DST[31-0] = SRC[0-31]
- BIT_REFLECT16: DST[15-0] = SRC[0-15]
- BIT_REFLECT8: DST[7-0] = SRC[0-7]
- MOD2: Remainder from Polynomial division modulus 2
CRC32 instruction for 64-bit source operand and 64-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[63-0] & := \text{BIT\_REFLECT64}(\text{SRC}[63-0]) \\
\text{TEMP2}[31-0] & := \text{BIT\_REFLECT32}(\text{DEST}[31-0]) \\
\text{TEMP3}[95-0] & := \text{TEMP1}[63-0] \times 32 \\
\text{TEMP4}[95-0] & := \text{TEMP2}[31-0] \times 64 \\
\text{TEMP5}[95-0] & := \text{TEMP3}[95-0] \text{ XOR } \text{TEMP4}[95-0] \\
\text{TEMP6}[31-0] & := \text{TEMP5}[95-0] \text{ MOD2 11EDC6F41H} \\
\text{DEST}[31-0] & := \text{BIT\_REFLECT}(\text{TEMP6}[31-0]) \\
\text{DEST}[63-32] & := \text{00000000H}
\end{align*}
\]

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[31-0] & := \text{BIT\_REFLECT32}(\text{SRC}[31-0]) \\
\text{TEMP2}[31-0] & := \text{BIT\_REFLECT32}(\text{DEST}[31-0]) \\
\text{TEMP3}[63-0] & := \text{TEMP1}[31-0] \times 32 \\
\text{TEMP4}[63-0] & := \text{TEMP2}[31-0] \times 32 \\
\text{TEMP5}[63-0] & := \text{TEMP3}[63-0] \text{ XOR } \text{TEMP4}[63-0] \\
\text{TEMP6}[31-0] & := \text{TEMP5}[63-0] \text{ MOD2 11EDC6F41H} \\
\text{DEST}[31-0] & := \text{BIT\_REFLECT}(\text{TEMP6}[31-0]) \\
\text{DEST}[31-0] & := \text{BIT\_REFLECT}(\text{TEMP6}[31-0])
\end{align*}
\]

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[15-0] & := \text{BIT\_REFLECT16}(\text{SRC}[15-0]) \\
\text{TEMP2}[31-0] & := \text{BIT\_REFLECT32}(\text{DEST}[31-0]) \\
\text{TEMP3}[47-0] & := \text{TEMP1}[15-0] \times 32 \\
\text{TEMP4}[47-0] & := \text{TEMP2}[31-0] \times 16 \\
\text{TEMP5}[47-0] & := \text{TEMP3}[47-0] \text{ XOR } \text{TEMP4}[47-0] \\
\text{TEMP6}[31-0] & := \text{TEMP5}[47-0] \text{ MOD2 11EDC6F41H} \\
\text{DEST}[31-0] & := \text{BIT\_REFLECT}(\text{TEMP6}[31-0])
\end{align*}
\]

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[7-0] & := \text{BIT\_REFLECT8}(\text{SRC}[7-0]) \\
\text{TEMP2}[31-0] & := \text{BIT\_REFLECT32}(\text{DEST}[31-0]) \\
\text{TEMP3}[39-0] & := \text{TEMP1}[7-0] \times 32 \\
\text{TEMP4}[39-0] & := \text{TEMP2}[31-0] \times 8 \\
\text{TEMP5}[39-0] & := \text{TEMP3}[39-0] \text{ XOR } \text{TEMP4}[39-0] \\
\text{TEMP6}[31-0] & := \text{TEMP5}[39-0] \text{ MOD2 11EDC6F41H} \\
\text{DEST}[31-0] & := \text{BIT\_REFLECT}(\text{TEMP6}[31-0]) \\
\text{DEST}[63-32] & := \text{00000000H}
\end{align*}
\]

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

\[
\begin{align*}
\text{TEMP1}[7-0] & := \text{BIT\_REFLECT8}(\text{SRC}[7-0]) \\
\text{TEMP2}[31-0] & := \text{BIT\_REFLECT32}(\text{DEST}[31-0]) \\
\text{TEMP3}[39-0] & := \text{TEMP1}[7-0] \times 32 \\
\text{TEMP4}[39-0] & := \text{TEMP2}[31-0] \times 8 \\
\text{TEMP5}[39-0] & := \text{TEMP3}[39-0] \text{ XOR } \text{TEMP4}[39-0] \\
\text{TEMP6}[31-0] & := \text{TEMP5}[39-0] \text{ MOD2 11EDC6F41H} \\
\text{DEST}[31-0] & := \text{BIT\_REFLECT}(\text{TEMP6}[31-0])
\end{align*}
\]

**Flags Affected**

None.
Intel C/C++ Compiler Intrinsic Equivalent

unsigned int _mm_crc32_u8( unsigned int crc, unsigned char data )
unsigned int _mm_crc32_u16( unsigned int crc, unsigned short data )
unsigned int _mm_crc32_u32( unsigned int crc, unsigned int data )
unsigned __int64 _mm_crc32_u64( unsigned __int64 crc, unsigned __int64 data )

SIMD Floating-Point Exceptions
None.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
If LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.SSE4_2[Bit 20] = 0.
If LOCK prefix is used.
**CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:W1G E6 /r VCVTDQ2PD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed signed doubleword integers from xmm2/mem to two packed double precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:W1G E6 /r VCVTDQ2PD ymm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed signed doubleword integers from xmm2/mem to four packed double precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:W0 E6 /r VCVTDQ2PD xmm1[k1]{z}, xmm2/m64/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert 2 packed signed doubleword integers from xmm2/m64/m32bcst to eight packed double precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:W0 E6 /r VCVTDQ2PD ymm1[k1]{z}, xmm2/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:W0 E6 /r VCVTDQ2PD zmm1[k1]{z}, ymm2/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double precision floating-point values in the destination operand (the first operand).

**EVEX encoded versions:** The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

**VEX.256 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

**VEX.128 encoded version:** The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Operation
VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Figure 3-11. CVDQ2PD (VEX.256 encoded version)
VCVTDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+63:i] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
    ELSE
      DEST[i+63:i] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
    FI;
  ELSE
    IF *merging-masking*; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTDQ2PD (VEX.256 Encoded Version)
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] := 0

VCVTDQ2PD (VEX.128 Encoded Version)
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0

CVTDQ2PD (128-bit Legacy SSE Version)
DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTDQ2PD _mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD _mm512_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD _mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD _mm256_cvtepi32_pd ( __m128i src);
VCVTDQ2PD _mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD _mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD _mm128_cvtepi32_pd ( __m128i src);
VCVTDQ2PD _mm128_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD _mm128_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
VCVTDQ2PD __m128d _mm_cvtepi32_pd ( __m128i src)
Other Exceptions
VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-51, “Type E5 Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>CVTDQ2PS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.0F:WIG 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed signed doubleword integers from xmm2/mem to four packed single precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VCVDQ2PS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.0F:WIG 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert eight packed signed doubleword integers from ymm2/mem to eight packed single precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>VCVDQ2PS ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.0F:W0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVDQ2PS xmm1 [k1]%1111b,</td>
<td>B</td>
<td>V/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmm2/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F:W0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVDQ2PS ymm1 [k1]%1111b,</td>
<td>B</td>
<td>V/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ymm2/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F:W0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVDQ2PS zmm1 [k1]%1111b,</td>
<td>B</td>
<td>V/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zmm2/m512/m32bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single precision floating-point values in the destination operand.

**EVEX encoded versions:** The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Operation

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Register
(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1
    FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VCVTDQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] := 0
                FI
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
VCVTDQ2PS (VEX.256 Encoded Version)
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[159:128] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224])
DEST[MAXVL-1:256] := 0

VCVTDQ2PS (VEX.128 Encoded Version)
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:128] := 0

CVTDQ2PS (128-bit Legacy SSE Version)
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVDQ2PS __m512 _mm512_cvtepi32_ps( __m512i a);
VCVDQ2PS __m512 _mm512_mask_cvtepi32_ps( __m512 s, __mmask16 k, __m512i a);
VCVDQ2PS __m512 _mm512_maskz_cvtepi32_ps(____mmask16 k, __m512i a);
VCVDQ2PS __m512 _mm512_cvt_roundepi32_ps(____m512i a, int r);
VCVDQ2PS __m512 _mm512_maskz_cvt_roundepi32_ps(____mmask16 k, __m512i a, int r);
VCVDQ2PS __m256 _mm256_cvtepi32_ps(____m256i src);
VCVDQ2PS __m128 _mm_cvtepi32_ps(____m128i src);

SIMD Floating-Point Exceptions
Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTPD2DQ—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F2.0F:Wig E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F2.0F:Wig E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W1 E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W1 E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W1 E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts packed double precision floating-point values in the source operand (second operand) to packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ($2^w-1$, where $w$ represents the number of bits in the destination format) is returned.

EVE encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Operation

VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    END;
DEST[MAXVL-1:VL/2] := 0

Figure 3-12. VCVTPD2DQ (VEX.256 encoded version)
VCVTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
        ELSE
          DEST[i+31:i] :=
          Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL/2] := 0

VCVTPD2DQ (VEX.256 Encoded Version)
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192])
DEST[MAXVL-1:128] := 0

VCVTPD2DQ (VEX.128 Encoded Version)
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64] := 0

CVTPD2DQ (128-bit Legacy SSE Version)
DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ __m256i _mm512_cvtpd_epi32( __m512d a);
CVTPD2DQ __m256i _mm512_mask_cvtpd_epi32( __m256i s, __mmask8 k, __m512d a);
CVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32( __mmask8 k, __m512d a);
CVTPD2DQ __m256i _mm512_cvt_roundpd_epi32( __m512d a, int r);
CVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int r);
CVTPD2DQ __m128i _mm256_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m256d a);
CVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32( __mmask8 k, __m256d a);
CVTPD2DQ __m128i _mm_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m128d a);
CVTPD2DQ __m128i _mm_maskz_cvtpd_epi32( __mmask8 k, __m128d a);
CVTPD2DQ __m128i _mm256_cvtpd_epi32 ( __m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32 ( __m128d src)

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
See Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
**CVTPD2PI—Convert Packed Double Precision Floating-Point Values to Packed Dword Integers**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2D /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts two packed double precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);

**Intel C/C++ Compiler Intrinsic Equivalent**

CVTPD2PI _mm_cvtpd_pi32(_m128d a)

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Table 23-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTPD2PS—Convert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>CVTPD2PS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed double precision floating-point values in xmm2/mem to two single precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VCVTPD2PS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed double precision floating-point values in ymm2/mem to four single precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VCVTPD2PS xmm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two single precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2PS xmm1[k1][z], xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four single precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2PS xmm1[k1][z], ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight single precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2PS ymm1[k1][z], zmm2/m512/m64bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight packed single precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Register**

KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

k := j * 64

IF k1[j] OR *no writemask*

THEN

DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 0

---

**Figure 3-13. VCVTPD2PS (VEX.256 encoded version)**
VCVTPD2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
      THEN
      IF (EVEX.b = 1)
          THEN
              DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
          ELSE
              DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
      FI;
      ELSE
      IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
          DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

VCVTPD2PS (VEX.256 Encoded Version)
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
DEST[MAXVL-1:128] := 0

VCVTPD2PS (VEX.128 Encoded Version)
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] := 0

CVTPD2PS (128-bit Legacy SSE Version)
DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PS __m256 _mm512_cvtpd_ps(__m512d a);
VCVTPD2PS __m256 _mm512_mask_cvtpd_ps(__m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_maskz_cvtpd_ps(__mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_cvtrndpd_ps(__m512d a, int r);
VCVTPD2PS __m256 _mm512_mask_cvtrndpd_ps(__m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 _mm512_maskz_cvtrndpd_ps(__mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 _mm256_cvtpd_ps(__m256d a);

VCVTPD2PS __m128 _mm_cvtpd_ps(__m128d a)

SIMD Floating-Point Exceptions
Invalid, Precision, Underflow, Overflow, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTPI2PD—Convert Packed Dword Integers to Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2A /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two packed signed doubleword integers from mm/mem64 to two packed double precision floating-point values in xmm.</td>
</tr>
</tbody>
</table>

NOTES:
1. Operation is different for different operand sets; see the Description section.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands xmm, mm**: the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.

- **For operands xmm, m64**: the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

\[
\text{DEST}[63:0] := \text{Convert Integer To Double Precision Floating Point}(\text{SRC}[31:0]); \\
\text{DEST}[127:64] := \text{Convert Integer To Double Precision Floating Point}(\text{SRC}[63:32]);
\]

### Intel C/C++ Compiler Intrinsic Equivalent

`CVTPI2PD __m128d _mm_cvtpi32_pd(__m64 a)`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 23-6, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTPI2PS—Convert Packed Dword Integers to Packed Single Precision Floating-Point Values

Opcode/Instruction: NP 0F 2A /r
CVTPI2PS xmm, mm/m64

Op/En: RM
64-Bit Mode: Valid
Compat/Leg Mode: Valid
Description: Convert two signed doubleword integers from mm/m64 to two single precision floating-point values in xmm.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Converts two packed signed doubleword integers in the source operand (second operand) to two packed single precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation
DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)

Intel C/C++ Compiler Intrinsic Equivalent
CVTPI2PS _m128 _mm_cvtpi32_ps(_m128 a, _m64 b)

SIMD Floating-Point Exceptions
Precision.

Other Exceptions
See Table 23-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
**CVTPS2DQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 5B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 5B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \((2^w-1\), where \(w\) represents the number of bits in the destination format) is returned.

**EVEX encoded versions:** The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.**
Operation
VCVTPS2DQ (Encoded Versions) When SRC Operand is a Register
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
                        Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
                ELSE
                    DEST[i+31:i] :=
                        Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTPS2DQ (VEX.256 Encoded Version)
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])

VCVTPS2DQ (VEX.128 Encoded Version)
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] := 0

CVTPS2DQ (128-bit Legacy SSE Version)
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2DQ __m512i _mm512_cvtps_epi32( __m512 a);
VCVTPS2DQ __m512i _mm512_mask_cvtps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_maskz_cvtps_epi32( __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_cvtps_roundps_epi32( __m512 a, int r);
VCVTPS2DQ __m128i _mm128_cvtps_epi32( __m128 a);
VCVTPS2DQ __m256i _mm256_cvtps_epi32( __m256 a);
VCVTPS2DQ __m256i _mm256_mask_cvtps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2DQ __m128i _mm_mask_cvtps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i _mm_maskz_cvtps_epi32( __mmask8 k, __m128 a);
CVTPS2DQ __m256i _mm256_cvtps_epi32( __m256 a)

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If VEX.vvv != 1111B or EVEX.vvv != 1111B.
CVTPS2PD—Convert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed single precision floating-point values in xmm2/m64 to two packed double precision floating-point values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed single precision floating-point values in xmm2/m128 to four packed double precision floating-point values in ymm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WO 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed single precision floating-point values in xmm2/m64/m32bcst to packed double precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VEX.256.0F.WO 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed double precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>VEX.512.0F.WO 5A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed single precision floating-point values in ymm2/m256/b32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

**Description**

Converts two, four or eight packed single precision floating-point values in the source operand (second operand) to two, four or eight packed double precision floating-point values in the destination operand (first operand).

**EVEX encoded versions:** The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPS2PD (EVEX Encoded Versions) When SRC Operand is a Register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
    ELSE IF *merging-masking* ; merging-masking
      THEN DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VCVTPS2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN IF (EVEX.b = 1)
      THEN DEST[i+63:i] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
      ELSE FI
    ELSE Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
    ELSE FI
ENDFOR
IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+63:i] := 0
FI

ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTPS2PD (VEX.256 Encoded Version)
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] := 0

VCVTPS2PD (VEX.128 Encoded Version)
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] := 0

CVTPS2PD (128-bit Legacy SSE Version)
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2PD __m512d _mm512_cvtps_pd(__m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd(__m512d s, __mmask8 k, __m512d a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd(__mmask8 k, __m512d a);
VCVTPS2PD __m512d _mm512_cvtps_roundpd(__m256 a, int sae);
VCVTPS2PD __m512d _mm512_mask_cvtps_roundpd(__m512d s, __mmask8 k, __m512d a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvtps_roundpd(__mmask8 k, __m512d a, int sae);
VCVTPS2PD __m256d _mm256_cvtps_pd(__m128d a);
VCVTPS2PD __m128d _mm128_cvtps_pd(__m256d s, __mmask8 k, __m128d a);
VCVTPS2PD __m128d _mm128_cvtps_roundpd(__mmask8 k, __m128d a);
VCVTPS2PD __m128d _mm128_mask_cvtps_pd(__m128d s, __mmask8 k, __m128d a);
VCVTPS2PD __m128d _mm128_maskz_cvtps_pd(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2D /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two packed single precision floating-point values from xmm/m64 to two packed signed doubleword integers in mm.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts two packed single precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

\[
\text{DEST}[31:0] := \text{Convert SINGLE PRECISION Floating Point To Integer}(\text{SRC}[31:0]); \\
\text{DEST}[63:32] := \text{Convert SINGLE PRECISION Floating Point To Integer}(\text{SRC}[63:32]); \\
\]

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTPS2PI __m64 _mm_cvtps_pi32(__m128 a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 23-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 2D  Ir</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.</td>
</tr>
<tr>
<td>CVTSD2SI r32, xmm1/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W0 2D  Ir</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.</td>
</tr>
<tr>
<td>CVTSD2SI r32, xmm1/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W0 2D  Ir</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer r32.</td>
</tr>
<tr>
<td>CVTSD2SI r32, xmm1/m64</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.</td>
</tr>
</tbody>
</table>

### NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VCVTSD2SI (EVEX Encoded Version)
IF SRC is register AND (EVEX.b = 1)
THEN
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
THEN
   DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
ELSE
   DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI

(V)CVTSD2SI
IF 64-Bit Mode and OperandSize = 64
THEN
   DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
ELSE
   DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvtsd_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvtsd_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTSD2SS—Convert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one double precision floating-point value in xmm2/m64 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value in xmm3/m64 to one single precision floating-point value and merge with high bits in xmm2 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts a double precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single precision floating-point value in the destination operand.

When the “convert-from” operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
**Operation**

**VCVTSD2SS (EVEX Encoded Version)**

IF (SRC2 *is register*) AND (EVEX.b = 1)
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI;

IF k1[0] or *no writemask*
   THEN DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[31:0] remains unchanged*
       ELSE ; zeroing-masking
           THEN DEST[31:0] := 0
       FI;
   FI;

DEST[MAXVL-1:128] := 0

**VCVTSD2SS (VEX.128 Encoded Version)**

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[MAXVL-1:128] := 0

**VCVTSD2SS (128-bit Legacy SSE Version)**

DEST[31:0] := Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[MAXVL-1:32] Unmodified *)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2SS __m128 __mm_mask_cvtss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 __mm_maskz_cvtss( __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 __m128_cvt_roundsd__ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 __mm_mask_cvt_roundsd__ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 __mm_maskz_cvt_roundsd__ss( __mmask8 k, __m128 a, __m128d b, int r);
CVTSD2SS __m128_mm_cvt_roundsd__ss( __m128 a, __m128d b)

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
### CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 2A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one signed doubleword integer from r32/m32 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>CVTSI2SD xmm1, r32/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 REX.W 0F 2A /r</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>CVTSI2SD xmm1, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W0 2A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VCVTSD2SD xmm1, xmm2, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 2A /r</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VCVTSD2SD xmm1, xmm2, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W0 2A /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one signed doubleword integer from r/m32 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VCVTSD2SD xmm1, xmm2, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 2A /r</td>
<td>C</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one signed quadword integer from r/m64 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VCVTSD2SD xmm1, xmm2, r/m64{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### NOTES:
1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the "convert-from" source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSD2SD is encoded with VEX.L=0. Encoding VCVTSD2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VCVTSI2SD (EVEX Encoded Version)
IF (SRC2 *is register*) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
IF 64-Bit Mode And OperandSize = 64
THEN
  DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
  DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VCVTSI2SD (VEX.128 Encoded Version)
IF 64-Bit Mode And OperandSize = 64
THEN
  DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
  DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

CVTSI2SD
IF 64-Bit Mode And OperandSize = 64
THEN
  DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
ELSE
  DEST[63:0] := Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d _mm_cvtsi32_sd(__m128d s, __int64 a, int b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” if W1; else see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions,” if W1; else see Table 2-59, “Type E10NF Class Exception Conditions.”
## CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 2A /r CVTSI2SS xmm1, r/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE</td>
<td>Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W0 2A /r CVCTS2SS xmm1, xmm2, r/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1 2A /r CVTSI2SS xmm1, xmm2, r/m64</td>
<td>B</td>
<td>V/N.E. (^1)</td>
<td>AVX</td>
<td>Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 2A /r CVTSI2SS xmm1, xmm2, r/m32[er]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W1 2A /r CVTSI2SS xmm1, xmm2, r/m64[er]</td>
<td>C</td>
<td>V/N.E. (^1)</td>
<td>AVX512F</td>
<td>Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRM(r/m) (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM(r/m) (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM(r/m) (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the "convert-from" source operand to a single precision floating-point value in the destination operand (first operand). The "convert-from" source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The converted result in written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VCVTSS (EVEX Encoded Version)
IF (SRC2 is register*) AND (EVEX.b = 1)
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI;
IF 64 Bit Mode And OperandSize = 64
   THEN
      DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
   ELSE
      DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
   FI;
DEST[MAXVL-1:128] := 0

VCVTSS (VEX.128 Encoded Version)
IF 64 Bit Mode And OperandSize = 64
   THEN
      DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
   ELSE
      DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
   FI;
DEST[MAXVL-1:128] := 0

VCVTSS (128-bit Legacy SSE Version)
IF 64 Bit Mode And OperandSize = 64
   THEN
      DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
   ELSE
      DEST[31:0] := Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
   FI;
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSS __m128 _mm_cvti32_ss(__m128 s, int a);
VCVTSS __m128 _mm_cvti64_ss(__m128 s, __int64 a);
VCVTSS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
VCVTSS __m128 _mm_cvtsi32_ss(__m128 a, int b);

SIMD Floating-Point Exceptions
Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
CVTSS2SD—Convert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one single precision floating-point value in xmm2/m32 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 5A /r CVTSS2SD xmm1, xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 5A /r CVTSS2SD xmm1 {k1}[z], xmm2, xmm3/m32{sae}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value in xmm3/m32 to one double precision floating-point value and merge with high bits of xmm2 under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts a single precision floating-point value in the "convert-from” source operand to a double precision floating-point value in the destination operand. When the "convert-from” source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The "convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

**VCVTS2SD (EVEX Encoded Version)**

if k1[0] or *no writemask*

THEN

\[ \text{DEST}[63:0] := \text{Convert Single Precision To Double Precision Floating Point}(\text{SRC2}[31:0]) \]

ELSE

\[
\begin{align*}
\text{IF } \text{*merging-masking*} & \quad \text{; merging-masking} \\
\text{THEN } \text{*DEST}[63:0] \text{ remains unchanged*} \\
\text{ELSE} & \quad \text{; zeroing-masking} \\
\text{THEN } \text{DEST}[63:0] = 0 \\
\end{align*}
\]

\[ F; \]

\[ \text{DEST}[127:64] := \text{SRC1}[127:64] \]

\[ \text{DEST}[\text{MAXVL}-1:128] := 0 \]

\]
VCVTSS2SD (VEX.128 Encoded Version)
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VCVTSS2SD (128-bit Legacy SSE Version)
DEST[63:0] := Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a, __m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a, __m128 b);
CVTSS2SD __m128d_mm_cvtsd_sd(__m128d a, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer

**Op/En**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format F3 0F 2D lr</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>CVTSS2SI r32, xmm1/m32</td>
<td>A</td>
<td>N/A</td>
</tr>
<tr>
<td>CVTSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>N/A</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W0 2D lr</td>
<td>A</td>
<td>N/A</td>
</tr>
<tr>
<td>VCTXSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>N/A</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1 2D lr</td>
<td>A</td>
<td>N/A</td>
</tr>
<tr>
<td>VCTXSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 2D lr</td>
<td>B</td>
<td>N/A</td>
</tr>
<tr>
<td>EVXTSS2SI r64, xmm1/m32[er]</td>
<td>B</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W1 2D lr</td>
<td>B</td>
<td>N/A</td>
</tr>
<tr>
<td>EVXTSS2SI r64, xmm1/m32[er]</td>
<td>B</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Op / En**

- A: 64/32 bit Mode Support
- B: 128/64/32 bit Mode Support

**CPUID Feature Flag**

- SSE
- V.E.
- AVX
- AVX-512F

**Description**

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^w-1, where w represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VCVTSS2SI (EVEX Encoded Version)
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;

(V)CVTSS2SI (Legacy and VEX.128 Encoded Version)
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SI int _mm_cvtss_i32( __m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32( __m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64( __m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64( __m128 a, int r);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:
#UD   If VEX.vvvv !. 1111B.
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
### CVTTPD2DQ—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert two packed double precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G E6 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed double precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1E6 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double precision floating-point values in zmm2/m512/m64bcst(sae) to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts two, four or eight packed double precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

**VEX.256 encoded version:** The source operand is a YMM register or 256- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128- bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Operation

VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL/2] := 0
VCVTTPD2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] :=
                Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
        ELSE
            DEST[i+31:i] :=
                Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI
ENDFOR

DEST[MAXVL-1:VL/2] := 0

VCVTTPD2DQ (VEX.256 Encoded Version)

DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[MAXVL-1:128] := 0

VCVTTPD2DQ (VEX.128 Encoded Version)

DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64] := 0

VCVTTPD2DQ (128-bit Legacy SSE Version)

DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] := 0
DEST[MAXVL-1:128] (unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m256i __mm512_cvttpd_epi32(__m512d a);
VCVTTPD2DQ __m256i __mm512_mask_cvttpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i __mm512_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
VCVTTPD2DQ __m256i __mm512_cvtt_roundpd_epi32(__m512d a, int sae);
VCVTTPD2DQ __m256i __mm512_mask_cvtt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i __mm256_mask_cvttpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m128i __mm256_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
VCVTTPD2DQ __m128i __mm256_cvttpd_epi32(__m256d src);
VCVTTPD2DQ __m128i __mm_cvttpd_epi32 (__m128d src);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTTPD2PI—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2C /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two packed double precision floating-point values from xmm/m128 to two packed signed doubleword integers in mm using truncation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts two packed double precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] := Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[127:64]);

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI __m64 __mm_cvtpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Table 23-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTTPS2DQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert four packed single precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.</td>
</tr>
<tr>
<td>VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert eight packed single precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst (sae)</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts four, eight or sixteen packed single precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 encoded version:** The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Register
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
  ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTTPS2DQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO 15
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
          ELSE
            DEST[i+31:i] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
          FI;
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] := 0
          FI
        FI;
      ENDIF
    ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTTPS2DQ (VEX.256 Encoded Version)

DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[159:128] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])
VCVTTPS2DQ (VEX.128 Encoded Version)
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] := 0

VCVTTPS2DQ (128-bit Legacy SSE Version)
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPS2DQ __m512i _mm512_cvttps_epi32( __m512 a);
VCVTTPS2DQ __m512i _mm512_mask_cvttps_epi32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_maskz_cvttps_epi32( __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_cvtt_roundps_epi32( __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_mask_cvtt_roundps_epi32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_maskz_cvtt_roundps_epi32( __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m256i _mm256_mask_cvttps_epi32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2DQ __m256i _mm256_maskz_cvttps_epi32( __mmask8 k, __m256 a);
VCVTTPS2DQ __m128i _mm_mask_cvttps_epi32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm_maskz_cvttps_epi32( __mmask8 k, __m128 a);
VCVTTPS2DQ __m256i _mm256_cvttps_epi32 ( __m256 a)
VCVTTPS2DQ __m128i _mm_cvttps_epi32 ( __m128 a)

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
CVTTPS2PI—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Dword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2C /r</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert two single precision floating-point values from xmm/m64 to two signed doubleword signed integers in mm using truncation.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts two packed single precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

\[
\text{DEST}[31:0] := \text{Convert Single Precision Floating Point To Integer Truncate}(\text{SRC}[31:0]); \\
\text{DEST}[63:32] := \text{Convert Single Precision Floating Point To Integer Truncate}(\text{SRC}[63:32]);
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

CVTTPS2PI __m64 __mm_cvttps_pi32(__m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

See Table 23-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
CVTTSD2SI—Convert With Truncation Scalar Double Precision Floating-Point Value to Signed Integer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 OF 2C /r CVTTSD2SI r32, xmm1/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>F2 REX.W OF 2C /r CVTTSD2SI r64, xmm1/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W0 2C /r VCVTTS2DI r32, xmm1/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 2C /r VCVTTS2DI r64, xmm1/m64</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W0 2C /r VCVTTS2DI r32, xmm1/m64{sa}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 2C /r VCVTTS2DI r64, xmm1/m64{sa}</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.</td>
</tr>
</tbody>
</table>

NOTES:
1. Software should ensure VCVTTS2DI is encoded with VEX.L=0. Encoding VCVTTS2DI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts a double precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.
Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.
Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

**(V)CVTTS2D2SI (All Versions)**

IF 64-Bit Mode and OperandSize = 64
THEN
   DEST[63:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
ELSE
   DEST[31:0] := Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTSD2SI int _mm_cvttsd_i32( __m128d a);
VCVTTSD2SI int _mm_cvtt_roundsd_i32( __m128d a, int sae);
VCVTTSD2SI __int64 _mm_cvttsd_i64( __m128d a);
VCVTTSD2SI __int64 _mm_cvtt_roundsd_i64( __m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32( __m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64( __m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:

#UD IF VEX.vvvv != 1111B.

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
CVTTSS2SI—Convert With Truncation Scalar Single Precision Floating-Point Value to Integer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 2C /r CVTTSS2SI r32, xmm1/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W0 2C /r CVTTSS2SI r32, xmm1/m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1 2C /r CVTTSS2SI r64, xmm1/m32</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W0 2C /r CVTTSS2SI r32, xmm1/m32{sae}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W1 2C /r CVTTSS2SI r64, xmm1/m32{sae}</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.</td>
</tr>
</tbody>
</table>

NOTES:
1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

---

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Fixed</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts a single precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

(V)CVTTSS2SI (All Versions)
IF 64-Bit Mode and OperandSize = 64
THEN
  DEST[63:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
ELSE
  DEST[31:0] := Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
FI;

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTSS2SI int _mm_cvttss_i32( __m128 a);
VCVTTSS2SI int _mm_cvtt_roundss_i32( __m128 a, int sae);
VCVTTSS2SI __int64 _mm_cvttss_i64( __m128 a);
VCVTTSS2SI __int64 _mm_cvtt_roundss_i64( __m128 a, int sae);
CVTTS2SI int _mm_cvttss_si32( __m128 a);
CVTTS2SI __int64 _mm_cvttss_si64( __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
See Table 2-20, ”Type 3 Class Exception Conditions,” additionally:
#UD  If VEX.vvv != 1111B.
EVEX-encoded instructions, see Table 2-48, ”Type E3NF Class Exception Conditions.”
CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Description
Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

Operation
IF OperandSize = 16 (* CWD instruction *)
    THEN
        DX := SignExtend(AX);
    ELSE IF OperandSize = 32 (* CDQ instruction *)
        EDX := SignExtend(EAX); Fi;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
        RDX := SignExtend(RAX); Fi;
    Fi;

Flags Affected
None.

Exceptions (All Operating Modes)
#UD If the LOCK prefix is used.
DAA—Decimal Adjust AL After Addition

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>DAA</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Decimal adjust AL after addition.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode
THEN
#UD;
ELSE
old_AL := AL;
old_CF := CF;
CF := 0;
IF (((AL AND 0FH) > 9) or AF = 1)
THEN
AL := AL + 6;
CF := old_CF or (Carry from AL := AL + 6);
AF := 1;
ELSE
AF := 0;
FI;
IF ((old_AL > 99H) or (old_CF = 1))
THEN
AL := AL + 60H;
CF := 1;
ELSE
CF := 0;
FI;
FI;

Example

ADD AL, BL  Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX
            After: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
DAA          Before: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
            After: AL=14H BL=35H EFLAGS(OSZAPC)=X00111
DAA          Before: AL=2EH BL=35H EFLAGS(OSZAPC)=110000
            After: AL=34H BL=35H EFLAGS(OSZAPC)=X00101
Flags Affected
The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
### DAS—Decimal Adjust AL After Subtraction

#### Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2F</td>
<td>DAS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Decimal adjust AL after subtraction.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

#### Operation

IF 64-Bit Mode
   THEN
      #UD;
   ELSE
      old_AL := AL;
      old_CF := CF;
      CF := 0;
      IF (((AL AND 0FH) > 9) or AF = 1)
         THEN
            AL := AL - 6;
            CF := old_CF or (Borrow from AL := AL - 6);
            AF := 1;
         ELSE
            AF := 0;
         FI;
      FI;
      IF ((old_AL > 99H) or (old_CF = 1))
         THEN
            AL := AL - 60H;
            CF := 1;
         FI;
   FI;

#### Example

**SUB AL, BL**
Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
After: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111

**DAA**
Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111

#### Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.
Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
DEC—Decrement by 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE /1</td>
<td>DEC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m8 by 1.</td>
</tr>
<tr>
<td>REX + FE /1</td>
<td>DEC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Decrement r/m8 by 1.</td>
</tr>
<tr>
<td>FF /1</td>
<td>DEC r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m16 by 1.</td>
</tr>
<tr>
<td>FF /1</td>
<td>DEC r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m32 by 1.</td>
</tr>
<tr>
<td>REX.W + FF /1</td>
<td>DEC r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Decrement r/m64 by 1.</td>
</tr>
<tr>
<td>48+rw</td>
<td>DEC r16</td>
<td>O</td>
<td>N.E.</td>
<td>Valid</td>
<td>Decrement r16 by 1.</td>
</tr>
<tr>
<td>48+rd</td>
<td>DEC r32</td>
<td>O</td>
<td>N.E.</td>
<td>Valid</td>
<td>Decrement r32 by 1.</td>
</tr>
</tbody>
</table>

NOTES:
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction’s 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation
DEST := DEST - 1;

Flags Affected
The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

* #GP(0) If the destination operand is located in a non-writable segment.
* #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
* #SS(0) If the DS, ES, FS, or GS register contains a NULL segment selector.
* #PF(fault-code) If a memory operand effective address is outside the SS segment limit.
* #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
* #UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

* #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
DIV—Unsigned Divide

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /6</td>
<td>DIV r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned divide AX by r/m8, with result stored in AL := Quotient, AH := Remainder.</td>
</tr>
<tr>
<td>REX + F6 /6</td>
<td>DIV r/m8^1</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned divide AX by r/m8, with result stored in AL := Quotient, AH := Remainder.</td>
</tr>
<tr>
<td>F7 /6</td>
<td>DIV r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.</td>
</tr>
<tr>
<td>F7 /6</td>
<td>DIV r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.</td>
</tr>
<tr>
<td>REX.W + F7 /6</td>
<td>DIV r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-15.

**Table 3-15. DIV Action**

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
<th>Remainder</th>
<th>Maximum Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word/byte</td>
<td>AX</td>
<td>r/m8</td>
<td>AL</td>
<td>AH</td>
<td>255</td>
</tr>
<tr>
<td>Doubleword/word</td>
<td>DX:AX</td>
<td>r/m16</td>
<td>AX</td>
<td>DX</td>
<td>65,535</td>
</tr>
<tr>
<td>Quadword/doubleword</td>
<td>EDX:EAX</td>
<td>r/m32</td>
<td>EAX</td>
<td>EDX</td>
<td>$2^{32} - 1$</td>
</tr>
<tr>
<td>Doublequadword/quadword</td>
<td>RDX:RAX</td>
<td>r/m64</td>
<td>RAX</td>
<td>RDX</td>
<td>$2^{64} - 1$</td>
</tr>
</tbody>
</table>
Operation

IF SRC = 0
THEN #DE; Fi; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
THEN
  temp := AX / SRC;
  IF temp > FFH
  THEN #DE; (* Divide error *)
  ELSE
    AL := temp;
    AH := AX MOD SRC;
  FI;
ELSE IF OperandSize = 16 (* Doubleword/word operation *)
THEN
  temp := DX:AX / SRC;
  IF temp > FFFFH
  THEN #DE; (* Divide error *)
  ELSE
    AX := temp;
    DX := DX:AX MOD SRC;
  FI;
ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
THEN
  temp := EDX:EAX / SRC;
  IF temp > FFFFFFFFH
  THEN #DE; (* Divide error *)
  ELSE
    EAX := temp;
    EDX := EDX:EAX MOD SRC;
  FI;
ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
THEN
  temp := RDX:RAX / SRC;
  IF temp > FFFFFFFFFFFFFFFFH
  THEN #DE; (* Divide error *)
  ELSE
    RAX := temp;
    RDX := RDX:RAX MOD SRC;
  FI;
FI;
FI;

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.
Protected Mode Exceptions
#DE If the source operand (divisor) is 0
If the quotient is too large for the designated register.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#DE If the source operand (divisor) is 0.
If the quotient is too large for the designated register.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#DE If the source operand (divisor) is 0.
If the quotient is too large for the designated register.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#DE If the source operand (divisor) is 0
If the quotient is too large for the designated register.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**DIVPD—Divide Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5E /r DIVPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Divide packed double precision floating-point values in xmm1 by packed double precision floating-point values in xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128:66.66.0F:W1 5E /r VDIVPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256:66.66.0F:W1 5E /r VDIVPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128:66.66.0F:W1 5E /r VDIVPD xmm1[k1]{k1}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed double precision floating-point values in xmm2 by packed double precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256:66.66.0F:W1 5E /r VDIVPD ymm1[k1]{k1}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed double precision floating-point values in ymm2 by packed double precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512:66.66.0F:W1 5E /r VDIVPD zmm1[k1]{k1}, zmm2, zmm3/m512/m64bcst(er)</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Divide packed double precision floating-point values in zmm2 by packed double precision floating-point values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (r)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (r)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD divide of the double precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

**EVEX encoded versions:** The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL:1:256) of the corresponding destination are zeroed.

**VEX.128 encoded version:** The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL:1:128) of the corresponding destination are zeroed.

**128-bit Legacy SSE version:** The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL:1:128) of the corresponding destination are unmodified.
Operation

**VDIVPD (EVEX Encoded Versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  ; refer to Table 15-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN


ELSE


FI;

ELSE

IF *merging-masking*  ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE  ; zeroing-masking

DEST[i+63:i] := 0

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VDIVPD (VEX.256 Encoded Version)**

DEST[63:0] := SRC1[63:0] / SRC2[63:0]


DEST[MAXVL-1:256] := 0;

**VDIVPD (VEX.128 Encoded Version)**

DEST[63:0] := SRC1[63:0] / SRC2[63:0]


DEST[MAXVL-1:128] := 0;

**DIVPD (128-bit Legacy SSE Version)**

DEST[63:0] := SRC1[63:0] / SRC2[63:0]


DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m512d __mm512_div_pd(__m512d a, __m512d b);
VDIVPD __m512d __mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d __mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d __mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d __mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d __mm128_div_pd(__m128d a, __m128d b);
VDIVPD __m128d __mm128_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d __mm128_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d __mm512_div_round_pd(__m512d a, __m512d b, int);
VDIVPD __m512d __mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d __mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d __mm256_div_pd(__m256d a, __m256d b);
DIVPD __m128d __mm_div_pd(__m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
## DIVPS—Divide Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5E /r DIVPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Divide packed single precision floating-point values in xmm1 by packed single precision floating-point values in xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 5E /r VDIVPS xmm1 {k1}[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed single precision floating-point values in xmm2 by packed single precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 5E /r VDIVPS ymm1 {k1}[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Divide packed single precision floating-point values in ymm2 by packed single precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 5E /r VDIVPS zmm1 {k1}[z], zmm2, zmm3/m512/m32bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Divide packed single precision floating-point values in zmm2 by packed single precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD divide of the four, eight or sixteen packed single precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

**EVEX encoded versions:** The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

VDIVPS (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+31:i] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC1[i+31:i] / SRC2[i+31:i]
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] := 0
                FI
        FI;
    ENDFOR

DEST[MAXVL-1:VL] := 0

VDIVPS (VEX.256 Encoded Version)

DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[MAXVL-1:256] := 0;

VDIVPS (VEX.128 Encoded Version)

DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[MAXVL-1:128] := 0;
DIVPS (128-bit Legacy SSE Version)

DEST[31:0] := SRC1[31:0] / SRC2[31:0]
DEST[95:64] := SRC1[95:64] / SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPS __m512 _mm512_div_ps( __m512 a, __m512 b);
VDIVPS __m512 __m512_mask_div_ps( __m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 __m512_maskz_div_ps( __mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d __mm256_mask_div_pd( __m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d __mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d __mm128_mask_div_pd( __m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d __mm128_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 __m512_div_round_ps( __m512 s, __m512 a, __m512 b, int);
VDIVPS __m512 __m512_mask_div_round_ps( __m512 s, __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 __m512_maskz_div_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 __m256_div_ps( __m256 s, __m256 a, __m256 b);
DIVPS __m256 __m256_div_ps( __m256 s, __m256 a, __m256 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
**DIVSD—Divide Scalar Double Precision Floating-Point Value**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5E /r DIVSD xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Divide low double precision floating-point value in xmm1 by low double precision floating-point value in xmm2/m64.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 5E /r VDIVSD xmm1, xmm2, xmm3/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}[z], xmm2, xmm3/m64[er]</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Divide low double precision floating-point value in xmm2 by low double precision floating-point value in xmm3/m64.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM[reg (r, w)]</td>
<td>ModRM[reg/m (r)]</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM[reg (w)]</td>
<td>VEX.vvvv (r)</td>
<td>ModRM[reg/m (r)]</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM[reg (w)]</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM[reg/m (r)]</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Divides the low double precision floating-point value in the first source operand by the low double precision floating-point value in the second source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

VDIVSD (EVEX Encoded Version)
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] := SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VDIVSD (VEX.128 Encoded Version)
DEST[63:0] := SRC1[63:0] / SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

DIVSD (128-bit Legacy SSE Version)
DEST[63:0] := DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd( __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd( __m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd( __mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd ( __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
DIVSS—Divide Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5E /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Divide low single precision floating-point value in xmm1 by low single precision floating-point value in xmm2/m32.</td>
</tr>
<tr>
<td>VEX.LI.GF.30.F.W5E 5E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.</td>
</tr>
<tr>
<td>EVEX.LI.GF.30.F.W0 5E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Divide low single precision floating-point value in xmm2 by low single precision floating-point value in xmm3/m32.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Divides the low single precision floating-point value in the first source operand by the low single precision floating-point value in the second source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.
Operation

**VDIVSS (EVEX Encoded Version)**

If (EVEX.b = 1) AND SRC2 *is a register*

THEN

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

Fl;

IF k1[0] or *no writemask*

THEN

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
```

ELSE

```
IF *merging-masking* ; merging-masking
THEN *DEST[31:0] remains unchanged* ; zeroing-masking
ELSE

```
THEN DEST[31:0] := 0
```

Fl;

Fl;

```
```

```
DEST[MAXVL-1:128] := 0
```

**VDIVSS (VEX.128 Encoded Version)**

```
DEST[31:0] := SRC1[31:0] / SRC2[31:0]
```

```
```

```
DEST[MAXVL-1:128] := 0
```

**DIVSS (128-bit Legacy SSE Version)**

```
DEST[31:0] := DEST[31:0] / SRC[31:0]
```

```
DEST[MAXVL-1:32] (Unmodified)
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss( __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss( __m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss( __mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);
```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions."

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
DPPD—Dot Product of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 41 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPPD xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Selectively multiply packed double precision floating-point values from xmm1 with packed double precision floating-point values from xmm2, add and selectively store the packed double precision floating-point values to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 41 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDPPD xmm1,xmm2, xmm3/m128, imm8</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Selectively multiply packed double precision floating-point values from xmm2 with packed double precision floating-point values from xmm3, add and selectively store the packed double precision floating-point values to xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Conditionally multiplies the packed double precision floating-point values in the destination operand (first operand) with the packed double precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The two resulting double precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPD follows the NaN forwarding rules stated in the Software Developer’s Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.
**Operation**

**DP_primitive (SRC1, SRC2)**

If (imm8[4] = 1)
   THEN Temp1[63:0] := DEST[63:0] * SRC[63:0]; // update SIMD exception flags
   ELSE Temp1[63:0] := +0.0; Fl;

IF (imm8[5] = 1)
   THEN Temp1[127:64] := DEST[127:64] * SRC[127:64]; // update SIMD exception flags
   ELSE Temp1[127:64] := +0.0; Fl;
/* if unmasked exception reported, execute exception handler*/

Temp2[63:0] := Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)
   THEN DEST[63:0] := Temp2[63:0];
   ELSE DEST[63:0] := +0.0; Fl;
IF (imm8[1] = 1)
   THEN DEST[127:64] := Temp2[63:0];
   ELSE DEST[127:64] := +0.0; Fl;

**DPPD (128-bit Legacy SSE Version)**

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)

**VDPPD (VEX.128 Encoded Version)**

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] := 0

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

DPPD __m128d __mm_dp_pd (__m128d a, __m128d b, const int mask);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

**Other Exceptions**

See Table 2-19, "Type 2 Class Exception Conditions," additionally:

#UD If VEX.L= 1.
DPPS—Dot Product of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 40 /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Selectively multiply packed single precision floating-point values from xmm1 with packed single precision floating-point values from xmm2, add and selectively store the packed single precision floating-point values or zero values to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 40 /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single precision floating-point values from xmm1 with packed single precision floating-point values from xmm2/mem selectively add and store to xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.WIG 40 /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single precision floating-point values from ymm2 with packed single precision floating-point values from ymm3/mem, selectively add pairs of elements and store to ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

The four resulting single precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is “1”, the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPS follows the NaN forwarding rules stated in the Software Developer’s Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**Operation**

**DP_primitive (SRC1, SRC2)**

If (imm8[4] = 1)

THEN Temp1[31:0] := DEST[31:0] * SRC[31:0]; // update SIMD exception flags

ELSE Temp1[31:0] := +0.0; Fl;

IF (imm8[5] = 1)


ELSE Temp1[63:32] := +0.0; Fl;

IF (imm8[6] = 1)

THEN Temp1[95:64] := DEST[95:64] * SRC[95:64]; // update SIMD exception flags

ELSE Temp1[95:64] := +0.0; Fl;

IF (imm8[7] = 1)


ELSE Temp1[127:96] := +0.0; Fl;

Temp2[31:0] := Temp1[31:0] + Temp1[63:32]; // update SIMD exception flags

/* if unmasked exception reported, execute exception handler*/

Temp3[31:0] := Temp1[95:64] + Temp1[127:96]; // update SIMD exception flags

/* if unmasked exception reported, execute exception handler*/

Temp4[31:0] := Temp2[31:0] + Temp3[31:0]; // update SIMD exception flags

/* if unmasked exception reported, execute exception handler*/

IF (imm8[0] = 1)

THEN DEST[31:0] := Temp4[31:0];

ELSE DEST[31:0] := +0.0; Fl;

IF (imm8[1] = 1)

THEN DEST[63:32] := Temp4[31:0];

ELSE DEST[63:32] := +0.0; Fl;

IF (imm8[2] = 1)

THEN DEST[95:64] := Temp4[31:0];

ELSE DEST[95:64] := +0.0; Fl;

IF (imm8[3] = 1)

THEN DEST[127:96] := Temp4[31:0];

ELSE DEST[127:96] := +0.0; Fl;

**DPPS (128-bit Legacy SSE Version)**

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);

DEST[MAXVL-1:128] (Unmodified)

**VDPPS (VEX.128 Encoded Version)**

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);

DEST[MAXVL-1:128] := 0

**VDPPS (VEX.256 Encoded Version)**

DEST[127:0] := DP_Primitive(SRC1[127:0], SRC2[127:0]);


**Flags Affected**

None.
**Intel C/C++ Compiler Intrinsic Equivalent**

(V)DPSS __m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);
VDPPS __m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

**Other Exceptions**

See Table 2-19, “Type 2 Class Exception Conditions.”
EMMS—Empty MMX Technology State

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 77</td>
<td>EMMS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set the x87 FPU tag word to empty.</td>
</tr>
</tbody>
</table>

**Description**

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
x87FPUTagWord := FFFFH;
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

void _mm_empty()

**Flags Affected**

None

**Protected Mode Exceptions**

- #UD If CR0.EM[bit 2] = 1.
- #NM If CR0.TS[bit 3] = 1.
- #MF If there is a pending FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
ENCODEKEY128—Encode 128-Bit Key With Key Locker

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operands 4—5</th>
<th>Operands 6—7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Implicit XMM0 (r, w)</td>
<td>Implicit XMM1—2 (w)</td>
<td>Implicit XMM4—6 (w)</td>
</tr>
</tbody>
</table>

**Description**

The ENCODEKEY128\(^1\) instruction wraps a 128-bit AES key from the implicit operand XMM0 into a key handle that is then stored in the implicit destination operands XMM0—2. The explicit source operand specifies handle restrictions, if any. The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

**Operation**

ENCODEKEY128

\#GP (0) if a reserved bit\(^2\) in SRC[31:0] is set

\[\text{InputKey}[127:0] := \text{XMM0};\]
\[\text{KeyMetadata}[2:0] = \text{SRC}[2:0];\]
\[\text{KeyMetadata}[23:3] = 0; \quad \text{// Reserved for future usage}\]
\[\text{KeyMetadata}[27:24] = 0; \quad \text{// KeyType is AES-128 (value of 0)}\]
\[\text{KeyMetadata}[127:28] = 0; \quad \text{// Reserved for future usage}\]

\[\text{Handle}[383:0] := \text{WrapKey128(InputKey}[127:0], \text{KeyMetadata}[127:0], \text{IwKey}\_\text{Integrity Key}[127:0], \text{IwKey}\_\text{Encryption Key}[255:0]);\]

\[\text{DEST}[0] := \text{IwKey}\_\text{NoBackup};\]
\[\text{DEST}[4:1] := \text{IwKey}\_\text{KeySource}[3:0];\]
\[\text{DEST}[31:5] = 0;\]
\[\text{XMM0} := \text{Handle}[127:0]; \quad \text{// AAD}\]
\[\text{XMM1} := \text{Handle}[255:128]; \quad \text{// Integrity Tag}\]
\[\text{XMM2} := \text{Handle}[383:256]; \quad \text{// CipherText}\]
\[\text{XMM4} := 0; \quad \text{// Reserved for future usage}\]
\[\text{XMM5} := 0; \quad \text{// Reserved for future usage}\]
\[\text{XMM6} := 0; \quad \text{// Reserved for future usage}\]
\[\text{RFLAGS.OF, SF, ZF, AF, PF, CF} := 0;\]

**Flags Affected**

All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

---

1. Further details on Key Locker and usage of this instruction can be found here:
2. SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.
Intel C/++ Compiler Intrinsic Equivalent
ENCODEKEY128  unsigned int _mm_encodekey128_u32(unsigned int htype, __m128i key, void* h);

Exceptions (All Operating Modes)
#GP     If reserved bit is set in source register value.
#UD     If the LOCK prefix is used.
          If CPUID.07H:ECX.KL[bit 23] = 0.
          If CR4.KL = 0.
          If CPUID.19H:EBX.AESKLE[bit 0] = 0.
          If CR0.EM = 1.
          If CR4.OSFXSR = 0.
#NM     If CR0.TS = 1.
**ENCODEKEY256—Encode 256-Bit Key With Key Locker**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 FB 11:rrr:bbb ENCODEKEY256 r32, r32 &lt;XMM0-6&gt;</td>
<td>A</td>
<td>V/V</td>
<td>AESKLE</td>
<td>Wrap a 256-bit AES key from XMM1:XMM0 into a key handle and store it in XMM0—3.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operands 3—4</th>
<th>Operands 5—9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Implicit XMM0—1 (r, w)</td>
<td>Implicit XMM2—6 (w)</td>
</tr>
</tbody>
</table>

### Description

The ENCODEKEY256\(^1\) instruction wraps a 256-bit AES key from the implicit operand XMM1:XMM0 into a key handle that is then stored in the implicit destination operands XMM0-3.

The explicit source operand is a general-purpose register and specifies what handle restrictions should be built into the handle.

The explicit destination operand is populated with information on the source of the key and its attributes. XMM4 through XMM6 are reserved for future usages and software should not rely upon them being zeroed.

### Operation

**ENCODEKEY256**

\#GP (0) if a reserved bit\(^2\) in SRC[31:0] is set

InputKey[255:0] := XMM1:XMM0;

KeyMetadata[2:0] := SRC[2:0];

KeyMetadata[23:3] := 0; // Reserved for future usage

KeyMetadata[27:24] := 1; // KeyType is AES-256 (value of 1)

KeyMetadata[127:28] := 0; // Reserved for future usage

Handle[511:0] := WrapKey256(InputKey[255:0], KeyMetadata[127:0], IWKey.Integrity Key[127:0], IWKey.Encryption Key[255:0]);

DEST[0] := IWKey.NoBackup;

DEST[4:1] := IWKey.KeySource[3:0];

DEST[31:5] := 0;

XMM0 := Handle[127:0]; // AAD

XMM1 := Handle[255:128]; // Integrity Tag

XMM2 := Handle[383:256]; // CipherText[127:0]

XMM3 := Handle[511:384]; // CipherText[255:128]

XMM4 := 0; // Reserved for future usage

XMM5 := 0; // Reserved for future usage

XMM6 := 0; // Reserved for future usage

RFLAGS.OF, SF, ZF, AF, PF, CF := 0;

---

1. Further details on Key Locker and usage of this instruction can be found here:

2. SRC[31:3] are currently reserved for future usages. SRC[2], which indicates a no-decrypt restriction, is reserved if CPUID.19H:EAX[2] is 0. SRC[1], which indicates a no-encrypt restriction, is reserved if CPUID.19H:EAX[1] is 0. SRC[0], which indicates a CPL=0-only restriction, is reserved if CPUID.19H:EAX[0] is 0.
Flags Affected
All arithmetic flags (OF, SF, ZF, AF, PF, CF) are cleared to 0. Although they are cleared for the currently defined operations, future extensions may report information in the flags.

Intel C/C++ Compiler Intrinsic Equivalent
ENCODEKEY256 unsigned int _mm_encodekey256_u32(unsigned int htype, ___m128i key_lo, ___m128i key_hi, void* h);

Exceptions (All Operating Modes)
#GP If reserved bit is set in source register value.
#UD If the LOCK prefix is used.
If CPUID.07H:ECX.KL[bit 23] = 0.
If CR4.KL = 0.
If CPUID.19H:EBX.AESKLE[bit 0] = 0.
If CR0.EM = 1.
If CR4.OSFXSR = 0.
#NM If CR0.TS = 1.
ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 1E FB ENDBR32</td>
<td>ZO</td>
<td>V/V</td>
<td>CET_IBT</td>
<td>Terminate indirect branch in 32-bit and compatibility mode.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Terminate an indirect branch in 32 bit and compatibility mode.

### Operation

IF EndbranchEnabled(CPL) & (IA32_EFER.LMA = 0 | (IA32_EFER.LMA=1 & CS.L = 0)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = IDLE
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = IDLE
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;

### Flags Affected

None.

### Exceptions

None.
ENDBR64—Terminate an Indirect Branch in 64-bit Mode

**Description**
Terminate an indirect branch in 64 bit mode.

**Operation**
IF EndbranchEnabled(CPL) & IA32_EFER.LMA = 1 & CS.L = 1
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = IDLE
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = IDLE
      IA32_S_CET.SUPPRESS = 0
  FI;
FI;

**Flags Affected**
None.

**Exceptions**
None.

---

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 1E FA ENDBR64</td>
<td>ZO</td>
<td>V/V</td>
<td>CET_IBT</td>
<td>Terminate indirect branch in 64-bit mode.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
**ENQCMD—Enqueue Command**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 38 F8 (11):rrr:bbb</td>
<td>A</td>
<td>V/V</td>
<td>ENQCMD</td>
<td>Atomically enqueue 64-byte user command from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.</td>
</tr>
<tr>
<td>ENQCMD r32/r64, m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The ENQCMD instruction allows software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the following format:

```plaintext
  511
  32 31 30 20 19 0
```

- **DEVICE SPECIFIC COMMAND**
- **PRIV**
- **RESERVED**
- **PASID**

**Figure 3-16. 64-Byte Data Written to Enqueue Registers**

Bits 19:0 convey the process address space identifier (PASID), a value which system software may assign to individual software threads. Bit 31 contains privilege identification (0 = user; 1 = supervisor). Devices implementing enqueue registers may use these two values along with a device-specific command in the upper 60 bytes.

The ENQCMD instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically. Bits 31:0 of the source operand must be zero.

The instruction then formats those 64 bytes into **command data** with a format consistent with that given in Figure 3-16:

- Command[19:0] get IA32_PASID[19:0].
- Command[30:20] are zero.
- Command[31] is 0 (indicating user; this value is used regardless of CPL).
- Command[511:32] get bits 511:32 of the source operand that was read from memory.

The ENQCMD instruction uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden). The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

---

1. It is expected that system software will load the IA32_PASID MSR so that bits 19:0 contain the PASID of the current software thread. The MSR’s valid bit, IA32_PASID[31], must be 1. For additional details on the IA32_PASID MSR, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

2. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).
An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store’s command data is never combined with that of any other store to the same address. Unlike other stores, an enqueue store returns a status, which the ENQCMD instruction loads into the ZF flag in the RFLAGS register:

- **ZF** = 0 (success) reports that the 64-byte command data was written atomically to a device’s enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- **ZF** = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons. This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

Availability of the ENQCMD instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.(EAX=07H, ECX=0H):ECX[bit 29]).

**Operation**

```plaintext
IF IA32_PASID[31] = 0
    THEN #GP;
ELSE
    COMMAND := (SRC & ~FFFFFFFFH) | (IA32_PASID & FFFFFH);
    DEST := COMMAND;
FI;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

ENQCMD int_enqcmd(void *dst, const void *src)

**Flags Affected**

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

- **#GP(0)** For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
  - If destination linear address is not aligned to a 64-byte boundary.
  - If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
  - If bits 31:0 of the source operand are not all zero.
- **#SS(0)** For an illegal address in the SS segment.
- **#PF(fault-code)** For a page fault.
- **#UD** If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
  - If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
If destination linear address is not aligned to a 64-byte boundary.
If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
If bits 31:0 of the source operand are not all zero.

#UD If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real-address mode. Additionally:

#PF(fault-code) For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in non-canonical form.

#GP(0) If the memory address is in non-canonical form.
If destination linear address is not aligned to a 64-byte boundary.
If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
If bits 31:0 of the source operand are not all zero.

#PF(fault-code) For a page fault.

#UD If CPUID.07H.0H:ECX.ENQCMD[bit 29].
If the LOCK prefix is used.
ENQCMDS—Enqueue Command Supervisor

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 3B 89 l(11):rrr:bbb</td>
<td>A</td>
<td>V/V</td>
<td>ENQCMD</td>
<td>Atomically enqueue 64-byte command with PASID from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.</td>
</tr>
<tr>
<td>ENQCMDS r32/r64, m512</td>
<td>A</td>
<td>V/V</td>
<td>ENQCMD</td>
<td>Atomically enqueue 64-byte command with PASID from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The ENQCMDS instruction allows system software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the format given in Figure 3-16 and explained in the section on “ENQCMD—Enqueue Command.”

The ENQCMDS instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically. Bits 30:20 of the source operand must be zero.

ENQCMDS formats its source data differently from ENQCMD. Specifically, it formats them into **command data** as follows:

- **Command[19:0]** get bits 19:0 of the source operand that was read from memory. These 20 bits communicate a process address-space identifier (PASID).
- **Command[30:20]** are zero.
- **Command[511:31]** get bits 511:31 of the source operand that was read from memory. Bit 31 communicates a privilege identification (0 = user; 1 = supervisor).

The ENQCMDS instruction then uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).\(^1\) The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store’s command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMDS instruction loads into the ZF flag in the RFLAGS register:

- **ZF = 0** (success) reports that the 64-byte command data was written atomically to a device’s enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- **ZF = 1** (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons.

---

1. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).
This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

The ENQCMDS instruction may be executed only if CPL = 0. Availability of the ENQCMDS instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.(EAX=07H, ECX=0H):ECX[bit 29]).

Operation

\[
\text{DEST} := \text{SRC};
\]

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMDS int_enqcmds(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

- #GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
  If destination linear address is not aligned to a 64-byte boundary.
  If the current privilege level is not 0.
  If bits 30:20 of the source operand are not all zero.
- #SS(0) For an illegal address in the SS segment.
- #PF(fault-code) For a page fault.
- #UD If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
  If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
  If destination linear address is not aligned to a 64-byte boundary.
  If bits 30:20 of the source operand are not all zero.
- #UD If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
  If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) The ENQCMDS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in non-canonical form.
- #GP(0) If the memory address is in non-canonical form.
  If destination linear address is not aligned to a 64-byte boundary.
  If the current privilege level is not 0.
  If bits 30:20 of the source operand are not all zero.
- #PF(fault-code) For a page fault.
#UD If CPUID.07H.0H:ECX.ENQCMD[bit 29]. If the LOCK prefix is used.
ENTER—Make Stack Frame for Procedure Parameters

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB iw 00</td>
<td>ENTER imm16, 0</td>
<td>II</td>
<td>Valid</td>
<td>Valid</td>
<td>Create a stack frame for a procedure.</td>
</tr>
<tr>
<td>CB iw 01</td>
<td>ENTER imm16,1</td>
<td>II</td>
<td>Valid</td>
<td>Valid</td>
<td>Create a stack frame with a nested pointer for a procedure.</td>
</tr>
<tr>
<td>CB iw ib</td>
<td>ENTER imm16, imm8</td>
<td>II</td>
<td>Valid</td>
<td>Valid</td>
<td>Create a stack frame with nested pointers for a procedure.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>iw</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Creates a stack frame (comprising of space for dynamic storage and 1-32 frame pointer storage) for a procedure. The first operand (imm16) specifies the size of the dynamic storage in the stack frame (that is, the number of bytes of dynamically allocated on the stack for the procedure). The second operand (imm8) gives the lexical nesting level (0 to 31) of the procedure. The nesting level (imm8 mod 32) and the OperandSize attribute determine the size in bytes of the storage space for frame pointers.

The nesting level determines the number of frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. The default size of the frame pointer is the StackAddrSize attribute, but can be overridden using the 66H prefix. Thus, the OperandSize attribute determines the size of each frame pointer that will be copied into the stack frame and the data being transferred from SP/ESP/RSP register into the BP/EBP/RBP register.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded. Use of 66H prefix changes frame pointer operand size to 16 bits.

When the 66H prefix is used and causing the OperandSize attribute to be less than the StackAddrSize, software is responsible for the following:

- The companion LEAVE instruction must also use the 66H prefix,
- The value in the RBP/EBP register prior to executing “66H ENTER” must be within the same 16KByte region of the current stack pointer (RSP/ESP), such that the value of RBP/EBP after “66H ENTER” remains a valid address in the stack. This ensures “66H LEAVE” can restore 16-bits of data from the stack.
Operation
AllocSize := imm16;
NestingLevel := imm8 MOD 32;
IF (OperandSize = 64)
  THEN
    Push(RBP); (* RSP decrements by 8 *)
    FrameTemp := RSP;
  ELSE IF OperandSize = 32
    THEN
      Push(EBP); (* (E)SP decrements by 4 *)
      FrameTemp := ESP; Fi;
    ELSE (* OperandSize = 16 *)
      Push(BP); (* RSP or (E)SP decrements by 2 *)
      FrameTemp := SP;
  FI;
IF NestingLevel = 0
  THEN GOTO CONTINUE;
FI;
IF (NestingLevel > 1)
  THEN FOR i := 1 to (NestingLevel - 1)
      DO
        IF (OperandSize = 64)
          THEN
            RBP := RBP - 8;
            Push([RBP]); (* Quadword push *)
          ELSE IF OperandSize = 32
            THEN
              IF StackSize = 32
                THEN
                  EBP := EBP - 4;
                  Push([EBP]); (* Doubleword push *)
                ELSE (* StackSize = 16 *)
                  BP := BP - 4;
                  Push([BP]); (* Doubleword push *)
                FI;
            ELSE (* OperandSize = 16 *)
              IF StackSize = 64
                THEN
                  RBP := RBP - 2;
                  Push([RBP]); (* Word push *)
              ELSE IF StackSize = 32
                THEN
                  EBP := EBP - 2;
                  Push([EBP]); (* Word push *)
                ELSE (* StackSize = 16 *)
                  BP := BP - 2;
                  Push([BP]); (* Word push *)
              FI;
            FI;
          ELSE (* OperandSize = 16 *)
            IF StackSize = 64
              THEN
                RBP := RBP - 2;
                Push([RBP]); (* Word push *)
            ELSE IF StackSize = 32
              THEN
                EBP := EBP - 2;
                Push([EBP]); (* Word push *)
              ELSE (* StackSize = 16 *)
                BP := BP - 2;
                Push([BP]); (* Word push *)
            FI;
          FI;
      OD;
  FI;
IF (OperandSize = 64) (* nestinglevel 1 *)
THEN
    Push(FrameTemp); (* Quadword push and RSP decrements by 8 *)
ELSE IF OperandSize = 32
    THEN
        Push(FrameTemp);FI; (* Doubleword push and (E)SP decrements by 4 *)
ELSE (* OperandSize = 16 *)
    Push(FrameTemp); (* Word push and RSP|ESP|SP decrements by 2 *)
FI;
CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP := FrameTemp;
        RSP := RSP − AllocSize;
ELSE IF OperandSize = 32
    THEN
        EBP := FrameTemp;
        ESP := ESP − AllocSize; FI;
ELSE (* OperandSize = 16 *)
    BP := FrameTemp[15:1]; (* Bits 16 and above of applicable RBP/EBP are unmodified *)
    SP := SP − AllocSize;
FI;
END;

Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current
    stack segment) would cause a page fault.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#SS If the new value of the SP or ESP register is outside the stack segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#SS(0) If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current
    stack segment) would cause a page fault.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs or if a write using the final value of the stack pointer (within the current
    stack segment) would cause a page fault.
#UD If the LOCK prefix is used.
EXTRACTPS—Extract Packed Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 17 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTRACTPS reg/m32, xmm1, imm8</td>
<td>A</td>
<td>VV</td>
<td>SSE4_1</td>
<td>Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEXTRACTPS reg/m32, xmm1, imm8</td>
<td>A</td>
<td>VV</td>
<td>AVX</td>
<td>Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEXTRACTPS reg/m32, xmm1, imm8</td>
<td>B</td>
<td>VV</td>
<td>AVX512F</td>
<td>Extract one single precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Extracts a single precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single precision floating-point value is stored in the low 32-bits of the destination operand.

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.vvvv/EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VEXTRACTPS (EVEX and VEX.128 Encoded Version)

```
SRC_OFFSET := IM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] := 0
ELSE
    DEST[31:0] := (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```
**EXTRACTPS (128-bit Legacy SSE Version)**

\[ \text{SRC\_OFFSET} := \text{IMM8}[1:0] \]

IF (64-Bit Mode and DEST is register)
   DEST[31:0] := (SRC[127:0] >> (SRC\_OFFSET*32)) AND 0FFFFFFFFh
   DEST[63:32] := 0
ELSE
   DEST[31:0] := (SRC[127:0] >> (SRC\_OFFSET*32)) AND 0FFFFFFFFh
FI

**Intel C/C++ Compiler Intrinsic Equivalent**

\[ \text{EXTRACTPS int } \_\text{mm}\_\text{extract}\_\text{ps}(\_\text{m}128\ a, \text{const int nidx}); \]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

VEX-encoded instructions, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-57, “Type E9NF Class Exception Conditions.”
Additionally:

- \#UD IF VEX.L = 0.
- \#UD IF VEX.vvvv !\(=\) 1111B or EVEX.vvv !\(=\) 1111B.
F2XM1—Compute $2^x-1$

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F0</td>
<td>F2XM1</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with $(2^{ST(0)} - 1)$.</td>
</tr>
</tbody>
</table>

**Description**

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range $-1.0$ to $+1.0$. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-1.0$ to $-0$</td>
<td>$-0.5$ to $0$</td>
</tr>
<tr>
<td>$-0$</td>
<td>$-0$</td>
</tr>
<tr>
<td>$+0$</td>
<td>$+0$</td>
</tr>
<tr>
<td>$+0$ to $+1.0$</td>
<td>$+0$ to $1.0$</td>
</tr>
</tbody>
</table>

Values other than 2 can be exponentiated using the following formula:

$$x^y := 2^{y \cdot \log_2 x}$$

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

$$ST(0) := (2^{ST(0)} - 1);$$

**FPU Flags Affected**

- **C1**: Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
- **C0, C2, C3**: Undefined.

**Floating-Point Exceptions**

- **#IS**: Stack underflow occurred.
- **#IA**: Source operand is an SNaN value or unsupported format.
- **#D**: Source is a denormal value.
- **#U**: Result is too small for destination format.
- **#P**: Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD**: If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FABS—Absolute Value

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 E1</td>
<td>FABS</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST with its absolute value.</td>
</tr>
</tbody>
</table>

Description
Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

Table 3-17. Results Obtained from FABS

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>− F</td>
<td>+ F</td>
</tr>
<tr>
<td>− 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ F</td>
<td>+ F</td>
</tr>
<tr>
<td>+ ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:
F Means finite floating-point value.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
ST(0) := |ST(0)|;

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FADD/FADDP/FIADD—Add**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /0</td>
<td>FADD m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m32fp to ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DC /0</td>
<td>FADD m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m64fp to ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>D8 C0+i</td>
<td>FADD ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(0) to ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DC C0+i</td>
<td>FADD ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(i) to ST(0) and store result in ST(i).</td>
</tr>
<tr>
<td>DE C0+i</td>
<td>FADDP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(0) to ST(i), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE C1</td>
<td>FADDP</td>
<td>Valid</td>
<td>Valid</td>
<td>Add ST(0) to ST(1), store result in ST(1), and pop the register stack.</td>
</tr>
<tr>
<td>DA /0</td>
<td>FIADD m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m32int to ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DE /0</td>
<td>FIADD m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Add m16int to ST(0) and store result in ST(0).</td>
</tr>
</tbody>
</table>

**Description**

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward -∞ mode, in which case the result is -0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-18.
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FIADD

    THEN
    DEST := DEST + ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (* Source operand is floating-point value *)

    DEST := DEST + SRC;

FI;

IF Instruction = FADDP

    THEN
    PopRegisterStack;

FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.

C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.

#IA  Operand is an SNaN value or unsupported format.

#D  Operands are infinities of unlike sign.

#O  Source operand is a denormal value.

#U  Result is too small for destination format.

#P  Result is too large for destination format.

Value cannot be represented exactly in destination format.
Protected Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.

Real-Address Mode Exceptions

- **#GP**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#UD**: If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made.
- **#UD**: If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)**: If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)**: If the memory address is in a non-canonical form.
- **#NM**: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- **#MF**: If there is a pending x87 FPU exception.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used.
**FBLD—Load Binary Coded Decimal**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /4</td>
<td>FBLD m80bcd</td>
<td>Valid</td>
<td>Valid</td>
<td>Convert BCD value to floating-point and push onto the FPU stack.</td>
</tr>
</tbody>
</table>

**Description**

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of -0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```
TOP := TOP – 1;
ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);
```

**FPU Flags Affected**

- C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack overflow occurred.

**Protected Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  
  If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FBSTP—Store BCD Integer and Pop

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

Table 3-19. FBSTP Results

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>F ≤ − 1</td>
<td>− D</td>
</tr>
<tr>
<td>− 1 &lt; F &lt; 0</td>
<td>**</td>
</tr>
<tr>
<td>− 0</td>
<td>− 0</td>
</tr>
<tr>
<td>+ 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ 0 &lt; F &lt; +1</td>
<td>**</td>
</tr>
<tr>
<td>F ≥ +1</td>
<td>+ D</td>
</tr>
<tr>
<td>+ ∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>*</td>
</tr>
</tbody>
</table>

NOTES:

F Means finite floating-point value.
D Means packed-BCD number.
* Indicates floating-point invalid-operation (#IA) exception.
** ±0 or ±1, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST := BCD(ST(0));
PopRegisterStack;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.
Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Converted value that exceeds 18 BCD digits in length.
Source operand is an SNaN, QNaN, ±∞, or in an unsupported format.

#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0) If a segment register is being loaded with a segment selector that points to a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0_EM[bit 2] or CR0_TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
FCHS—Change Sign

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 E0</td>
<td>FCHS</td>
<td>Valid</td>
<td>Valid</td>
<td>Complements sign of ST(0).</td>
</tr>
</tbody>
</table>

Description
Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-20. FCHS Results

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>− F</td>
<td>+ F</td>
</tr>
<tr>
<td>− 0</td>
<td>+ 0</td>
</tr>
<tr>
<td>+ 0</td>
<td>− 0</td>
</tr>
<tr>
<td>+ F</td>
<td>− F</td>
</tr>
<tr>
<td>+ ∞</td>
<td>− ∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:
* F means finite floating-point value.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
SignBit(ST(0)) := NOT (SignBit(ST(0)));

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
FCLEX/FNCLEX—Clear Exceptions

**Description**
Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

**IA-32 Architecture Compatibility**
When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCSR register.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**
```
FPUSStatusWord[0:7] := 0;
FPUSStatusWord[15] := 0;
```

**FPU Flags Affected**
The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**
None.

**Protected Mode Exceptions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#NM</td>
<td>CR0.EM[bit 2] or CR0.TS[bit 3] = 1.</td>
<td></td>
</tr>
<tr>
<td>#UD</td>
<td>If the LOCK prefix is used.</td>
<td></td>
</tr>
</tbody>
</table>

**Real-Address Mode Exceptions**
Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FCMOVcc—Floating-Point Conditional Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DA C0+i</td>
<td>FCMOVB ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below (CF=1).</td>
</tr>
<tr>
<td>DA C8+i</td>
<td>FCMOVE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if equal (ZF=1).</td>
</tr>
<tr>
<td>DA D0+i</td>
<td>FCMOVBE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>DA D8+i</td>
<td>FCMOVU ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if unordered (PF=1).</td>
</tr>
<tr>
<td>DB C0+i</td>
<td>FCMOVNE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below (CF=0).</td>
</tr>
<tr>
<td>DB C8+i</td>
<td>FCMOVNBE ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>DB D0+i</td>
<td>FCMOVNU ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Move if not unordered (PF=0).</td>
</tr>
</tbody>
</table>

NOTES:
1. See IA-32 Architecture Compatibility section below.

Description
Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic is given in the Description column above and in Chapter 8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOVcc instructions. Software can check if the FCMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS” in this chapter). If both the CMOV and FPU feature bits are set, the FCMOVcc instructions are supported.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation
IF condition TRUE
THEN ST(0) := ST(i);
FI;

FPU Flags Affected
C1 Set to 0 if stack underflow occurred.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.

Integer Flags Affected
None.
Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FCOM/FCOMP/FCOMPP—Compare Floating-Point Values

**Description**

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that –0.0 is equal to +0.0.

This instruction checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /2</td>
<td>FCOM m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m32fp.</td>
</tr>
<tr>
<td>DC /2</td>
<td>FCOM m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m64fp.</td>
</tr>
<tr>
<td>D8 D0+i</td>
<td>FCOM ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i).</td>
</tr>
<tr>
<td>D8 D1</td>
<td>FCOM</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(1).</td>
</tr>
<tr>
<td>D8 /3</td>
<td>FCOMP m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m32fp and pop register stack.</td>
</tr>
<tr>
<td>DC /3</td>
<td>FCOMP m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with m64fp and pop register stack.</td>
</tr>
<tr>
<td>D8 D8+i</td>
<td>FCOMP ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i) and pop register stack.</td>
</tr>
<tr>
<td>D8 D9</td>
<td>FCOMP</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(1) and pop register stack.</td>
</tr>
<tr>
<td>DE D9</td>
<td>FCOMPP</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(1) and pop register stack twice.</td>
</tr>
</tbody>
</table>

### Table 3-21. FCOM/FCOMP/FCOMPP Results

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; SRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; SRC</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) = SRC</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered*</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTES:**

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.
Operation

CASE (relation of operands) OF
   ST > SRC: C3, C2, C0 := 000;
   ST < SRC: C3, C2, C0 := 001;
   ST = SRC: C3, C2, C0 := 100;
ESAC;

IF ST(0) or SRC = NaN or unsupported format
THEN
   #IA
   IF FPUControlWord.IM = 1
   THEN
      C3, C2, C0 := 111;
      FI;
   FI;
IF Instruction = FCOMP
THEN
   PopRegisterStack;
FI;
IF Instruction = FCOMPP
THEN
   PopRegisterStack;
   PopRegisterStack;
FI;

FPU Flags Affected
C1 Set to 0.
C0, C2, C3 See table on previous page.

Floating-Point Exceptions
#IS Stack underflow occurred.
#IA One or both operands are NaN values or have unsupported formats.
#D One or both operands are denormal values.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
   current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB F0+i</td>
<td>FCOMI ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i) and set status flags accordingly.</td>
</tr>
<tr>
<td>DF F0+i</td>
<td>FCOMIP ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.</td>
</tr>
<tr>
<td>DB E8+i</td>
<td>FUCOMI ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.</td>
</tr>
<tr>
<td>DF E8+i</td>
<td>FUCOMIP ST, ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.</td>
</tr>
</tbody>
</table>

Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

Table 3-22. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results

<table>
<thead>
<tr>
<th>Comparison Results*</th>
<th>ZF</th>
<th>PF</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 &gt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST0 &lt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST0 = ST(i)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered**</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTES:
* See the IA-32 Architecture Compatibility section below.
** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF, and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.
Operation

CASE (relation of operands) OF
   ST(0) > ST(i): ZF, PF, CF := 000;
   ST(0) < ST(i): ZF, PF, CF := 001;
   ST(0) = ST(i): ZF, PF, CF := 100;
ESAC;

IF Instruction is FCOMI or FCOMIP
   THEN
      IF ST(0) or ST(i) = NaN or unsupported format
         THEN
            #IA
            IF FPUControlWord.IM = 1
               THEN
                  ZF, PF, CF := 111;
               FI;
         FI;
   FI;

IF Instruction is FUCOMI or FUCOMIP
   THEN
      IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
         THEN
            ZF, PF, CF := 111;
         ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
            #IA;
            IF FPUControlWord.IM = 1
               THEN
                  ZF, PF, CF := 111;
               FI;
      FI;
   FI;

IF Instruction is FCOMIP or FUCOMIP
   THEN
      PopRegisterStack;
   FI;

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 Not affected.

Floating-Point Exceptions

#IS Stack underflow occurred.

#IA (FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats.
(FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.
Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FCOS—Cosine

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FF</td>
<td>FCOS</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with its approximate cosine.</td>
</tr>
</tbody>
</table>

Description

Computes the approximate cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range \(-2^{63}\) to \(+2^{63}\). The following table shows the results obtained when taking the cosine of various classes of numbers.

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (\infty)</td>
<td>*</td>
</tr>
<tr>
<td>- (F)</td>
<td>-1 to +1</td>
</tr>
<tr>
<td>- 0</td>
<td>+1</td>
</tr>
<tr>
<td>+ 0</td>
<td>+1</td>
</tr>
<tr>
<td>+ (F)</td>
<td>-1 to +1</td>
</tr>
<tr>
<td>+ (\infty)</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:

- \(F\) Means finite floating-point value.
- * Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range \(-2^{63}\) to \(+2^{63}\) can be reduced to the range of the instruction by subtracting an appropriate integer multiple of \(2\pi\). However, even within the range \(-2^{63}\) to \(+2^{63}\), inaccurate results can occur because the finite approximation of \(\pi\) used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than \(3\pi/8\). See the sections titled “Approximation of \(\pi\)” and “Transcendental Instruction Accuracy” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a discussion of the proper value to use for \(\pi\) in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

IF |ST(0)| < \(2^{63}\) THEN
  C2 := 0;
  ST(0) := FCOS(ST(0)); // approximation of cosine
ELSE (* Source operand is out-of-range *)
  C2 := 1;
FI;
FPU Flags Affected

C1
Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
Undefined if C2 is 1.

C2
Set to 1 if outside range \((-2^{63} < \text{source operand} < +2^{63})\); otherwise, set to 0.
C0, C3
Undefined.

Floating-Point Exceptions

#IS  Stack underflow occurred.
#IA  Source operand is an SNaN value, ∞, or unsupported format.
#D   Source is a denormal value.
#P   Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM   CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF   If there is a pending x87 FPU exception.
#UD   If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FDECSTP—Decrement Stack-Top Pointer

Description
Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
IF TOP = 0
    THEN TOP := 7;
    ELSE TOP := TOP – 1;
FI;

FPU Flags Affected
The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions
None.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F6</td>
<td>FDECSTP</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement TOP field in FPU status word.</td>
</tr>
</tbody>
</table>
FDIV/FDIVP/FIDIV—Divide

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /6</td>
<td>FDIV m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m32fp and store result in ST(0).</td>
</tr>
<tr>
<td>DC /6</td>
<td>FDIV m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m64fp and store result in ST(0).</td>
</tr>
<tr>
<td>D8 F0+i</td>
<td>FDIV ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DC F8+i</td>
<td>FDIV ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(i) by ST(0) and store result in ST(i).</td>
</tr>
<tr>
<td>DE F8+i</td>
<td>FDIVP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE F9</td>
<td>FDIVP</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.</td>
</tr>
<tr>
<td>DA /6</td>
<td>FIDIV m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m32int and store result in ST(0).</td>
</tr>
<tr>
<td>DE /6</td>
<td>FIDIV m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Divide ST(0) by m16int and store result in ST(0).</td>
</tr>
</tbody>
</table>

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.
Table 3-24. FDIV/FDIVP/FIDIV Results

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>*</td>
</tr>
<tr>
<td>$-F$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$-I$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$-0$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$+0$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$+I$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$+F$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**
F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF SRC = 0
THEN
#Z;
ELSE
IF Instruction is FIDIV
THEN
DEST := DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
ELSE (* Source operand is floating-point value *)
DEST := DEST / SRC;
FI;
FI;
IF Instruction = FDIVP
THEN
PopRegisterStack;
FI;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.
Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
±∞ / ±∞; ±0 / ±0
#D Source is a denormal value.
#Z DEST / ±0, where DEST is not equal to ±0.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.
FDIVR/FDIVRP/FIDIVR—Reverse Divide

**Description**

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.
When the source operand is an integer 0, it is treated as a +0. This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF DEST = 0
THEN
  #Z;
ELSE
  IF Instruction = FIDIVR
  THEN
    DEST := ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
  ELSE (* Source operand is floating-point value *)
    DEST := SRC / DEST;
  FI;
FI;

IF Instruction = FDIVRP
THEN
  PopRegisterStack;
FI;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

---

**Table 3-25. FDIVR/FDIVRP/FIDIVR Results**

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>−∞</td>
<td>−∞</td>
<td>*</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−F</td>
<td>+0</td>
<td>+F</td>
<td>**</td>
<td>**</td>
<td>−F</td>
<td>−0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−I</td>
<td>+0</td>
<td>+F</td>
<td>**</td>
<td>**</td>
<td>−F</td>
<td>−0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>−0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>−0</td>
<td>−0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+I</td>
<td>−0</td>
<td>−F</td>
<td>**</td>
<td>**</td>
<td>+F</td>
<td>+0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+F</td>
<td>−0</td>
<td>−F</td>
<td>**</td>
<td>**</td>
<td>+F</td>
<td>+0</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>−∞</td>
<td>−∞</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Operand is an SNaN value or unsupported format.
  ±∞ / ±∞; ±0 / ±0
#D Source is a denormal value.
#Z SRC / ±0, where SRC is not equal to ±0.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.
FFREE—Free Floating-Point Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD C0+i</td>
<td>FFREE ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Sets tag for ST(i) to empty.</td>
</tr>
</tbody>
</table>

**Description**
Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**
TAG(i) := 11B;

**FPU Flags Affected**
C0, C1, C2, C3 undefined.

**Floating-Point Exceptions**
None

**Protected Mode Exceptions**
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**
Same exceptions as in protected mode.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**
Same exceptions as in protected mode.
FICOM/FICOMP—Compare Integer

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

These instructions perform an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to "unordered."

The sign of zero is ignored, so that –0.0 := +0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

CASE (relation of operands) OF
  ST(0) > SRC: C3, C2, C0 := 000;
  ST(0) < SRC: C3, C2, C0 := 001;
  ST(0) = SRC: C3, C2, C0 := 100;
  Unordered: C3, C2, C0 := 111;
ESAC;

IF Instruction = FICOMP
  THEN
    PopRegisterStack;
  FI;

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 See table on previous page.

Floating-Point Exceptions

#IS Stack underflow occurred.
#IA One or both operands are NaN values or have unsupported formats.
#D One or both operands are denormal values.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FILD—Load Integer

### Description
Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation
TOP := TOP − 1;
ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);

### FPU Flags Affected
- C1 Set to 1 if stack overflow occurred; set to 0 otherwise.
- C0, C2, C3 Undefined.

### Floating-Point Exceptions
- #IS Stack overflow occurred.

### Protected Mode Exceptions
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used.

### Opcode Instruction 64-Bit Mode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /0</td>
<td>FILD m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m16int onto the FPU register stack.</td>
</tr>
<tr>
<td>DB /0</td>
<td>FILD m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m32int onto the FPU register stack.</td>
</tr>
<tr>
<td>DF /5</td>
<td>FILD m64int</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m64int onto the FPU register stack.</td>
</tr>
</tbody>
</table>
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
**FINCSTP—Increment Stack-Top Pointer**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F7</td>
<td>FINCSTP</td>
<td>Valid</td>
<td>Valid</td>
<td>Increment the TOP field in the FPU status register.</td>
</tr>
</tbody>
</table>

**Description**

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF TOP = 7
   THEN TOP := 0;
   ELSE TOP := TOP + 1;
FI;

**FPU Flags Affected**

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
**FINIT/FNINIT—Initialize Floating-Point Unit**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B DB E3</td>
<td>FINIT</td>
<td>Valid</td>
<td>Valid</td>
<td>Initialize FPU after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>DB E3</td>
<td>FNINIT¹</td>
<td>Valid</td>
<td>Valid</td>
<td>Initialize FPU without checking for pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. See IA-32 Architecture Compatibility section below.

**Description**

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers. This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

**Operation**

FPUC ControlWord := 037FH;
FPUC StatusWord := 0;
FPUC TagWord := FFFFH;
FPUC DataPointer := 0;
FPUC InstructionPointer := 0;
FPUC LastInstructionOpcode := 0;

**FPU Flags Affected**

C0, C1, C2, C3 set to 0.

**Floating-Point Exceptions**

None.
Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FIST/FISTP—Store Integer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/S Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /2</td>
<td>FIST m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m16int.</td>
</tr>
<tr>
<td>DB /2</td>
<td>FIST m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m32int.</td>
</tr>
<tr>
<td>DF /3</td>
<td>FISTP m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m16int and pop register stack.</td>
</tr>
<tr>
<td>DB /3</td>
<td>FISTP m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m32int and pop register stack.</td>
</tr>
<tr>
<td>DF /7</td>
<td>FISTP m64int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m64int and pop register stack.</td>
</tr>
</tbody>
</table>

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>−1 ≤ F &lt; −0</td>
<td>−1</td>
</tr>
<tr>
<td>−0</td>
<td>**</td>
</tr>
<tr>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>+0 &lt; F &lt; +1</td>
<td>**</td>
</tr>
<tr>
<td>F ≥ +1</td>
<td>+1</td>
</tr>
<tr>
<td>+ ∞ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>*</td>
</tr>
</tbody>
</table>

NOTES:

F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-operation (#IA) exception.
** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
**Operation**

DEST := Integer(ST(0));

IF Instruction = FISTP
    THEN
      PopRegisterStack;

FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Indicates rounding direction of if the inexact exception (#P) is generated: 0 := not roundup; 1 := roundup.
    Set to 0 otherwise.

C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.

#IA  Converted value is too large for the destination format.
    Source operand is an SNaN, QNaN, ±∞, or unsupported format.

#P  Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)  If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD  If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS  If a memory operand effective address is outside the SS segment limit.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD  If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code)  If a page fault occurs.

#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.

#UD  If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FISTTP—Store Integer With Truncation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF /1</td>
<td>FISTTP m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m16int with truncation.</td>
</tr>
<tr>
<td>DB /1</td>
<td>FISTTP m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m32int with truncation.</td>
</tr>
<tr>
<td>DD /1</td>
<td>FISTTP m64int</td>
<td>Valid</td>
<td>Valid</td>
<td>Store ST(0) in m64int with truncation.</td>
</tr>
</tbody>
</table>

**Description**

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>$F \leq -1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1 &lt; F &lt; +1$</td>
<td>$0$</td>
</tr>
<tr>
<td>$F = 1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+\infty$ or Value Too Large for DEST Format</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>*</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-operation (#IA) exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

DEST := ST;
pop ST;

**Flags Affected**

C1 is cleared; C0, C2, C3 undefined.

**Numeric Exceptions**

Invalid, Stack Invalid (stack underflow), Precision.

**Protected Mode Exceptions**

#GP(0) If the destination is in a nonwritable segment.
For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#NM If CR0.EM[bit 2] = 1.
If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
If the LOCK prefix is used.
Real Address Mode Exceptions
GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM If CR0.EM[bit 2] = 1.
   If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
   If the LOCK prefix is used.

Virtual 8086 Mode Exceptions
GP(0) If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM If CR0.EM[bit 2] = 1.
   If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.SSE3[bit 0] = 0.
   If the LOCK prefix is used.
#PF(fault-code) For a page fault.
#AC(0) For unaligned memory reference if the current privilege is 3.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
   If the LOCK prefix is used.
FLD—Load Floating-Point Value

Description

Pushes the source operand onto the FPU register stack. The source operand can be in single precision, double precision, or double extended-precision floating-point format. If the source operand is in single precision or double precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

**NOTE**

When the FLD instruction loads a denormal value and the DM bit in the CW is not masked, an exception is flagged but the value is still pushed onto the x87 stack.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF SRC is ST(i)
  THEN
  temp := ST(i);
  F1;
  TOP := TOP – 1;
IF SRC is memory-operand
  THEN
    ST(0) := ConvertToDoubleExtendedPrecisionFP(SRC);
  ELSE (* SRC is ST(i) *)
    ST(0) := temp;
  F1;

**FPU Flags Affected**

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.

C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow or overflow occurred.

#IA Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).

#D Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /0</td>
<td>FLD m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m32fp onto the FPU register stack.</td>
</tr>
<tr>
<td>DD /0</td>
<td>FLD m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m64fp onto the FPU register stack.</td>
</tr>
<tr>
<td>DB /5</td>
<td>FLD m80fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Push m80fp onto the FPU register stack.</td>
</tr>
<tr>
<td>D9 C0+i</td>
<td>FLD ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Push ST(i) onto the FPU register stack.</td>
</tr>
</tbody>
</table>
Protected Mode Exceptions

#GP(0) If destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
**Description**

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include \( +1.0 \), \( +0.0 \), \( \log_2 10 \), \( \log_2 e \), \( \pi \), \( \log_{10} 2 \), and \( \log_e 2 \). For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled “Approximation of Pi” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of the \( \pi \) constant.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

**Operation**

\[
\text{TOP} := \text{TOP} - 1; \\
\text{ST}(0) := \text{CONSTANT};
\]

**FPU Flags Affected**

- C1 Set to 1 if stack overflow occurred; otherwise, set to 0.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack overflow occurred.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FLDCW—Load x87 FPU Control Word

Description
Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU’s mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
FPUControlWord := SRC;

FPU Flags Affected
C0, C1, C2, C3 undefined.

Floating-Point Exceptions
None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next “waiting” floating-point instruction.

Protected Mode Exceptions
- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /5</td>
<td>FLDCW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FPU control word from m2byte.</td>
</tr>
</tbody>
</table>
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FLDENV—Load x87 FPU Environment

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /4</td>
<td>FLDENV m14/28byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FPU environment from m14byte or m28byte.</td>
</tr>
</tbody>
</table>

**Description**

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

FPUCtrlWord := SRC[FPUCtrlWord];
FPUStrWord := SRC[FPUStrWord];
FPUCntWord := SRC[FPUCntWord];
FPUDtaPointer := SRC[FPUDtaPointer];
FPUIntrPointer := SRC[FPUIntrPointer];
FPULstInstnOdpc := SRC[FPULstInstnOdpc];

**FPU Flags Affected**

The C0, C1, C2, C3 flags are loaded.

**Floating-Point Exceptions**

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next "waiting" floating-point instruction.
Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FMUL/FMULP/FIMUL—Multiply

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value) and stores the product in the ST(0) register. The two-operand version multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or $\infty$. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /1</td>
<td>FMUL m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m32fp and store result in ST(0).</td>
</tr>
<tr>
<td>DC /1</td>
<td>FMUL m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m64fp and store result in ST(0).</td>
</tr>
<tr>
<td>D8 C8+i</td>
<td>FMUL ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DC C8+i</td>
<td>FMUL ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(i) by ST(0) and store result in ST(i).</td>
</tr>
<tr>
<td>DE C8+i</td>
<td>FMULP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.</td>
</tr>
<tr>
<td>DE C9</td>
<td>FMULP</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.</td>
</tr>
<tr>
<td>DA /1</td>
<td>FIMUL m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m32int and store result in ST(0).</td>
</tr>
<tr>
<td>DE /1</td>
<td>FIMUL m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply ST(0) by m16int and store result in ST(0).</td>
</tr>
</tbody>
</table>
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FIMUL
    THEN
        DEST := DEST * ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST := DEST * SRC;
FI;

IF Instruction = FMULP
    THEN
        PopRegisterStack;
FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.

C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.

#IA  Operand is an SNaN value or unsupported format.
    One operand is ±0 and the other is ±∞.

#D  Source operand is a denormal value.

#U  Result is too small for destination format.

#O  Result is too large for destination format.

#P  Value cannot be represented exactly in destination format.
Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FNOP—No Operation

### Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 D0</td>
<td>FNOP</td>
<td>Valid</td>
<td>Valid</td>
<td>No operation is performed.</td>
</tr>
</tbody>
</table>

### Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register and the FPU Instruction Pointer.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.
FPATAN—Partial Arctangent

**Description**
Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than $+\pi$.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point ($-X,Y$) is in the second quadrant, resulting in an angle between $\pi/2$ and $\pi$, while a point ($X,-Y$) is in the fourth quadrant, resulting in an angle between $0$ and $-\pi/2$. A point ($-X,-Y$) is in the third quadrant, giving an angle between $-\pi/2$ and $-\pi$.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>$-\infty$</th>
<th>$-F$</th>
<th>$-0$</th>
<th>$+0$</th>
<th>$+F$</th>
<th>$+\infty$</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$-3\pi/4^*$</td>
<td>$-\pi/2$</td>
<td>$-\pi/2$</td>
<td>$-\pi/2$</td>
<td>$-\pi/2$</td>
<td>$-\pi/4^*$</td>
<td>NaN</td>
</tr>
<tr>
<td>$-F$</td>
<td>$-\pi$ to $-\pi/2$</td>
<td>$-\pi/2$</td>
<td>$-\pi/2$</td>
<td>$-\pi/2$</td>
<td>$-\pi/2$ to $-0$</td>
<td>$-0$</td>
<td>NaN</td>
</tr>
<tr>
<td>$-0$</td>
<td>$-0^*$</td>
<td>$-0^*$</td>
<td>$-0^*$</td>
<td>$-0$</td>
<td>$-0$</td>
<td>$-0$</td>
<td>NaN</td>
</tr>
<tr>
<td>$+0$</td>
<td>$+0^*$</td>
<td>$+0^*$</td>
<td>$+0^*$</td>
<td>$+0$</td>
<td>$+0$</td>
<td>$+0$</td>
<td>NaN</td>
</tr>
<tr>
<td>$+F$</td>
<td>$+0^*$</td>
<td>$+0^*$</td>
<td>$+0^*$</td>
<td>$+0$</td>
<td>$+0$</td>
<td>$+0$</td>
<td>NaN</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+3\pi/4^*$</td>
<td>$+\pi/2$</td>
<td>$+\pi/2$</td>
<td>$+\pi/2$ to $+0$</td>
<td>$+0$</td>
<td>NaN</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**
- $F$ Means finite floating-point value.
- * Table 8-10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, specifies that the ratios $0/0$ and $\infty/\infty$ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the $0/0$ or $\infty/\infty$ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, arctangent($0,0$) etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**
The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$0 \leq |ST(1)| < |ST(0)| < +\infty$
**Operation**

ST(1) := arctan(ST(1) / ST(0));
PopRegisterStack;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.
C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.
#IA  Source operand is an SNan value or unsupported format.
#D   Source operand is a denormal value.
#U   Result is too small for destination format.
#P   Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF  If there is a pending x87 FPU exception.
#UD  If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FPREM—Partial Remainder

Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or modulus), and stores the result in ST(0). The remainder represents the following value:

Remainder := ST(0) − (Q ∗ ST(1))

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-31. FPREM Results

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>-∞</th>
<th>-F</th>
<th>-0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>-F</td>
<td>ST(0)</td>
<td>-F or -0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-F or -0</td>
<td>ST(0)</td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>*</td>
<td>*</td>
<td>-0</td>
<td>-0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>ST(0)</td>
<td>+F or +0</td>
<td>*</td>
<td>*</td>
<td>+F or +0</td>
<td>ST(0)</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:

F Means finite floating-point value.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the partial remainder. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU.
status word. This information is important in argument reduction for the tangent function (using a modulus of \(\pi/4\)), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
D := \text{exponent(ST(0))} - \text{exponent(ST(1))};
\]

**IF** \(D < 64\)

**THEN**

\[
Q := \text{Integer(TruncateTowardZero(ST(0) / ST(1)))};
ST(0) := ST(0) - (ST(1) \times Q);
C2 := 0;
C0, C3, C1 := \text{LeastSignificantBits(Q); (* Q2, Q1, Q0 *)}
\]

**ELSE**

\[
C2 := 1;
N := \text{An implementation-dependent number between 32 and 63};
QQ := \text{Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^{(D - N)})};
ST(0) := ST(0) - (ST(1) \times QQ \times 2^{(D - N)})
\]

**FI**;

**FPU Flags Affected**

- \(C0\) Set to bit 2 (Q2) of the quotient.
- \(C1\) Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
- \(C2\) Set to 0 if reduction complete; set to 1 if incomplete.
- \(C3\) Set to bit 1 (Q1) of the quotient.

**Floating-Point Exceptions**

- \(#IS\) Stack underflow occurred.
- \(#IA\) Source operand is an SNaN value, modulus is 0, dividend is \(\infty\), or unsupported format.
- \(#D\) Source operand is a denormal value.
- \(#U\) Result is too small for destination format.

**Protected Mode Exceptions**

- \(#NM\) CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- \(#MF\) If there is a pending x87 FPU exception.
- \(#UD\) If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
**FPREM1—Partial Remainder**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F5</td>
<td>FPREM1</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1).</td>
</tr>
</tbody>
</table>

**Description**

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or modulus), and stores the result in ST(0). The remainder represents the following value:

\[
\text{Remainder} = ST(0) - (Q \times ST(1))
\]

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of \([ST(0) / ST(1)]\) toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>ST(0)</td>
<td>±F or −0</td>
<td>*</td>
<td>*</td>
<td>±F or −0</td>
<td>ST(0)</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>*</td>
<td>*</td>
<td>−0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>ST(0)</td>
<td>±F or +0</td>
<td>*</td>
<td>*</td>
<td>±F or +0</td>
<td>ST(0)</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F  Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is −∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the "Operation" section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the partial remainder. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of \(\pi/4\)), because it locates the original angle in the correct one of eight sectors of the unit circle.
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[ D := \text{exponent(ST(0))} - \text{exponent(ST(1))}; \]

IF \( D < 64 \)

THEN

\[ Q := \text{Integer(RoundTowardNearestInteger(ST(0) / ST(1))}; \]

\[ \text{ST(0)} := \text{ST(0)} - (\text{ST(1)} \times Q); \]

\[ C2 := 0; \]

\[ C0, C3, C1 := \text{LeastSignificantBits(Q)}; \] (* Q2, Q1, Q0 *)

ELSE

\[ C2 := 1; \]

\[ N := \text{An implementation-dependent number between 32 and 63}; \]

\[ QQ := \text{Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^{D - N})}; \]

\[ \text{ST(0)} := \text{ST(0)} - (\text{ST(1)} \times QQ \times 2^{D - N}); \]

FI;

**FPU Flags Affected**

- \( C0 \) Set to bit 2 (Q2) of the quotient.
- \( C1 \) Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
- \( C2 \) Set to 0 if reduction complete; set to 1 if incomplete.
- \( C3 \) Set to bit 1 (Q1) of the quotient.

**Floating-Point Exceptions**

- \#IS Stack underflow occurred.
- \#IA Source operand is an SNaN value, modulus (divisor) is 0, dividend is \( \infty \), or unsupported format.
- \#D Source operand is a denormal value.
- \#U Result is too small for destination format.

**Protected Mode Exceptions**

- \#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- \#MF If there is a pending x87 FPU exception.
- \#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FPTAN—Partial Tangent

Description
Computes the approximate tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than ±2^63. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^63 to +2^63 can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^63 to +2^63, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than 3π/8. See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a discussion of the proper value to use for π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F2</td>
<td>FPTAN</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(0) with its approximate tangent and push 1 onto the FPU stack.</td>
</tr>
</tbody>
</table>

Table 3-33. FPTAN Results

<table>
<thead>
<tr>
<th>ST(0) SRC</th>
<th>ST(0) DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
</tr>
<tr>
<td>−F</td>
<td>−F to +F</td>
</tr>
<tr>
<td>−0</td>
<td>0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+F</td>
<td>−F to +F</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

NOTES:
F Means finite floating-point value.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Vol. 2A 3-421
Operation
IF ST(0) < 2^{63}
    THEN
        C2 := 0;
        ST(0) := fptan(ST(0)); // approximation of tan
        TOP := TOP – 1;
        ST(0) := 1.0;
    ELSE (* Source operand is out-of-range *)
        C2 := 1;
FI;

FPU Flags Affected
C1 Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
Set if result was rounded up; cleared otherwise.
C2 Set to 1 if outside range (−2^{63} < source operand < +2^{63}); otherwise, set to 0.
C0, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow or overflow occurred.
#IA Source operand is an SNaN value, ∞, or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FRNDINT—Round to Integer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FC</td>
<td>FRNDINT</td>
<td>Valid</td>
<td>Valid</td>
<td>Round ST(0) to an integer.</td>
</tr>
</tbody>
</table>

**Description**

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is $\infty$, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[ \text{ST}(0) := \text{RoundToIntegralValue}(\text{ST}(0)); \]

**FPU Flags Affected**

- C1: Set to 0 if stack underflow occurred.
- C1: Set if result was rounded up; cleared otherwise.
- C0, C2, C3: Undefined.

**Floating-Point Exceptions**

- #IS: Stack underflow occurred.
- #IA: Source operand is an SNaN value or unsupported format.
- #D: Source operand is a denormal value.
- #P: Source operand is not an integral value.

**Protected Mode Exceptions**

- #NM: CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF: If there is a pending x87 FPU exception.
- #UD: If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
FRSTOR—Restore x87 FPU State

FRSTOR—Restore x87 FPU State

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

FPUControlWord := SRC[FPUControlWord];
FPUStatusWord := SRC[FPUStatusWord];
FPUTagWord := SRC[FPUTagWord];
FPUDataPointer := SRC[FPUDataPointer];
FPUInstructionPointer := SRC[FPUInstructionPointer];
FPULastInstructionOpcode := SRC[FPULastInstructionOpcode];
ST(0) := SRC[ST(0)];
ST(1) := SRC[ST(1)];
ST(2) := SRC[ST(2)];
ST(3) := SRC[ST(3)];
ST(4) := SRC[ST(4)];
ST(5) := SRC[ST(5)];
ST(6) := SRC[ST(6)];
ST(7) := SRC[ST(7)];

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next “waiting” floating-point instruction.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.

#UD If the LOCK prefix is used.
FSAVE/FNSAVE—Store x87 FPU State

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B DD /6</td>
<td>FSAVE m94/108byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU state to m94byte or m108byte after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.</td>
</tr>
<tr>
<td>DD /6</td>
<td>FNSAVE¹ m94/108byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU environment to m94byte or m108byte without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.</td>
</tr>
</tbody>
</table>

NOTES:
1. See IA-32 Architecture Compatibility section below.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see “FINIT/FNINIT—Initialize Floating-Point Unit” in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a “clean” FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.
**Operation**

(* Save FPU State and Registers *)

DEST[FPUCtrlWord] := FPUCtrlWord;
DEST[FPUSTatusWord] := FPUSTatusWord;
DEST[FPUTagWord] := FPUTagWord;
DEST[FPUDataPointer] := FPUDataPointer;
DEST[FPUIInstructionPointer] := FPUIInstructionPointer;
DEST[FPULastInstructionOpcode] := FPULastInstructionOpcode;

DEST[ST(0)] := ST(0);
DEST[ST(1)] := ST(1);
DEST[ST(2)] := ST(2);
DEST[ST(3)] := ST(3);
DEST[ST(4)] := ST(4);
DEST[ST(5)] := ST(5);
DEST[ST(6)] := ST(6);
DEST[ST(7)] := ST(7);

(* Initialize FPU *)

FPUCtrlWord := 037FH;
FPUSTatusWord := 0;
FPUTagWord := FFFFH;
FPUDataPointer := 0;
FPUIInstructionPointer := 0;
FPULastInstructionOpcode := 0;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are saved and then cleared.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#GP(0) If destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
**FSCALE—Scale**

**Description**

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[ ST(0) := ST(0) \times 2^{\text{RoundTowardZero(ST(1))}}; \]

**FPU Flags Affected**

- **C1**: Set to 0 if stack underflow occurred.
- **C0, C2, C3**: Undefined.

---

### Table 3-34. FSCALE Results

<table>
<thead>
<tr>
<th>ST(1)</th>
<th>−∞</th>
<th>−F</th>
<th>−0</th>
<th>+0</th>
<th>+F</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>NaN</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>−0</td>
<td>−F</td>
<td>−F</td>
<td>−F</td>
<td>−F</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>+0</td>
<td>+F</td>
<td>+F</td>
<td>+F</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>NaN</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

- F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[ ST(0) := ST(0) \times 2^{\text{RoundTowardZero(ST(1))}}; \]

**FPU Flags Affected**

- **C1**: Set to 0 if stack underflow occurred.
- **C0, C2, C3**: Undefined.
Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Source operand is an SNaN value or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM CR0_EM[bit 2] or CR0_TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FSIN—Sine

**Description**

Computes an approximation of the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range \(-2^{63}\) to \(+2^{63}\). The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>SRC (ST(0))</th>
<th>DEST (ST(0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
</tr>
<tr>
<td>−F</td>
<td>−1 to +1</td>
</tr>
<tr>
<td>0</td>
<td>−0</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>+F</td>
<td>−1 to +1</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range \(−2^{63}\) to \(+2^{63}\) can be reduced to the range of the instruction by subtracting an appropriate integer multiple of \(2\pi\). However, even within the range \(-2^{63}\) to \(+2^{63}\), inaccurate results can occur because the finite approximation of \(\pi\) used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than \(3\pi/4\). See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a discussion of the proper value to use for \(\pi\) in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\text{IF } -2^{63} < \text{ST}(0) < 2^{63} \text{ THEN}
\]

\[
\begin{align*}
& C2 := 0; \\
& \text{ST}(0) := \text{fsin} \left( \text{ST}(0) \right) \text{; // approximation of the mathematical sin function} \\
& \text{ELSE (" Source operand out of range ")}
\end{align*}
\]

\[
C2 := 1;
\]

**FPU Flags Affected**

- **C1**  Set to 0 if stack underflow occurred.
  Set if result was rounded up; cleared otherwise.

- **C2**  Set to 1 if outside range \((-2^{63} < \text{source operand} < +2^{63})\); otherwise, set to 0.

- **C0, C3**  Undefined.
Floating-Point Exceptions

#IS Stack underflow occurred.
#IA Source operand is an SNaN value, $\infty$, or unsupported format.
#D Source operand is a denormal value.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FSINCOS—Sine and Cosine

**Description**

Computes both the approximate sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range \(-2^{63}\) to \(+2^{63}\). The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

<table>
<thead>
<tr>
<th>SRC</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0)</td>
<td>ST(1) Cosine</td>
</tr>
<tr>
<td>(-\infty)</td>
<td>*</td>
</tr>
<tr>
<td>(-F)</td>
<td>(-1 + 1)</td>
</tr>
<tr>
<td>(-0)</td>
<td>(+1)</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+1)</td>
</tr>
<tr>
<td>(+F)</td>
<td>(-1 + 1)</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>*</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

- F Means finite floating-point value.
- * Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range \(-2^{63}\) to \(+2^{63}\) can be reduced to the range of the instruction by subtracting an appropriate integer multiple of \(2\pi\). However, even within the range \(-2^{63}\) to \(+2^{63}\), inaccurate results can occur because the finite approximation of \(\pi\) used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than \(3\pi/8\). See the sections titled “Approximation of \(\pi\)” and “Transcendental Instruction Accuracy” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a discussion of the proper value to use for \(\pi\) in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
Operation

IF ST(0) < $2^{63}$
THEN
    C2 := 0;
    TEMP := fcos(ST(0)); // approximation of cosine
    ST(0) := fsin(ST(0)); // approximation of sine
    TOP := TOP − 1;
    ST(0) := TEMP;
ELSE (* Source operand out of range *)
    C2 := 1;
FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs.
Set if result was rounded up; cleared otherwise.
C2 Set to 1 if outside range ($-2^{63} < $ source operand $ < +2^{63}$); otherwise, set to 0.
C0, C3 Undefined.

Floating-Point Exceptions

#IS Stack underflow or overflow occurred.
#IA Source operand is an SNaN value, $\infty$, or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
FSQRT—Square Root

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 FA</td>
<td>FSQRT</td>
<td>Valid</td>
<td>Valid</td>
<td>Computes square root of ST(0) and stores the result in ST(0).</td>
</tr>
</tbody>
</table>

**Description**

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-37. FSQRT Results**

<table>
<thead>
<tr>
<th>SRC (ST(0))</th>
<th>DEST (ST(0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>*</td>
</tr>
<tr>
<td>−芬</td>
<td>*</td>
</tr>
<tr>
<td>−芬</td>
<td>−芬</td>
</tr>
<tr>
<td>+芬</td>
<td>+芬</td>
</tr>
<tr>
<td>+芬</td>
<td>+芬</td>
</tr>
<tr>
<td>+ ∞</td>
<td>+ ∞</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

ST(0) := SquareRoot(ST(0));

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow occurred.
#IA Source operand is an SNaN value or unsupported format.
Source operand is a negative value (except for −0).
#D Source operand is a denormal value.
#P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FST/FSTP—Store Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 /2</td>
<td>FST m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m32fp.</td>
</tr>
<tr>
<td>DD /2</td>
<td>FST m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m64fp.</td>
</tr>
<tr>
<td>DD D0+i</td>
<td>FST ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to ST(i).</td>
</tr>
<tr>
<td>D9 /3</td>
<td>FSTP m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m32fp and pop register stack.</td>
</tr>
<tr>
<td>DD /3</td>
<td>FSTP m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m64fp and pop register stack.</td>
</tr>
<tr>
<td>DB /7</td>
<td>FSTP m80fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to m80fp and pop register stack.</td>
</tr>
<tr>
<td>DD D8+i</td>
<td>FSTP ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Copy ST(0) to ST(i) and pop register stack.</td>
</tr>
</tbody>
</table>

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single precision or double precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single precision or double precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value’s identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

DEST := ST(0);
IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 := not roundup; 1 := roundup.

C0, C2, C3 Undefined.
Floating-Point Exceptions

#IS  Stack underflow occurred.
#IA  If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U   Result is too small for the destination format.
#O   Result is too large for the destination format.
#P   Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)  If the destination is located in a non-writable segment.
        If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
        If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#NM    CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD    If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#NM  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD  If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#NM    CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.
#UD    If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#NM    CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF    If there is a pending x87 FPU exception.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD    If the LOCK prefix is used.
FSTCW/FNSTCW—Store x87 FPU Control Word

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B D9 /7</td>
<td>FSTCW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU control word to m2byte after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>D9 /7</td>
<td>FNSTCW1 m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU control word to m2byte without checking for pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

NOTES:
1. See IA-32 Architecture Compatibility section below.

Description
Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation
DEST := FPUControlWord;

FPU Flags Affected
The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions
None.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSTENV/FNSTENV—Store x87 FPU Environment

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B D9 /6</td>
<td>FSTENV m14/28byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU environment to m14byte or m28byte after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.</td>
</tr>
<tr>
<td>D9 /6</td>
<td>FNSTENV1 m14/28byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU environment to m14byte or m28byte without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.</td>
</tr>
</tbody>
</table>

NOTES:
1. See IA-32 Architecture Compatibility section below.

Description
Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation
DEST[FPUControlWord] := FPUControlWord;
DEST[FPUStatusWord] := FPUStatusWord;
DEST[FPUTagWord] := FPUTagWord;
DEST[FPUDataPointer] := FPUDataPointer;
DEST[FPUInstructionPointer] := FPUInstructionPointer;
DEST[FPULastInstructionOpcode] := FPULastInstructionOpcode;

FPU Flags Affected
The C0, C1, C2, and C3 are undefined.
Floating-Point Exceptions
None.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSTSW/FNSTSW—Store x87 FPU Status Word

Description
Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled "Branching and Conditional Moves on FPU Condition Codes" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility
When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation
DEST := FPUStatusWord;

FPU Flags Affected
The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions
None.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B DD /7</td>
<td>FSTSW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word at m2byte after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>9B DF E0</td>
<td>FSTSW AX</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>DD /7</td>
<td>FNSTSW m2byte</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word at m2byte without checking for pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>DF E0</td>
<td>FNSTSW AX</td>
<td>Valid</td>
<td>Valid</td>
<td>Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

NOTES:
1. See IA-32 Architecture Compatibility section below.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSUB/FSUBP/FISUB—Subtract

**Description**

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-38 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST − SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward −∞ mode, in which case the result is −0. This instruction also guarantees that +0 − (−0) = +0, and that −0 − (+0) = −0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /4</td>
<td>FSUB m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m32fp from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DC /4</td>
<td>FSUB m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m64fp from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DB E0+i</td>
<td>FSUB ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(i) from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DC E8+i</td>
<td>FSUB ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(i) and store result in ST(i).</td>
</tr>
<tr>
<td>DE E8+i</td>
<td>FSUBP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.</td>
</tr>
<tr>
<td>DE E9</td>
<td>FSUBP</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.</td>
</tr>
<tr>
<td>DA /4</td>
<td>FISUB m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m32int from ST(0) and store result in ST(0).</td>
</tr>
<tr>
<td>DE /4</td>
<td>FISUB m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract m16int from ST(0) and store result in ST(0).</td>
</tr>
</tbody>
</table>
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FISUB

THEN

\[ \text{DEST} := \text{DEST} - \text{ConvertToDoubleExtendedPrecisionFP}(	ext{SRC}); \]

ELSE (* Source operand is floating-point value *)

\[ \text{DEST} := \text{DEST} - \text{SRC}; \]

FI;

IF Instruction = FSUBP

THEN

PopRegisterStack;

FI;

**FPU Flags Affected**

C1  Set to 0 if stack underflow occurred.
    Set if result was rounded up; cleared otherwise.

C0, C2, C3  Undefined.

**Floating-Point Exceptions**

#IS  Stack underflow occurred.

#IA  Operand is an SNan value or unsupported format.
     Operands are infinities of like sign.

#D  Source operand is a denormal value.

#U  Result is too small for destination format.

#O  Result is too large for destination format.

#P  Value cannot be represented exactly in destination format.

**Table 3-38. FSUB/FSUBP/FISUB Results**

<table>
<thead>
<tr>
<th>DEST</th>
<th>− ∞</th>
<th>− F or − I</th>
<th>− 0</th>
<th>+ 0</th>
<th>+ F or + I</th>
<th>+ ∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>− ∞</td>
<td>*</td>
<td>− ∞</td>
<td>− ∞</td>
<td>− ∞</td>
<td>− ∞</td>
<td>− ∞</td>
<td>NaN</td>
</tr>
<tr>
<td>− F</td>
<td>+ ∞</td>
<td>±F or ±0</td>
<td>DEST</td>
<td>DEST</td>
<td>− F</td>
<td>− ∞</td>
<td>NaN</td>
</tr>
<tr>
<td>− 0</td>
<td>+ ∞</td>
<td>−SRC</td>
<td>±0</td>
<td>− 0</td>
<td>−SRC</td>
<td>− ∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+ 0</td>
<td>+ ∞</td>
<td>−SRC</td>
<td>+ 0</td>
<td>±0</td>
<td>−SRC</td>
<td>− ∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+ F</td>
<td>+ ∞</td>
<td>+ F</td>
<td>DEST</td>
<td>DEST</td>
<td>±F or ±0</td>
<td>− ∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+ ∞</td>
<td>+ ∞</td>
<td>+ ∞</td>
<td>+ ∞</td>
<td>+ ∞</td>
<td>+ ∞</td>
<td>*</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F  Means finite floating-point value.

I  Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
FSUBR/FSUBRP/FISUBR—Reverse Subtract

Description
Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single precision or double precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /5</td>
<td>FSUBR m32fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from m32fp and store result in ST(0).</td>
</tr>
<tr>
<td>DC /5</td>
<td>FSUBR m64fp</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from m64fp and store result in ST(0).</td>
</tr>
<tr>
<td>D8 E8+i</td>
<td>FSUBR ST(0), ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from ST(i) and store result in ST(0).</td>
</tr>
<tr>
<td>DC EO+i</td>
<td>FSUBR ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(i) from ST(0) and store result in ST(i).</td>
</tr>
<tr>
<td>DE EO+i</td>
<td>FSUBRP ST(i), ST(0)</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(i) from ST(0), store result in ST(i), and pop register stack.</td>
</tr>
<tr>
<td>DE E1</td>
<td>FSUBRP</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(1) from ST(0), store result in ST(1), and pop register stack.</td>
</tr>
<tr>
<td>DA /5</td>
<td>FISUBR m32int</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from m32int and store result in ST(0).</td>
</tr>
<tr>
<td>DE /5</td>
<td>FISUBR m16int</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract ST(0) from m16int and store result in ST(0).</td>
</tr>
</tbody>
</table>
This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FISUBR  
THEN  
DEST := ConvertToDoubleExtendedPrecisionFP(SRC) − DEST;  
ELSE (* Source operand is floating-point value *)  
DEST := SRC − DEST; FI;

IF Instruction = FSUBRP  
THEN  
PopRegisterStack; FI;

**FPU Flags Affected**

C1  
Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.

C0, C2, C3  
Undefined.

**Floating-Point Exceptions**

#IS  
Stack underflow occurred.

#IA  
Operand is an SNaN value or unsupported format.  
Operands are infinities of like sign.

#D  
Source operand is a denormal value.

#U  
Result is too small for destination format.

#O  
Result is too large for destination format.

#P  
Value cannot be represented exactly in destination format.

---

**Table 3-39. FSUBR/FSUBRP/FISUBR Results**

<table>
<thead>
<tr>
<th>DEST</th>
<th>−∞</th>
<th>−F or −1</th>
<th>−0</th>
<th>+0</th>
<th>+F or +1</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>×</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>−∞</td>
<td>±F or ±0</td>
<td>−DEST</td>
<td>−DEST</td>
<td>+ F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>−∞</td>
<td>SRC</td>
<td>±0</td>
<td>+0</td>
<td>SRC</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>−∞</td>
<td>SRC</td>
<td>−0</td>
<td>±0</td>
<td>SRC</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+ F</td>
<td>−∞</td>
<td>− F</td>
<td>−DEST</td>
<td>−DEST</td>
<td>±F or ±0</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+ ∞</td>
<td>−∞</td>
<td>− ∞</td>
<td>−∞</td>
<td>− ∞</td>
<td>− ∞</td>
<td>− ∞</td>
<td>*NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F  
Means finite floating-point value.

I  
Means integer.

*  
Indicates floating-point invalid-arithmetic-operand (#IA) exception.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
FTST—TEST

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; 0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; 0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) = 0.0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered” and the invalid operation exception is generated.

The sign of zero is ignored, so that \((-0.0 := 0.0)\).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

CASE (relation of operands) OF
- Not comparable: C3, C2, C0 := 111;
- ST(0) > 0.0: C3, C2, C0 := 000;
- ST(0) < 0.0: C3, C2, C0 := 001;
- ST(0) = 0.0: C3, C2, C0 := 100;
ESAC;

FPU Flags Affected

C1 Set to 0.
C0, C2, C3 See Table 3-40.

Floating-Point Exceptions

#IS Stack underflow occurred.
#IA The source operand is a NaN value or is in an unsupported format.
#D The source operand is a denormal value.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating-Point Values

Description
Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that –0.0 is equal to +0.0.

Table 3-41. FUCOM/FUCOMP/FUCOMPP Results

<table>
<thead>
<tr>
<th>Comparison Results*</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 &gt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST0 &lt; ST(i)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST0 = ST(i)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTES:
* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
Operation

CASE (relation of operands) OF
    ST > SRC:  C3, C2, C0 := 000;
    ST < SRC:  C3, C2, C0 := 001;
    ST = SRC:  C3, C2, C0 := 100;
ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
    THEN
        C3, C2, C0 := 111;
    ELSE (* ST(0) or SRC is SNaN or unsupported format *)
        #IA;
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 := 111;
            FI;
    FI;

IF Instruction = FUCOMP
    THEN
        PopRegisterStack;
    FI;

IF Instruction = FUCOMPP
    THEN
        PopRegisterStack;
    FI;

FPU Flags Affected

C1 Set to 0 if stack underflow occurred.
C0, C2, C3 See Table 3-41.

Floating-Point Exceptions

#IS Stack underflow occurred.
#IA One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
#D One or both operands are denormal values.

Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
FXAM—Examine Floating-Point

Description
Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

Table 3-42. FXAM Results

<table>
<thead>
<tr>
<th>Class</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsupported</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Normal finite number</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Infinity</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zero</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Empty</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Denormal number</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation
C1 := sign bit of ST; (* 0 for positive, 1 for negative *)
CASE (class of value or number in ST(0)) OF
  Unsupported: C3, C2, C0 := 000;
  NaN: C3, C2, C0 := 001;
  Normal: C3, C2, C0 := 010;
  Infinity: C3, C2, C0 := 011;
  Zero: C3, C2, C0 := 100;
  Empty: C3, C2, C0 := 101;
  Denormal: C3, C2, C0 := 110;
ESAC.

FPU Flags Affected
C1 Sign of value in ST(0).
C0, C2, C3 See Table 3-42.

Floating-Point Exceptions
None.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
FXCH—Exchange Register Contents

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 C8+i</td>
<td>FXCH ST(i)</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange the contents of ST(0) and ST(i).</td>
</tr>
<tr>
<td>D9 C9</td>
<td>FXCH</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange the contents of ST(0) and ST(1).</td>
</tr>
</tbody>
</table>

**Description**

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF (Number-of-operands) is 1

THEN

\[
\begin{align*}
\text{temp} & := \text{ST}(0); \\
\text{ST}(0) & := \text{SRC}; \\
\text{SRC} & := \text{temp}; \\
\end{align*}
\]

ELSE

\[
\begin{align*}
\text{temp} & := \text{ST}(0); \\
\text{ST}(0) & := \text{ST}(1); \\
\text{ST}(1) & := \text{temp}; \\
\end{align*}
\]

F:

**FPU Flags Affected**

C1 Set to 0.

C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow occurred.

**Protected Mode Exceptions**

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FXRSTOR—Restore x87 FPU, MMX, XMM, and MXCSR State**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-43 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-46 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-47 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-43, Table 3-46, or Table 3-47. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

**Operation**

IF 64-Bit Mode

THEN

(x87 FPU, MMX, XMM15-XMM0, MXCSR) := Load(SRC);

ELSE

(x87 FPU, MMX, XMM7-XMM0, MXCSR) := Load(SRC);

FI;

**x87 FPU and SIMD Floating-Point Exceptions**

None.
Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See align-
ment check exception [#AC] below.)
For an attempt to set reserved bits in MXCSR.

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
If instruction is preceded by a LOCK prefix.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 16-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
For an attempt to set reserved bits in MXCSR.

#NM If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code) For a page fault.

#AC For unaligned memory reference.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.
If memory operand is not aligned on a 16-byte boundary, regardless of segment.
For an attempt to set reserved bits in MXCSR.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.
If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.
If instruction is preceded by a LOCK prefix.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
**FXSAVE—Save x87 FPU, MMX Technology, and SSE State**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:rr/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512-byte memory location depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area.

The operation of FXSAVE in non-64-bit modes is described first.

### Non-64-Bit Mode Operation

Table 3-43 shows the layout of the state information in memory when the processor is operating in legacy modes.

#### Table 3-43. Non-64-Bit-Mode Layout of FXSAVE and FXRSTOR Memory Region

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rsvd</td>
<td>FCS</td>
<td>FIP[31:0]</td>
<td>FOP</td>
<td>Rsvd</td>
<td>FTw</td>
<td>FSW</td>
<td>FCW</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td>Rsvd</td>
<td>FDS</td>
<td>FDP[31:0]</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST0/MM0</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST1/MM1</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST2/MM2</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST3/MM3</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST4/MM4</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST5/MM5</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST6/MM6</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST7/MM7</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM0</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM1</td>
<td>176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM2</td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM3</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM4</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM5</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM6</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM7</td>
<td>272</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-43 are defined in Table 3-44.

Table 3-44. Field Definitions

<table>
<thead>
<tr>
<th>Field</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCW</td>
<td>x87 FPU Control Word (16 bits). See Figure 8-6 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the layout of the x87 FPU control word.</td>
</tr>
<tr>
<td>FSW</td>
<td>x87 FPU Status Word (16 bits). See Figure 8-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the layout of the x87 FPU status word.</td>
</tr>
<tr>
<td>Abridged FTW</td>
<td>x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.</td>
</tr>
<tr>
<td>FOP</td>
<td>x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the layout of the x87 FPU opcode field.</td>
</tr>
</tbody>
</table>
| FIP            | x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed:
  32-bit mode — 32-bit IP offset.
  16-bit mode — low 16 bits are IP offset; high 16 bits are reserved.
  64-bit mode with REX.W — 64-bit IP offset.
  64-bit mode without REX.W — 32-bit IP offset.
  See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of the x87 FPU instruction pointer. |
| FCS            | x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H. |
The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

```
R7  R6  R5  R4  R3  R2  R1  R0
11   xx   xx   11   11   11   11
```

Here, 11B indicates empty stack elements and “xx” indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

```
R7  R6  R5  R4  R3  R2  R1  R0
0   1   1   1   0   0   0   0
```

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be used by an application program to pass a “clean” x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.
The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit floating-point data (assuming the stored data was not the contents of MMX technology registers) using Table 3-45.

<table>
<thead>
<tr>
<th>Exponent all 1's</th>
<th>Exponent all 0's</th>
<th>Fraction all 0's</th>
<th>J and M bits</th>
<th>FTW valid bit</th>
<th>x87 FTW</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0x</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>1</td>
<td>Zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1x</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>1</td>
<td>Special</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>Special</td>
</tr>
</tbody>
</table>

For all legal combinations above.

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

**IA-32e Mode Operation**

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-46), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-47), the FPU IP and FPU DP pointers are 32-bits.
### Table 3-46. Layout of the 64-Bit Mode FXSAVE64 Map (Requires REX.W = 1)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>FDP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIP</td>
<td>FOP</td>
<td>Reserved</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST0/MM0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST1/MM1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST2/MM2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST3/MM3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST4/MM4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST5/MM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST6/MM6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST7/MM7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMM15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 3-47. Layout of the 64-Bit Mode FXSAVE Map (REX.W = 0)

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reserved</td>
<td>FCS</td>
<td>FIP[31:0]</td>
<td>FOP</td>
<td>Reserved</td>
<td>FTW</td>
<td>FSW</td>
<td>FCW</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXCSR_MASK</td>
<td>MXCSR</td>
<td>Reserved</td>
<td>FDS</td>
<td>FDP[31:0]</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST0/MM0</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST1/MM1</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST2/MM2</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST3/MM3</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST4/MM4</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST5/MM5</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST6/MM6</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>ST7/MM7</td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM0</td>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM1</td>
<td>176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM2</td>
<td>192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM3</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM4</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM5</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM6</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM7</td>
<td>272</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM8</td>
<td>288</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM9</td>
<td>304</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM10</td>
<td>320</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM11</td>
<td>336</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM12</td>
<td>352</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM13</td>
<td>368</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM14</td>
<td>384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM15</td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>432</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>448</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>464</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>480</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>496</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Operation**

IF 64-Bit Mode

THEN

IF REX.W = 1

THEN

DEST := Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0,
MXCSR);

ELSE

DEST := Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);

FI;

ELSE

DEST := SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);

FI;

**Protected Mode Exceptions**

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the description of the alignment check exception [#AC] below.)

#SS(0) For an illegal address in the SS segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.

If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.

#UD If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

**Real-Address Mode Exceptions**

#GP If a memory operand is not aligned on a 16-byte boundary, regardless of segment.

If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If CR0.TS[bit 3] = 1.

If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.

If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

#PF(fault-code) For a page fault.

#AC For unaligned memory reference.

#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#PF(fault-code) For a page fault.

#NM If CR0.TS[bit 3] = 1.

If CR0.EM[bit 2] = 1.

#UD If CPUID.01H:EDX.FXSR[bit 24] = 0.

If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Implementation Note
The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B. This order vary for FXSAVE for different processor implementations.
FXTRACT—Extract Exponent and Significand

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F4</td>
<td>Valid</td>
<td>Valid</td>
<td>Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.</td>
</tr>
</tbody>
</table>

**Description**
Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand’s true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended \( \log_b(x) \) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of \(-\infty\) is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
\begin{align*}
\text{TEMP} & := \text{Significand}(\text{ST}(0)); \\
\text{ST}(0) & := \text{Exponent}(\text{ST}(0)); \\
\text{TOP} & := \text{TOP} - 1; \\
\text{ST}(0) & := \text{TEMP};
\end{align*}
\]

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow or overflow occurred.
#IA Source operand is an SNaN value or unsupported format.
#Z ST(0) operand is ±0.
#D Source operand is a denormal value.

**Protected Mode Exceptions**

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FYL2X—Compute y \times \log_2 x**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F1</td>
<td>FYL2X</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(1) with ((ST(1) \times \log_2 ST(0))) and pop the register stack.</td>
</tr>
</tbody>
</table>

**Description**

Computes \((ST(1) \times \log_2 (ST(0)))\), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-48. FYL2X Results**

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>−∞</th>
<th>−F</th>
<th>±0</th>
<th>+0&lt;+F&lt;+1</th>
<th>+1</th>
<th>+F &gt; +1</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>*</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>*</td>
<td>−∞</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−F</td>
<td>*</td>
<td>*</td>
<td>±0</td>
<td>±0</td>
<td>±0</td>
<td>−F</td>
<td>−∞</td>
<td>NaN</td>
</tr>
<tr>
<td>−0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>+0</td>
<td>−0</td>
<td>−0</td>
<td>−0</td>
<td>NaN</td>
</tr>
<tr>
<td>+0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>−0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>NaN</td>
</tr>
<tr>
<td>+F</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>−F</td>
<td>+0</td>
<td>+F</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>+∞</td>
<td>*</td>
<td>*</td>
<td>−∞</td>
<td>−∞</td>
<td>*</td>
<td>+∞</td>
<td>+∞</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.
* Indicates floating-point invalid-operation (#IA) exception.
** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ±0, the instruction returns ±0 with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base \(b\):

\[
\log_b x = \left(\log_2 b\right)^{-1} \times \log_2 x
\]

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[ST(1):= ST(1) \times \log_2 ST(0);\]

PopRegisterStack;

**FPU Flags Affected**

C1  
Set to 0 if stack underflow occurred.

Set if result was rounded up; cleared otherwise.

C0, C2, C3  
Undefined.
Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Either operand is an SNaN or unsupported format.
    Source operand in register ST(0) is a negative finite value
    (not -0).
#Z Source operand in register ST(0) is ±0.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**FYL2XP1—Compute \( y \times \log_2(x + 1) \)**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 F9</td>
<td>FYL2XP1</td>
<td>Valid</td>
<td>Valid</td>
<td>Replace ST(1) with ( ST(1) \times \log_2(ST(0) + 1.0) ) and pop the register stack.</td>
</tr>
</tbody>
</table>

**Description**

Computes \( (ST(1) \times \log_2(ST(0) + 1.0)) \), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

\[-(1 - \sqrt{2}/2) \to (1 - \sqrt{2}/2)\]

The source operand in ST(1) can range from \(-\infty\) to \(+\infty\). If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

**Table 3-49. FYL2XP1 Results**

<table>
<thead>
<tr>
<th>ST(0)</th>
<th>ST(1)</th>
<th>(-\infty \times (1 - \sqrt{2}/2)) to (-0)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+0) to ((1 - \sqrt{2}/2))</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>(+\infty)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>(-\infty)</td>
<td>NaN</td>
</tr>
<tr>
<td>(-F)</td>
<td>(+F)</td>
<td>(+0)</td>
<td>(-0)</td>
<td>(-F)</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>(-0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>(-0)</td>
<td>(-0)</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>(+0)</td>
<td>(-0)</td>
<td>(-0)</td>
<td>(+0)</td>
<td>(+0)</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>(+F)</td>
<td>(-F)</td>
<td>(-0)</td>
<td>(+0)</td>
<td>(+F)</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>(-\infty)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>(+\infty)</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**NOTES:**

F Means finite floating-point value.  
* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (\( \varepsilon \)) values, more significant digits can be retained by using the FYL2XP1 instruction than by using \((\varepsilon + 1)\) as an argument to the FYL2X instruction. The \((\varepsilon + 1)\) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where \( n \) is the logarithm base desired for the result of the FYL2XP1 instruction:

\[
\text{scale factor} := \log_n 2
\]

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

\[
ST(1) := ST(1) \times \log_2(ST(0) + 1.0);
\]

PopRegisterStack;
FPU Flags Affected
C1 Set to 0 if stack underflow occurred.
   Set if result was rounded up; cleared otherwise.
C0, C2, C3 Undefined.

Floating-Point Exceptions
#IS Stack underflow occurred.
#IA Either operand is an SNaN value or unsupported format.
#D Source operand is a denormal value.
#U Result is too small for destination format.
#O Result is too large for destination format.
#P Value cannot be represented exactly in destination format.

Protected Mode Exceptions
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF If there is a pending x87 FPU exception.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**GF2P8AFFINEINVQB—Galois Field Affine Transformation Inverse**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>GFNI</td>
<td>Computes inverse affine transformation in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX GFNI</td>
<td>Computes inverse affine transformation in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX GFNI</td>
<td>Computes inverse affine transformation in the finite field GF(2^8).</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1[k1]z, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL GFNI</td>
<td>Computes inverse affine transformation in the finite field GF(2^8).</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1[k1]z, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL GFNI</td>
<td>Computes inverse affine transformation in the finite field GF(2^8).</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1[k1]z, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F GFNI</td>
<td>Computes inverse affine transformation in the finite field GF(2^8).</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

The AFFINEINVQB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * inv(x) + b where "A" is an 8 by 8 bit matrix, and "x" and "b" are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial x^8 + x^4 + x^3 + x + 1.

One SIMD register (operand 1) holds "x" as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 "A" values, which are operated upon by the correspondingly aligned 8 "x" values in the first register. The "b" vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.
### Operation

```python
define affine_inverse_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        * inverse(x) is defined in the table above *
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
    return retbyte
```

### VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX Encoded Version)

KL, VL = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 := SRC2.qword[0]
    ELSE:
        tsrc2 := SRC2.qword[j]

FOR b := 0 to 7:
    IF k1[*B+b] OR *no writemask*:
        FOR i := 0 to 7:
            DEST.qword[j].byte[b] := affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] := 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
    DEST[MAX_VL-1:VL] := 0
VGF2P8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
    DEST[MAX_VL-1:VL] := 0

GF2P8AFFINEINVQB srcest, src1, imm8 (128b SSE Encoded Version)
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRDEST.qword[j].byte[b] := affine_inverse_byte(SRC1.qword[j], SRDEST.qword[j].byte[b], imm8)

Intel C/C++ Compiler Intrinsic Equivalent
(V)GF2P8AFFINEINVQB __m128i _mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEINVQB __m128i _mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGF2P8AFFINEINVQB __m256i _mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, __m256i, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_mask_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, __m256i, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m256i _mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m256i, __m256i, __m256i, __m256i, __m256i, int);
VGF2P8AFFINEINVQB __m512i _mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, __m512i, __m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_mask_gf2p8affineinv_epi64_epi8(__mmask128, __m512i, __m512i, __m512i, __m512i, __m512i, int);
VGF2P8AFFINEINVQB __m512i _mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask256, __m512i, __m512i, __m512i, __m512i, __m512i, int);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded: See Table 2-50, “Type E4NF Class Exception Conditions.”
**GF2P8AFFINEQB—Galois Field Affine Transformation**

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F3A CE /r /ib</td>
<td>GF2P8AFFINEQB xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>GFNI</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 CE /r</td>
<td>VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W1 CE /r</td>
<td>VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 CE /r</td>
<td>EVGF2P8AFFINEQB k1[x], xmm1, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 CE /r</td>
<td>EVGF2P8AFFINEQB k1[x], ymm1, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 CE /r</td>
<td>EVGF2P8AFFINEQB k1[x], zmm1, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

**Description**

The AFFINEB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by \( A \times x + b \) where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

**Operation**

```plaintext
define parity(x):
    t := 0  // single bit
    FOR i := 0 to 7:
        t = t xor x.bit[i]
    return t

define affine_byte(tsrc2qw, src1byte, imm):
    FOR i := 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] := parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte
```

---

GF2P8AFFINEQB—Galois Field Affine Transformation

Vol. 2A 3-479
VGF2P8AFFINEQB dest, src1, src2, imm8 (EVEX Encoded Version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR | j | := 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsr2 := SRC2.qword[0]
    ELSE:
        tsr2 := SRC2.qword[j]
    FOR b := 0 to 7:
        IF k1[*8+b] OR *no writemask*:
            DEST.qword[j].byte[b] := affine_byte(tsr2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] := 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
    DEST[MAX_VL-1:VL] := 0

VGF2P8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256)
FOR j := 0 TO KL-1:
    FOR b := 0 to 7:
        DEST.qword[j].byte[b] := affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
    DEST[MAX_VL-1:VL] := 0

GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE Encoded Version)
FOR j := 0 TO 1:
    FOR b := 0 to 7:
        SRCDEST.qword[j].byte[b] := affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)

Intel C/C++ Compiler Intrinsic Equivalent
(V)GF2P8AFFINEQB __m128i _mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
(V)GF2P8AFFINEQB __m128i _mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);
VGFP8AFFINEQB __m256i _mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);
VGFP8AFFINEQB __m256i _mm256_mask_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGFP8AFFINEQB __m256i _mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
VGFP8AFFINEQB __m512i _mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);
VGFP8AFFINEQB __m512i _mm512_mask_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);
VGFP8AFFINEQB __m512i _mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Legacy-encoded and VEX-encoded: See Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded: See Table 2-50, "Type E4NF Class Exception Conditions."
GF2P8MULB—Galois Field Multiply Bytes

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 CF /r</td>
<td>A</td>
<td>V/V</td>
<td>GFNI</td>
<td>Multiplies elements in the finite field GF(2^8).</td>
</tr>
<tr>
<td>GF2P8MULB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 CF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiplies elements in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VGF2P8MULB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 CF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiplies elements in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VGF2P8MULB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 CF /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiplies elements in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VGF2P8MULB xmm1[1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td>GFNI</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 CF /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiplies elements in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VGF2P8MULB ymm1[1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td>GFNI</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 CF /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiplies elements in the finite field GF(2^8).</td>
</tr>
<tr>
<td>VGF2P8MULB zmm1[1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td>GFNI</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The instruction multiplies elements in the finite field GF(2^8), operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field GF(2^8) is represented in polynomial representation with the reduction polynomial x^8 + x^4 + x^3 + x + 1.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

### Operation

```plaintext
define gf2p8mul_byte(src1byte, src2byte):
  tword := 0
  FOR i := 0 to 7:
    IF src2byte.bit[i]:
      tword := tword XOR (src1byte<< i)
      * carry out polynomial reduction by the characteristic polynomial p*
  FOR i := 14 downto 8:
    p := 0x11B << (i-8)  *0x11B = 0000_0001_0011 in binary*
    IF tword.bit[i]:
      tword := tword XOR p
  return tword.byte[0]
```

GF2P8MULB—Galois Field Multiply Bytes
**GF2P8MULB** dest, src1, src2 (EVEX Encoded Version)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
   ELSE IF *zeroing*:
      DEST.byte[j] := 0
   ELSE DEST.byte[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VGF2P8MULB** dest, src1, src2 (128b and 256b VEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256)

FOR j := 0 TO KL-1:
   DEST.byte[j] := gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])

DEST[MAX_VL-1:VL] := 0

**GF2P8MULB** srcdest, src1 (128b SSE Encoded Version)

FOR j := 0 TO 15:
   SRCDEST.byte[j] := gf2p8mul_byte(SRCDEST.byte[j], SRC1.byte[j])

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)GF2P8MULB __m128i _mm_gf2p8mul_epi8(__m128i, __m128i);
(V)GF2P8MULB __m128i _mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);
(V)GF2P8MULB __m128i _mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);
VGFP8MULB __m256i __mm256_gf2p8mul_epi8(__m256i, __m256i);
VGFP8MULB __m256i __mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);
VGFP8MULB __m256i __mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);
VGFP8MULB __m512i __mm512_gf2p8mul_epi8(__m512i, __m512i);
VGFP8MULB __m512i __mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);
VGFP8MULB __m512i __mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Legacy-encoded and VEX-encoded: See Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded: See Table 2-49, “Type E4 Class Exception Conditions.”
HADDPD—Packed Double Precision Floating-Point Horizontal Add

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 7C /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 7C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 7C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Adds the double precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-17 for HADDPD; see Figure 3-18 for VHADDPD.

![Figure 3-17. HADDPD—Packed Double Precision Floating-Point Horizontal Add](OM15993)
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

**HADDPD (128-bit Legacy SSE Version)**
DEST[63:0] := SRC1[127:64] + SRC1[63:0]
DEST[MAXVL-1:128] (Unmodified)

**VHADDPD (VEX.128 Encoded Version)**
DEST[63:0] := SRC1[127:64] + SRC1[63:0]
DEST[MAXVL-1:128] := 0

**VHADDPD (VEX.256 Encoded Version)**
DEST[63:0] := SRC1[127:64] + SRC1[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VHADDPD __m256d _mm256_hadd_pd (__m256d a, __m256d b);
HADDPD __m128d _mm_hadd_pd (__m128d a, __m128d b);

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.
Other Exceptions
See Table 2-19, "Type 2 Class Exception Conditions."
HADDPS—Packed Single Precision Floating-Point Horizontal Add

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 7C /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal add packed single precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>HADDPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128:F2.0F.WIG 7C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal add packed single precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VHADDPS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256:F2.0F.WIG 7C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal add packed single precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>VHADDPS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Adds the single precision floating-point values in the first and second dword of the destination operand and stores the result in the first dword of the destination operand.

Adds single precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).
See Figure 3-19 for HADDPS; see Figure 3-20 for VHADDPS.

**Figure 3-19. HADDPS—Packed Single Precision Floating-Point Horizontal Add**

**Figure 3-20. VHADDPS Operation**

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
**Operation**

**HADDPS (128-bit Legacy SSE Version)**

\[
\text{DEST}[31:0] := \text{SRC1}[63:32] + \text{SRC1}[31:0]
\]

\[
\text{DEST}[63:32] := \text{SRC1}[127:96] + \text{SRC1}[95:64]
\]

\[
\text{DEST}[95:64] := \text{SRC2}[63:32] + \text{SRC2}[31:0]
\]

\[
\text{DEST}[127:96] := \text{SRC2}[127:96] + \text{SRC2}[95:64]
\]

\[
\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}
\]

**VHADDPS (VEX.128 Encoded Version)**

\[
\text{DEST}[31:0] := \text{SRC1}[63:32] + \text{SRC1}[31:0]
\]

\[
\text{DEST}[63:32] := \text{SRC1}[127:96] + \text{SRC1}[95:64]
\]

\[
\text{DEST}[95:64] := \text{SRC2}[63:32] + \text{SRC2}[31:0]
\]

\[
\text{DEST}[127:96] := \text{SRC2}[127:96] + \text{SRC2}[95:64]
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

**VHADDPS (VEX.256 Encoded Version)**

\[
\text{DEST}[31:0] := \text{SRC1}[63:32] + \text{SRC1}[31:0]
\]

\[
\text{DEST}[63:32] := \text{SRC1}[127:96] + \text{SRC1}[95:64]
\]

\[
\text{DEST}[95:64] := \text{SRC2}[63:32] + \text{SRC2}[31:0]
\]

\[
\text{DEST}[127:96] := \text{SRC2}[127:96] + \text{SRC2}[95:64]
\]

\[
\text{DEST}[159:128] := \text{SRC1}[191:160] + \text{SRC1}[159:128]
\]

\[
\text{DEST}[191:160] := \text{SRC1}[255:224] + \text{SRC1}[223:192]
\]

\[
\]

\[
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{HADDPS} \quad \text{__m128} \; \text{__mm_hadd_ps} \; (\text{__m128} \; a, \text{__m128} \; b);
\]

\[
\text{VHADDPS} \quad \text{__m256} \; \text{__mm256_hadd_ps} \; (\text{__m256} \; a, \text{__m256} \; b);
\]

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Table 2-19, “Type 2 Class Exception Conditions.”
HLT—Halt

### Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>HLT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Halt</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Enter Halt state;

### Flags Affected

None.

### Protected Mode Exceptions

- **#GP(0)**: If the current privilege level is not 0.
- **#UD**: If the LOCK prefix is used.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.
HRESET—History Reset

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 3A F0 C0 /ib</td>
<td>HRESET imm8, &lt;EAX&gt;</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Requests the processor to selectively reset selected components of hardware history maintained by the current logical processor. HRESET operation is controlled by the implicit EAX operand. The value of the explicit imm8 operand is ignored. This instruction can only be executed at privilege level 0.

The HRESET instruction can be used to request reset of multiple components of hardware history. Prior to the execution of HRESET, the system software must take the following steps:

1. Enumerate the HRESET capabilities via CPUID.20H.0H:EBX, which indicates what components of hardware history can be reset.
2. Only the bits enumerated by CPUID.20H.0H:EBX can be set in the IA32_HRESET_ENABLE MSR. HRESET causes a general-protection exception (#GP) if EAX sets any bits that are not set in the IA32_HRESET_ENABLE MSR.

Any attempt to execute the HRESET instruction inside a transactional region will result in a transaction abort.

**Operation**

IF EAX = 0
THEN NOP
ELSE
    FOREACH i such that EAX[i] = 1
        Reset prediction history for feature i
FI

**Flags Affected**
None.

**Protected Mode Exceptions**

- #GP(0) If CPL > 0 or (EAX AND NOT IA32_HRESET_ENABLE) ≠ 0.
- #UD If CPUID.07H.01H:EAX.HRESET[bit 22] = 0.

**Real-Address Mode Exceptions**
Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

- #GP(0) HRESET instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 7D /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal subtract packed double precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG 7D /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed double precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F:WIG 7D /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed double precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>ModRM:reg (r, w)</th>
<th>ModRMr/m (r)</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The HSUBPD instruction subtracts horizontally the packed double precision floating-point numbers of both operands.

Subtracts the double precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-21 for HSUBPD; see Figure 3-22 for VHSUBPD.
HSUBPD—Packed Double Precision Floating-Point Horizontal Subtract

Figure 3-22. VHSUBPD operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

HSUBPD (128-bit Legacy SSE Version)
DEST[63:0] := SRC1[63:0] - SRC1[127:64]
DEST[MAXVL-1:128] (Unmodified)

VHSUBPD (VEX.128 Encoded Version)
DEST[63:0] := SRC1[63:0] - SRC1[127:64]
DEST[MAXVL-1:128] := 0

VHSUBPD (VEX.256 Encoded Version)
DEST[63:0] := SRC1[63:0] - SRC1[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
HSUBPD __m128d _mm_hsub_pd(__m128d a, __m128d b)
VHSUBPD __m256d _mm256_hsub_pd (__m256d a, __m256d b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.
Other Exceptions
See Table 2-19, "Type 2 Class Exception Conditions."
HSUBPS—Packed Single Precision Floating-Point Horizontal Subtract

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 7D /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Horizontal subtract packed single precision floating-point values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>HSUBPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.F2.0F.W1 7D /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed single precision floating-point values from xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VSUBPS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.F2.0F.W1 7D /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Horizontal subtract packed single precision floating-point values from ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>VSUBPS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Subtracts the single precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-23 for HSUBPS; see Figure 3-24 for VHSUBPS.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
Operation

**HSUBPS (128-bit Legacy SSE Version)**

\[ \begin{align*}
\text{DEST}[31:0] & := \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] & := \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] & := \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] & := \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & := 0
\end{align*} \]

**VHSUBPS (VEX.128 Encoded Version)**

\[ \begin{align*}
\text{DEST}[31:0] & := \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] & := \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] & := \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] & := \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\text{DEST}[\text{MAXVL}-1:128] & := 0
\end{align*} \]

**VHSUBPS (VEX.256 Encoded Version)**

\[ \begin{align*}
\text{DEST}[31:0] & := \text{SRC1}[31:0] - \text{SRC1}[63:32] \\
\text{DEST}[63:32] & := \text{SRC1}[95:64] - \text{SRC1}[127:96] \\
\text{DEST}[95:64] & := \text{SRC2}[31:0] - \text{SRC2}[63:32] \\
\text{DEST}[127:96] & := \text{SRC2}[95:64] - \text{SRC2}[127:96] \\
\end{align*} \]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
\text{HSUBPS} & \quad \text{__m128 } \text{ _mm_hsub_ps}(\text{__m128 a, __m128 b}); \\
\text{VHSUBPS} & \quad \text{__m256 } \text{ _mm256_hsub_ps (__m256 a, __m256 b});
\end{align*}
\]

**Exceptions**

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Table 2-19, “Type 2 Class Exception Conditions.”
IDIV—Signed Divide

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /7</td>
<td>IDIV r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide AX by r/m8, with result stored in: AL := Quotient, AH := Remainder.</td>
</tr>
<tr>
<td>REX + F6 /7</td>
<td>IDIV r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide AX by r/m8, with result stored in AL := Quotient, AH := Remainder.</td>
</tr>
<tr>
<td>F7 /7</td>
<td>IDIV r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide DX:AX by r/m16, with result stored in AX := Quotient, DX := Remainder.</td>
</tr>
<tr>
<td>F7 /7</td>
<td>IDIV r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Signed divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder.</td>
</tr>
<tr>
<td>REX.W + F7 /7</td>
<td>IDIV r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Signed divide RDX:RAX by r/m64, with result stored in RAX := Quotient, RDX := Remainder.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-51.

Table 3-51. IDIV Results

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Dividend</th>
<th>Divisor</th>
<th>Quotient</th>
<th>Remainder</th>
<th>Quotient Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word/byte</td>
<td>AX</td>
<td>r/m8</td>
<td>AL</td>
<td>AH</td>
<td>−128 to +127</td>
</tr>
<tr>
<td>Doubleword/word</td>
<td>DX:AX</td>
<td>r/m16</td>
<td>AX</td>
<td>DX</td>
<td>−32,768 to +32,767</td>
</tr>
<tr>
<td>Quadword/doubleword</td>
<td>EDX:EAX</td>
<td>r/m32</td>
<td>EAX</td>
<td>EDX</td>
<td>−231 to 231 − 1</td>
</tr>
<tr>
<td>Doublequadword/quadword</td>
<td>RDX:RAX</td>
<td>r/m64</td>
<td>RAX</td>
<td>RDX</td>
<td>−263 to 263 − 1</td>
</tr>
</tbody>
</table>
Operation

IF SRC = 0
    THEN #DE; (* Divide error *)
    Fl;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp := AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
            (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL := temp;
                AH := AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp := DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
                (* If a positive result is greater than 7FFFH or a negative result is less than 8000H *)
                THEN #DE; (* Divide error *)
                ELSE
                    AX := temp;
                    DX := DX:AX SignedModulus SRC;
            FI;
        ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
            temp := EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFH) or (temp < 80000000H)
                (* If a positive result is greater than 7FFFFFFFFFH or a negative result is less than 80000000H *)
                THEN #DE; (* Divide error *)
                ELSE
                    EAX := temp;
                    EDX := EDX:EAX SignedModulus SRC;
            FI;
        ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
            temp := RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
                (* If a positive result is greater than 7FFFFFFFFFFFFFFFH or a negative result is less than 8000000000000000H *)
                THEN #DE; (* Divide error *)
                ELSE
                    RAX := temp;
                    RDX := RDX:RAX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp := RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
                (* If a positive result is greater than 7FFFFFFFFFFFFFFFH or a negative result is less than 8000000000000000H *)
                THEN #DE; (* Divide error *)
                ELSE
                    RAX := temp;
                    RDX := RDX:RAX SignedModulus SRC;
            FI;
        FI;
    FI;

FI;
Flags Affected
The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions
#DE If the source operand (divisor) is 0.
The signed result (quotient) is too large for the destination.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#DE If the source operand (divisor) is 0.
The signed result (quotient) is too large for the destination.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#DE If the source operand (divisor) is 0.
The signed result (quotient) is too large for the destination.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#DE If the source operand (divisor) is 0
If the quotient is too large for the designated register.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
IMUL—Signed Multiply

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /5</td>
<td>IMUL r/m8¹</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>AX: AL * r/m byte.</td>
</tr>
<tr>
<td>F7 /5</td>
<td>IMUL r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>DX:AX := AX * r/m word.</td>
</tr>
<tr>
<td>F7 /5</td>
<td>IMUL r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>EDX:EAX := EAX * r/m32.</td>
</tr>
<tr>
<td>REX.W + F7 /5</td>
<td>IMUL r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>RDX:RAX := RAX * r/m64.</td>
</tr>
<tr>
<td>0F AF /r</td>
<td>IMUL r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>word register := word register * r/m16.</td>
</tr>
<tr>
<td>0F AF /r</td>
<td>IMUL r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>doubleword register := doubleword register * r/m32.</td>
</tr>
<tr>
<td>REX.W + 0F AF /r</td>
<td>IMUL r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Quadword register := Quadword register * r/m64.</td>
</tr>
<tr>
<td>6B /r ib</td>
<td>IMUL r16, r/m16, imm8</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>word register := r/m16 * sign-extended immediate byte.</td>
</tr>
<tr>
<td>6B /r ib</td>
<td>IMUL r32, r/m32, imm8</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>doubleword register := r/m32 * sign-extended immediate byte.</td>
</tr>
<tr>
<td>REX.W + 6B /r ib</td>
<td>IMUL r64, r/m64, imm8</td>
<td>RMI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Quadword register := r/m64 * sign-extended immediate byte.</td>
</tr>
<tr>
<td>69 /r iw</td>
<td>IMUL r16, r/m16, imm16</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>word register := r/m16 * immediate word.</td>
</tr>
<tr>
<td>69 /r id</td>
<td>IMUL r32, r/m32, imm32</td>
<td>RMI</td>
<td>Valid</td>
<td>Valid</td>
<td>doubleword register := r/m32 * immediate doubleword.</td>
</tr>
<tr>
<td>REX.W + 69 /r id</td>
<td>IMUL r64, r/m64, imm32</td>
<td>RMI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Quadword register := r/m64 * immediate doubleword.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8/16/32</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.

- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.

- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).
When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three-operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** — The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.

- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.

- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

**Operation**

IF (NumberOfOperands = 1)
   THEN IF (OperandSize = 8)
       THEN
           TMP_XP := AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
           AX := TMP_XP[15:0];
           IF SignExtend(TMP_XP[7:0]) = TMP_XP
               THEN CF := 0; OF := 0;
           ELSE CF := 1; OF := 1; FI;
       ELSE IF OperandSize = 16
           THEN
               TMP_XP := AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
               DX:AX := TMP_XP[31:0];
               IF SignExtend(TMP_XP[15:0]) = TMP_XP
                   THEN CF := 0; OF := 0;
               ELSE CF := 1; OF := 1; FI;
           ELSE IF OperandSize = 32
               THEN
                   TMP_XP := EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
                   EDX:EAX := TMP_XP[63:0];
                   IF SignExtend(TMP_XP[31:0]) = TMP_XP
                       THEN CF := 0; OF := 0;
                   ELSE CF := 1; OF := 1; FI;
               ELSE (* OperandSize = 64 *)
                   TMP_XP := RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                   EDX:EAX := TMP_XP[127:0];
                   IF SignExtend(TMP_XP[63:0]) = TMP_XP
                       THEN CF := 0; OF := 0;
                   ELSE CF := 1; OF := 1; FI;
               FI;
           FI;
   ELSE IF OperandSize = 64
       THEN
           TMP_XP := EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
           EDX:EAX := TMP_XP[127:0];
           IF SignExtend(TMP_XP[63:0]) = TMP_XP
               THEN CF := 0; OF := 0;
           ELSE CF := 1; OF := 1; FI;

   FI;
ELSE IF (NumberOfOperands = 2)
    THEN
        TMP_XP := DEST + SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DEST := TruncateToOperandSize(TMP_XP);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF := 1; OF := 1;
                ELSE CF := 0; OF := 0; FI;
        ELSE (* NumberOfOperands = 3 *)
            TMP_XP := SRC1 + SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
            DEST := TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF := 1; OF := 1;
                    ELSE CF := 0; OF := 0; FI;
            FI;
        FI;

Flags Affected
For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register is used to access memory and it contains a NULL NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
IN—Input From Port

### Description
Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation
```c
IF (PE = 1) and ((CPL > IOPL) or (VM = 1))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST := SRC; (* Read from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST := SRC; (* Read from selected I/O port *)
FI;
```

### Flags Affected
None.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compa/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E4 ib</td>
<td>IN AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Input byte from imm8 I/O port address into AL.</td>
</tr>
<tr>
<td>E5 ib</td>
<td>IN AX, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Input word from imm8 I/O port address into AX.</td>
</tr>
<tr>
<td>E5 ib</td>
<td>IN EAX, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Input dword from imm8 I/O port address into EAX.</td>
</tr>
<tr>
<td>EC</td>
<td>IN AL,DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input byte from I/O port in DX into AL.</td>
</tr>
<tr>
<td>ED</td>
<td>IN AX,DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input word from I/O port in DX into AX.</td>
</tr>
<tr>
<td>ED</td>
<td>IN EAX,DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input doubleword from I/O port in DX into EAX.</td>
</tr>
</tbody>
</table>
Protected Mode Exceptions

#GP(0)  If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

#PF(fault-code)  If a page fault occurs.

#UD  If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD  If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)  If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

#PF(fault-code)  If a page fault occurs.

#UD  If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

#PF(fault-code)  If a page fault occurs.

#UD  If the LOCK prefix is used.
INC—Increment by 1

**Opcode** | **Instruction** | **Op/En** | **64-Bit Mode** | **Compat/Leg Mode** | **Description**
---|---|---|---|---|---
FE /0 | INC r/m8 | M | Valid | Valid | Increment r/m byte by 1.
REX + FE /0 | INC r/m81 | M | Valid | N.E. | Increment r/m byte by 1.
FF /0 | INC r/m16 | M | Valid | Valid | Increment r/m word by 1.
FF /0 | INC r/m32 | M | Valid | Valid | Increment r/m doubleword by 1.
REX.W + FF /0 | INC r/m64 | M | Valid | N.E. | Increment r/m quadword by 1.
40+ rw2 | INC r16 | O | N.E. | Valid | Increment word register by 1.
40+ rd | INC r32 | O | N.E. | Valid | Increment doubleword register by 1.

**NOTES:**
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
2. 40H through 47H are REX prefixes in 64-bit mode.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/w (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>O</td>
<td>opcode + rd (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction’s 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

**Operation**

DEST := DEST + 1;

**Flags Affected**

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

**Protected Mode Exceptions**

- **#GP(0)**: If the destination operand is located in a non-writable segment.
- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#GP(0)**: If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used but the destination is not a memory operand.
Real-Address Mode Exceptions

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
INCSSPD/INCSSPQ—Increment Shadow Stack Pointer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F AE /05 INCSSPD r32</td>
<td>R</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Increment SSP by 4 * r32[7:0].</td>
</tr>
<tr>
<td>F3 REX.W 0F AE /05 INCSSPQ r64</td>
<td>R</td>
<td>V/N.E.</td>
<td>CET_SS</td>
<td>Increment SSP by 8 * r64[7:0].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>N/A</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction can be used to increment the current shadow stack pointer by the operand size of the instruction times the unsigned 8-bit value specified by bits 7:0 in the source operand. The instruction performs a pop and discard of the first and last element on the shadow stack in the range specified by the unsigned 8-bit value in bits 7:0 of the source operand.

**Operation**

IF CPL = 3
  IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
    THEN #UD; Fl;
  ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
      THEN #UD; Fl;
    FI;
  FI;

IF (operand size is 64-bit)
  THEN
    Range := R64[7:0];
    shadow_stack_load 8 bytes from SSP;
    IF Range > 0
      THEN shadow_stack_load 8 bytes from SSP + 8 * (Range - 1);
      FI;
    SSP := SSP + Range * 8;
  ELSE
    Range := R32[7:0];
    shadow_stack_load 4 bytes from SSP;
    IF Range > 0
      THEN shadow_stack_load 4 bytes from SSP + 4 * (Range - 1);
      FI;
    SSP := SSP + Range * 4;
  FI;

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

INCSSPD void _incsspd(int);
INCSSPQ void _incsspq(int);
Protected Mode Exceptions
#UD If the LOCK prefix is used.
  If CR4.CET = 0.
  IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
  IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#PF(fault-code) If a page fault occurs.

Real-Address Mode Exceptions
#UD The INCSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The INCSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
INS/INSB/INSW/INSD—Input from Port to String

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6C</td>
<td>INS m8, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INS m16, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INS m32, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6C</td>
<td>INSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
<tr>
<td>6D</td>
<td>INSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.¹</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be "DX," and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.
The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

These instructions may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

**Operation**

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
   THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
      IF (Any I/O Permission Bit for I/O port being accessed = 1)
         THEN (* I/O operation is not allowed *)
            #GP(0);
         ELSE (* I/O operation is allowed *)
            DEST := SRC; (* Read from I/O port *)
            FI;
      ELSE (Real Mode or Protected Mode with CPL IOPL *)
         DEST := SRC; (* Read from I/O port *)
         FI;
   FI;

Non-64-bit Mode:

IF (Byte transfer)
   THEN IF DF = 0
      THEN (E)DI := (E)DI + 1;
      ELSE (E)DI := (E)DI - 1; FI;
   ELSE IF (Word transfer)
      THEN IF DF = 0
         THEN (E)DI := (E)DI + 2;
         ELSE (E)DI := (E)DI - 2; FI;
      ELSE (* Doubleword transfer *)
         THEN IF DF = 0
            THEN (E)DI := (E)DI + 4;
            ELSE (E)DI := (E)DI - 4; FI;
         FI;
   FI;

FI64-bit Mode:

IF (Byte transfer)
   THEN IF DF = 0
      THEN (E|R)DI := (E|R)DI + 1;
      ELSE (E|R)DI := (E|R)DI - 1; FI;
   ELSE IF (Word transfer)
      THEN IF DF = 0
         THEN (E)DI := (E)DI + 2;
         ELSE (E)DI := (E)DI - 2; FI;
      ELSE (* Doubleword transfer *)
         THEN IF DF = 0
            THEN (E)DI := (E)DI + 4;
            ELSE (E)DI := (E)DI - 4; FI;
   FI;
ELSE (E|R)DI := (E|R)DI - 4; FI;

FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
If the destination is located in a non-writable segment.
If an illegal memory operand effective address in the ES segments is given.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
If the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
INSERTPS—Insert Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 21/r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Insert a single precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert a single precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

(register source form)
Copy a single precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

(memory source form)
Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

If VININSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.
Operation

VINSERTPS (VEX.128 and EVEX Encoded Version)

IF (SRC = REG) THEN COUNT_S := imm8[7:6]
ELSE COUNT_S := 0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
  0: TMP := SRC2[31:0]
  1: TMP := SRC2[63:32]
  2: TMP := SRC2[95:64]
  3: TMP := SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
  0: TMP2[31:0] := TMP
  1: TMP2[63:32] := TMP
  2: TMP2[95:64] := TMP
  3: TMP2[127:96] := TMP
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
ELSE DEST[31:0] := TMP2[31:0]
ELSE DEST[95:64] := TMP2[95:64]
DEST[MAXVL-1:128] := 0

INSERTPS (128-bit Legacy SSE Version)

IF (SRC = REG) THEN COUNT_S := imm8[7:6]
ELSE COUNT_S := 0
COUNT_D := imm8[5:4]
ZMASK := imm8[3:0]
CASE (COUNT_S) OF
  0: TMP := SRC[31:0]
  1: TMP := SRC[63:32]
  2: TMP := SRC[95:64]
  3: TMP := SRC[127:96]
ESAC;
CASE (COUNT_D) OF
  0: TMP2[31:0] := TMP
  2: TMP2[95:64] := TMP
  3: TMP2[127:96] := TMP
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
ELSE DEST[31:0] := TMP2[31:0]
ELSE DEST[95:64] := TMP2[95:64]
DEST[MAXVL-1:128] := 0

INSERTPS—Insert Scalar Single Precision Floating-Point Value
TMP2[63:0] := DEST[63:0]
3: TMP2[127:96] := TMP
    TMP2[95:0] := DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] := 00000000H
    ELSE DEST[31:0] := TMP2[31:0]
    ELSE DEST[95:64] := TMP2[95:64]
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VINERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:
#UD If VEX.L = 0.
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
INT n/INTO/INT3/INT1—Call to Interrupt Procedure

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>INT3</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate breakpoint trap.</td>
</tr>
<tr>
<td>CD ib</td>
<td>INT imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate software interrupt with vector specified by immediate byte.</td>
</tr>
<tr>
<td>CE</td>
<td>INTO</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Generate overflow trap if overflow flag is 1.</td>
</tr>
<tr>
<td>F1</td>
<td>INT1</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate debug trap.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>I</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The INT n instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT n instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT3 instruction uses a one-byte opcode (CC) and is intended for calling the debug exception handler with a breakpoint exception (#BP). (This one-byte form is useful because it can replace the first byte of any instruction at which a breakpoint is desired, including other one-byte instructions, without overwriting other instructions.)

The INT1 instruction also uses a one-byte opcode (F1) and generates a debug exception (#DB) without setting any bits in DR6. \(^1\) Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

An interrupt generated by the INTO, INT3, or INT1 instruction differs from one generated by INT n in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The action of the INT n instruction (including the INTO, INT3, and INT1 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT n instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

Each of the INT n, INTO, and INT3 instructions generates a general-protection exception (#GP) if the CPL is greater than the DPL value in the selected gate descriptor in the IDT. In contrast, the INT1 instruction can deliver a #DB

---

1. The mnemonic ICEBP has also been used for the instruction with opcode F1.
The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the interrupt vector table, and its pointers are called interrupt vectors.) The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

### Table 3-52. Decision Table

<table>
<thead>
<tr>
<th>PE</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IOPL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&lt;3</td>
<td>=3</td>
</tr>
<tr>
<td>DPL/CPL RELATIONSHIP</td>
<td>DPL&lt; CPL</td>
<td>DPL&gt; CPL</td>
<td>DPL= CPL or C</td>
<td>DPL= CPL &amp; NC</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERRUPT TYPE</td>
<td>S/W</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>GATE TYPE</td>
<td>-</td>
<td>-</td>
<td>Task</td>
<td>Trap or Interrupt</td>
<td>Trap or Interrupt</td>
<td>Trap or Interrupt</td>
<td>Trap or Interrupt</td>
<td>Trap or Interrupt</td>
</tr>
<tr>
<td>REAL-ADDRESS-MODE</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROTECTED-MODE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TRAP-OR-INTERRUPT-GATE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTER-PRIVILEGE-LEVEL-INTERRUPT</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTRA-PRIVILEGE-LEVEL-INTERRUPT</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERRUPT-FROM-VIRTUAL-8086-MODE</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASK-GATE</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#GP</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:
- Don’t Care.
- Y Yes, action taken.
- Blank Action not taken.
- S/W Applies to INT n, INT3, and INTO, but not to INT1.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate’s DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 17, “Control-flow Enforcement Technology (CET)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for CET details.
Instruction ordering. Instructions following an INT n may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the INT n have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible). This applies also to the INTO, INT3, and INT1 instructions, but not to executions of INTO when EFLAGS.OF = 0.

Operation
The following operational description applies not only to the INT n, INTO, INT3, or INT1 instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.
The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.
The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction error_code(num,idt,ext), where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is (num & FCH) | ext; (2) if idt is 1, the error code is (num < 3) | 2 | ext.
In many cases, the pseudofunction error_code is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt (INT n, INT3, or INTO), EXT is 0; otherwise (including INT1), EXT is 1.

IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
            THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
            ELSE
                IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
                    THEN
                        Consult bit n of the software interrupt redirection bit map in the TSS;
                        IF bit n is clear
                            THEN (* redirect interrupt to 8086 program interrupt handler *)
                                Push EFLAGS[15:0]; (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                                Push CS;
                                Push IP;
                                IF IOPL = 3
                                    THEN IF := 0; (* Clear interrupt flag *)
                                    ELSE VIF := 0; (* Clear virtual interrupt flag *)
                                FI;
                                TF := 0; (* Clear trap flag *)
                                load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
                            ELSE
                                IF IOPL = 3
                                    THEN GOTO PROTECTED-MODE;
                                    ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
                                FI;
                            FI;
                    ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                        IF (IA32_EFER.LMA = 0)
                            THEN (* Protected mode, or virtual-8086 mode interrupt *)
                                GOTO PROTECTED-MODE;
                            ELSE (* IA-32e mode interrupt *)
                                GOTO IA-32e-MODE;
                        ELSE
                            (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                            GOTO IA-32e-MODE;
                    FI;
                ELSE
                    IF IOPL = 3
                        THEN #GP(0); (* Bit 0 of error code is 0 because INT n *)
                    ELSE
                        GOTO IA-32e-MODE;
                    FI;
                FI;
            ELSE
                (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                GOTO IA-32e-MODE;
        FI;
    ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
        GOTO IA-32e-MODE;
    FI;
ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
    GOTO IA-32e-MODE;
}
INT n/INTO/INT3/INT1—Call to Interrupt Procedure

FOR REAL-ADDRESS-MODE:

IF ((vector_number « 2) + 3) is not within IDT limit
THEN #GP; FI;
IF stack not large enough for a 6-byte return information
THEN #SS; FI;
Push (EFLAGS[15:0]);
IF := 0; (* Clear interrupt flag *)
TF := 0; (* Clear trap flag *)
AC := 0; (* Clear AC flag *)
Push(CS);
Push(IP);
(* No error codes are pushed in real-address mode*)
CS := IDT(Descriptor (vector_number « 2), selector));
EIP := IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

FOR PROTECTED-MODE:

IF ((vector_number « 3) + 7) is not within IDT limits
or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
THEN #GP(error_code(vector_number,1,EXT)); FI;
(* idt operand to error_code set because vector is used *)
IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
THEN
   IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
      THEN #GP(error_code(vector_number,1,0)); FI;
      (* idt operand to error_code set because vector is used *)
      (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
   FI;
IF gate not present
THEN #NP(error_code(vector_number,1,EXT)); FI;
(* idt operand to error_code set because vector is used *)
IF task gate (* Specified in the selected interrupt table descriptor *)
THEN GOTO TASK-GATE;
ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
FI;
END;

FOR IA-32e-MODE:

IF INTO and CS.L = 1 (64-bit mode)
THEN #UD;
FI;
IF ((vector_number « 4) + 15) is not in IDT limits
or selected IDT descriptor is not an interrupt-, or trap-gate type
THEN #GP(error_code(vector_number,1,EXT)); FI;
(* idt operand to error_code set because vector is used *)
IF software interrupt (* Generated by INT n, INT3, or INTO; does not apply to INT1 *)
THEN
   IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
      THEN #GP(error_code(vector_number,1,0)); FI;
      (* idt operand to error_code set because vector is used *)
      (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
   FI;
FI;
IF gate not present
THEN #NP(error_code(vector_number,1,EXT));
(* idt operand to error_code set because vector is used *)
FI;
GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;

TASK-GATE: (* PE = 1, task gate *)
Read TSS selector in task gate (IDT descriptor):
   IF local/global bit is set to local or index not within GDT limits
      THEN #GP(error_code(TSS selector,0,EXT)); FI;
   (* idt operand to error_code is 0 because selector is used *)
Access TSS descriptor in GDT;
   IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
      THEN #GP(error_code(TSS selector,0,EXT)); FI;
   (* idt operand to error_code is 0 because selector is used *)
   IF TSS not present
      THEN #NP(error_code(TSS selector,0,EXT)); FI;
   (* idt operand to error_code is 0 because selector is used *)
SWITCH-TASKS (with nesting) to TSS;
IF interrupt caused by fault with error code
   THEN
      IF stack limit does not allow push of error code
         THEN #SS(EXT); FI;
      Push(error code);
   FI;
IF EIP not within code segment limit
   THEN #GP(EXT); FI;
END;

TRAP-OR-INTERRUPT-GATE:
Read new code-segment selector for trap or interrupt gate (IDT descriptor);
IF new code-segment selector is NULL
   THEN #GP(EXT); FI; (* Error code contains NULL selector *)
IF new code-segment selector is not within its descriptor table limits
   THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
   (* idt operand to error_code is 0 because selector is used *)
Read descriptor referenced by new code-segment selector;
   IF descriptor does not indicate a code segment or new code-segment DPL > CPL
      THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
   (* idt operand to error_code is 0 because selector is used *)
   IF new code-segment descriptor is not present,
      THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
   (* idt operand to error_code is 0 because selector is used *)
   IF new code segment is non-conforming with DPL < CPL
      THEN
         IF VM = 0
            THEN
               GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
               (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment, DPL < CPL *)
            ELSE (* VM = 1 *)
               IF new code-segment DPL ≠ 0
                  THEN #GP(error_code(new code-segment selector,0,EXT));
INT n/INTO/INT3/INT1—Call to Interrupt Procedure

INSTRUCTION SET REFERENCE, A-L

3-522 Vol. 2A

(* idt operand to error_code is 0 because selector is used *)
GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
(* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
FI;
ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
IF VM = 1
THEN #GP(error_code(new code-segment selector,0,EXT));
(* idt operand to error_code is 0 because selector is used *)
IF new code segment is conforming or new code-segment DPL = CPL
THEN
GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
#GP(error_code(new code-segment selector,0,EXT));
(* idt operand to error_code is 0 because selector is used *)
FI;
FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
THEN
(* Identify stack-segment selector for new privilege level in current TSS *)
IF current TSS is 32-bit
THEN
TSSstackAddress := (new code-segment DPL « 3) + 4;
IF (TSSstackAddress + 5) > current TSS limit
THEN #TS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 4);
NewESP := 4 bytes loaded from (TSS base + TSSstackAddress);
ELSE (* current TSS is 16-bit *)
TSSstackAddress := (new code-segment DPL « 2) + 2
IF (TSSstackAddress + 3) > current TSS limit
THEN #TS(error_code(current TSS selector,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSS := 2 bytes loaded from (TSS base + TSSstackAddress + 2);
NewESP := 2 bytes loaded from (TSS base + TSSstackAddress);
FI;
IF NewSS is NULL
THEN #TS(EXT); FI;
IF NewSS index is not within its descriptor-table limits
or NewSS RPL ≠ new code-segment DPL
THEN #TS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ new code-segment DPL
or new stack-segment Type does not indicate writable data segment
THEN #TS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
IF NewSS is not present
THEN #SS(error_code(NewSS,0,EXT)); FI;
(* idt operand to error_code is 0 because selector is used *)
NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
ELSE (* IA-32e mode *)

3-522 Vol. 2A

INT n/INTO/INT3/INT1—Call to Interrupt Procedure
IF IDT-gate IST = 0
    THEN TSSstackAddress := (new code-segment DPL « 3) + 4;
    ELSE TSSstackAddress := (IDT gate IST « 3) + 28;
    FI;
IF (TSSstackAddress + 7) > current TSS limit
    THEN #TS(error_code(current TSS selector,0,EXT); FI;
(* idt operand to error_code is 0 because selector is used *)
NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
NewSS := new code-segment DPL; (* NULL selector with RPL = new CPL *)
IF IDT-gate IST = 0
    THEN
        NewSSP := IA32_PLi_SSP (* where i = new code-segment DPL *)
    ELSE
        NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT-gate IST « 3)
        (* Check if shadow stacks are enabled at CPL 0 *)
        IF ShadowStackEnabled(CPL 0)
            THEN NewSSP := 8 bytes loaded from NewSSPAddress; FI;
        FI;
    FI;
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 24 bytes (error code pushed)
            or 20 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE
    FI;
IF IDT gate is 16-bit
    THEN
        IF new stack does not have room for 12 bytes (error code pushed)
            or 10 bytes (no error code pushed);
            THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE (* 64-bit IDT gate*)
        IF StackAddress is non-canonical
            THEN #SS(EXT); FI; (* Error code contains NULL selector *)
    FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limits
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ESP := NewESP;
        SS := NewSS; (* Segment descriptor information also loaded *)
    ELSE (* IA-32e mode *)
        IF instruction pointer from IDT gate contains a non-canonical address
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        RSP := NewRSP & FFFFFFFFFFFFFFH;
        SS := NewSS;
    FI;
IF IDT gate is 32-bit
    THEN
        CS:EIP := Gate(CS:EIP); (* Segment descriptor information also loaded *)
    ELSE
IF IDT gate 16-bit
  THEN
      CS:IP := Gate(CS:IP);
      (* Segment descriptor information also loaded *)
  ELSE (* 64-bit IDT gate *)
      CS:RIP := Gate(CS:RIP);
      (* Segment descriptor information also loaded *)
  FI;
FI;
IF IDT gate is 32-bit
  THEN
      Push(far pointer to old stack);
      (* Old SS and ESP, 3 words padded to 4 *)
      Push(EFLAGS);
      Push(far pointer to return instruction);
      (* Old CS and EIP, 3 words padded to 4 *)
      Push(ErrorCode); (* If needed, 4 bytes *)
  ELSE (* 64-bit IDT gate *)
      Push(far pointer to old stack);
      (* Old SS and SP, 2 words *)
      Push(EFLAGS(15:0));
      Push(far pointer to return instruction);
      (* Old CS and IP, 2 words *)
      Push(ErrorCode); (* If needed, 2 bytes *)
  FI;
FI;
IF ShadowStackEnabled(CPL) AND CPL = 3
  THEN
      IF IA32_EFER.LMA = 0
      THEN IA32_PL3_SSP := SSP;
      ELSE (* adjust so bits 63:N get the value of bit N–1, where N is the CPU's maximum linear-address width *)
          IA32_PL3_SSP := LA_adjust(SSP);
      FI;
  FI;
CPL := new code-segment DPL;
CS(RPL) := CPL;
IF ShadowStackEnabled(CPL)
  oldSSP := SSP
  SSP := NewSSP
  IF SSP & 0x07 != 0
      THEN #GP(0); FI;
  (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
  IF (SSP & ~0x1F) != ((SSP – 24) & ~0x1F)
      #GP(0); FI;
  IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
THEN #GP(0); Fi;
expected_token_value = SSP  (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT   (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value) != expected_token_value
THEN #GP(0); Fi;
IF oldSS.DPL != 3
ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(oldSSP);
FI;
FI;
IF EndbranchEnabled (CPL)
IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
IA32_S_CET.SUPPRESS = 0
FI;
IF IDT gate is interrupt gate
THEN IF := 0 (* Interrupt flag set to 0, interrupts disabled *); Fi;
TF := 0;
VM := 0;
RF := 0;
NT := 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
 (* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
THEN
  IF TSS limit < 9
  THEN #TS(error_code(current TSS selector,0,EXT)); Fi;
  (* idt operand to error_code is 0 because selector is used *)
  NewSS := 2 bytes loaded from (current TSS base + 8);
  NewESP := 4 bytes loaded from (current TSS base + 4);
  ELSE (* current TSS is 16-bit *)
  IF TSS limit < 5
  THEN #TS(error_code(current TSS selector,0,EXT)); Fi;
  (* idt operand to error_code is 0 because selector is used *)
  NewSS := 2 bytes loaded from (current TSS base + 4);
  NewESP := 2 bytes loaded from (current TSS base + 2);
  FI;
IF NewSS is NULL
  THEN #TS(EXT); Fi; (* Error code contains NULL selector * )
IF NewSS index is not within its descriptor table limits
or NewSS RPL ≠ 0
  THEN #TS(error_code(NewSS,0,EXT)); Fi;
  (* idt operand to error_code is 0 because selector is used *)
Read new stack-segment descriptor for NewSS in GDT or LDT;
IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
  THEN #TS(error_code(NewSS,0,EXT)); Fi;
  (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
  THEN #SS(error_code(NewSS,0,EXT)); Fi;
  (* idt operand to error_code is 0 because selector is used *)
NewSS := IA32_PLO_SSP  (* the new code-segment DPL must be 0 *)
IF IDT gate is 32-bit
THEN
If new stack does not have room for 40 bytes (error code pushed)
or 36 bytes (no error code pushed)
    THEN #SS(error_code(NewSS,0,EXT)); Fi;
    (* idt operand to error_code is 0 because selector is used *)
ELSE (* IDT gate is 16-bit)
    IF new stack does not have room for 20 bytes (error code pushed)
or 18 bytes (no error code pushed)
    THEN #SS(error_code(NewSS,0,EXT)); Fi;
    (* idt operand to error_code is 0 because selector is used *)
FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); Fi; (* Error code contains NULL selector *)
tempEFLAGS := EFLAGS;
VM := 0;
TF := 0;
RF := 0;
NT := 0;
IF service through interrupt gate
    THEN IF = 0; Fi;
TempSS := SS;
TempESP := ESP;
SS := NewSS;
ESP := NewESP;
    (* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
    Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
Push(TempEFlags);
Push(CS);
Push(EIP);
GS := 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS := 0;
DS := 0;
ES := 0;
CS := Gate(CS); (* Segment descriptor information also loaded *)
CS(RPL) := 0;
CPL := 0;
IF IDT gate is 32-bit
    THEN
        EIP := Gate(instruction pointer);
    ELSE (* IDT gate is 16-bit *)
        EIP := Gate(instruction pointer) AND 0000FFFFH;
FI;
IF ShadowStackEnabled(0)
    oldSSP := SSP
    SSP := NewSSP
    IF SSP & 0x07 != 0
        THEN #GP(0); Fi;
    (* Token and CS:LIP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
    IF (SSP & ~0x1F) != ((SSP - 24) & ~0x1F)
#GP(0); Fl;
IF ((IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
THEN #GP(0); Fl;
expected_token_value = SSP (* busy bit - bit position 0 - must be clear *)
new_token_value = SSP | BUSY_BIT (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8B(SSP, new_token_value, expected_token_value) != expected_token_value
THEN #GP(0); Fl;
FI;
IF EndbranchEnabled (CPL)
IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANHC; 
IA32_S_CET.SUPPRESS = 0
FI;
(* Start execution of new routine in Protected Mode *)
END;

INTRA-PRIVILEGE-LEVEL-INTERRUPT:
NewSSP = SSP;
CHECK_SS_TOKEN = 0
(* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
IF IDT-descriptor IST ≠ 0
THEN
TSSstackAddress := (IDT-descriptor IST « 3) + 2B;
IF (TSSstackAddress + 7) > TSS limit
THEN #TS(error_code(current TSS selector,0,EXT)); Fl;
(* idt operand to error_code is 0 because selector is used *)
NewRSP := 8 bytes loaded from (current TSS base + TSSstackAddress);
ELSE NewRSP := RSP;
FI;
ELSE IF ShadowStackEnabled(CPL)
THEN
NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT gate IST « 3) 
NewSSP := 8 bytes loaded from NewSSPAddress
CHECK_SS_TOKEN = 1
FI;
FI;
IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
THEN
IF current stack does not have room for 16 bytes (error code pushed)
or 12 bytes (no error code pushed)
THEN #SS(EXT); Fl; (* Error code contains NULL selector *)
ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
IF current stack does not have room for 8 bytes (error code pushed)
or 6 bytes (no error code pushed)
THEN #SS(EXT); Fl; (* Error code contains NULL selector *)
ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
IF NewRSP contains a non-canonical address
THEN #SS(EXT); (* Error code contains NULL selector *)
FI;
FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
THEN
IF instruction pointer from IDT gate is not within new code-segment limit
THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
ELSE
IF instruction pointer from IDT gate contains a non-canonical address
THEN #GP(EXT); Fl; (* Error code contains NULL selector *)
RSP := NewRSP & FFFFFFFFFFFFFFFF0H;
FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
THEN
Push (EFLAGS);
Push (far pointer to return instruction); (* 3 words padded to 4 *)
CS:EIP := Gate(CS:EIP); (* Segment descriptor information also loaded *)
Push (ErrorCode); (* If any *)
ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
Push (far pointer to old stack);
(* Old SS and SP, each an 8-byte push *)
Push (RFLAGS); (* 8-byte push *)
Push (far pointer to return instruction);
(* Old CS and RIP, each an 8-byte push *)
Push (ErrorCode); (* If needed, 8 bytes *)
CS:RIP := GATE(CS:RIP);
(* Segment descriptor information also loaded *)
FI;
FI;
CS(RPL) := CPL;
IF ShadowStackEnabled(CPL)
IF CHECK_SS_TOKEN == 1
THEN
IF NewSSP & 0x07 != 0
THEN #GP(0); Fl;
(* Token and CS:IP:oldSSP pushed on shadow stack must be contained in a naturally aligned 32-byte region *)
IF (NewSSP & ~0x1F) != ((NewSSP – 24) & ~0x1F)
#GP(0); Fl;
(* If IA32_EFER.LMA and CS.L) = 0 AND NewSSP[63:32] != 0)
#GP(0); Fl;
expected_token_value = NewSSP (* busy bit - bit position 0 - must be clear *)
nex_token_value = NewSSP | BUSY_BIT (* Set the busy bit *)
IF shadow_stack_lock_cmpxchg8b(NewSSP, new_token_value, expected_token_value) != expected_token_value
THEN #GP(0); Fl;
(* Align to next 8 byte boundary *)
tempSSP = SSP;
Shadow_stack_store 4 bytes of 0 to (NewSSP – 4)
SSP = newSSP & 0xFFFFFFFFFFFFFF8H;
(* push cs:ip:ssp on shadow stack * )
ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
IF CPL = 3
    THEN
        IA32_U_CET.TRACKER = WAIT_FOR ENDBRANCH
        IA32_U_CET.SUPPRESS = 0
    ELSE
        IA32_S_CET.TRACKER = WAIT FOR ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
    FI;
FI;
IF IDT gate is interrupt gate
    THEN IF := 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
    TF := 0;
    NT := 0;
    VM := 0;
    RF := 0;
END;

Flags Affected
The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on
the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the
interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions
#GP(error_code) If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code
segment limits.
    If the segment selector in the interrupt, trap, or task gate is NULL.
    If an interrupt, trap, or task gate, code segment, or TSS segment selector index is outside its
descriptor table limits.
    If the vector selects a descriptor outside the IDT limits.
    If an IDT descriptor is not an interrupt, trap, or task gate.
    If an interrupt is generated by the INT n, INT3, or INTO instruction and the DPL of an interrupt,
    trap, or task gate is less than the CPL.
    If the segment selector in an interrupt or trap gate does not point to a segment descriptor for
    a code segment.
    If the segment selector for a TSS has its local/global bit set for local.
    If a TSS segment descriptor specifies that the TSS is busy or not available.
    If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
    If the token and the stack frame to be pushed on shadow stack are not contained in a naturally
    aligned 32-byte region of the shadow stack.
    If "supervisor Shadow Stack" token on new shadow stack is marked busy.
    If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow
    stack" token is beyond 4GB.
    If SSP address in "supervisor shadow stack" token does not match SSP address in
    IA32_PLi_SSP (where i is the new CPL).
#SS(error_code) If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
   If the SS register is being loaded and the segment pointed to is marked not present.
   If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.

#NP(error_code) If code segment, interrupt gate, trap gate, task gate, or TSS is not present.

#TS(error_code) If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
   If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
   If the stack segment selector in the TSS is NULL.
   If the stack segment for the TSS is not a writable data segment.
   If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the interrupt vector number is outside the IDT limits.

#SS If stack limit violation on push.
   If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)  (For INT n, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt, trap, or task gate is not equal to 3.
   If the instruction pointer in the IDT or in the interrupt, trap, or task gate is beyond the code segment limits.
   If the segment selector in the interrupt, trap, or task gate is NULL.
   If a interrupt gate, trap gate, task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
   If the vector selects a descriptor outside the IDT limits.
   If an IDT descriptor is not an interrupt, trap, or task gate.
   If an interrupt is generated by INT n, INT3, or INTO and the DPL of an interrupt, trap, or task gate is less than the CPL.
   If the segment selector in an interrupt or trap gate does not point to a segment descriptor for a code segment.
   If the segment selector for a TSS has its local/global bit set for local.

#SS(error_code) If the SS register is being loaded and the segment pointed to is marked not present.
   If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.

#NP(error_code) If code segment, interrupt gate, trap gate, task gate, or TSS is not present.
#TS(error_code) If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
If DPL of the stack segment descriptor for the TSS’s stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
If the stack segment selector in the TSS is NULL.
If the stack segment for the TSS is not a writable data segment.
If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#OF If the INTO instruction is executed and the OF flag is set.

#UD If the LOCK prefix is used.

#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code) If the instruction pointer in the 64-bit interrupt gate or trap gate is non-canonical.
If the segment selector in the 64-bit interrupt or trap gate is NULL.
If the vector selects a descriptor outside the IDT limits.
If the vector points to a gate which is in non-canonical space.
If the vector points to a descriptor which is not a 64-bit interrupt gate or a 64-bit trap gate.
If the descriptor pointed to by the gate selector is outside the descriptor table limit.
If the descriptor pointed to by the gate selector is in non-canonical space.
If the descriptor pointed to by the gate selector is not a code segment.
If the descriptor pointed to by the gate selector doesn’t have the L-bit set, or has both the L-bit and D-bit set.
If the descriptor pointed to by the gate selector has DPL > CPL.
If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
If the token and the stack frame to be pushed on shadow stack are not contained in a naturally aligned 32-byte region of the shadow stack.
If “supervisor shadow stack” token on new shadow stack is marked busy.
If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4GB.
If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).

#SS(error_code) If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).

#NP(error_code) If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.

#TS(error_code) If an attempt to load RSP from the TSS causes an access to non-canonical space.
If the RSP from the TSS is outside descriptor table limits.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

#AC(EXT) If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.
INVD—Invalidate Internal Caches

<table>
<thead>
<tr>
<th>Opcode1</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 08</td>
<td>INVD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Flush internal caches; initiate flushing of external caches.</td>
</tr>
</tbody>
</table>

**Description**

Invalidates (flushes) the processor’s internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

On processors that support processor reserved memory, the INVD instruction cannot be executed when processor reserved memory protections are activated. See Section 36.5, “EPC and Management of EPC Pages,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D.

Some processors prevent execution of INVD after BIOS execution is complete. They report this by enumerating CPUID.(EAX=07H,ECX=1H):EAX[bit 30] as 1. On such processors, INVD cannot be executed if bit 0 of SR_BIOS_DONE (MSR address 151H) is 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

**Operation**

Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)

**Flags Affected**

None.
**Protected Mode Exceptions**

- **#GP(0)**  
  - If the current privilege level is not 0.
  - If the processor reserved memory protections are activated.
  - If CPUID.(EAX=07H, ECX=1H):EAX[30] = 1 and bit 0 is set in MSR_BIOS_DONE (MSR address 151H).

- **#UD**  
  - If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP(0)**  
  - If the processor reserved memory protections are activated.
  - If CPUID.(EAX=07H, ECX=1H):EAX[30] = 1 and bit 0 is set in MSR_BIOS_DONE (MSR address 151H).

- **#UD**  
  - If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- **#GP(0)**  
  - The INVD instruction cannot be executed in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
INVLPG—Invalidate TLB Entries

<table>
<thead>
<tr>
<th>Opcode1</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01/7</td>
<td>INVLPG m</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Invalidate TLB entries for page containing m.</td>
</tr>
</tbody>
</table>

NOTES:
1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.1

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction invalidates TLB entries associated with the current PCID and may or may not do so for TLB entries associated with other PCIDs. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see “MOV—Move to/from Control Registers” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

This instruction’s operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility
The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation
Invalidate(RelevantTLBEntries);
Continue; (* Continue execution *)

Flags Affected
None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD Operand is a register.
If the LOCK prefix is used.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A), the instruction invalidates all of them.
Real-Address Mode Exceptions
#UD Operand is a register.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) The INVLPG instruction cannot be executed at the virtual-8086 mode.

64-Bit Mode Exceptions
#GP(0) If the current privilege level is not 0.
#UD Operand is a register.
If the LOCK prefix is used.
INVPCID—Invalidate Process-Context Identifier

**Description**

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, “Caching Translation Information,” in the *Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.* Invalidation is based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits.

There are four INVPCID types currently defined:

- **Individual-address invalidation:** If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- **Single-context invalidation:** If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- **All-context invalidation, including global translations:** If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- **All-context invalidation:** If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-25. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

![Figure 3-25. INVPCID Descriptor](image)

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), the instruction invalidates all of them.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H; executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside IA-32e mode (see Section 4.10.1, “Process-Context Identifiers (PCIDs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

**Operation**

INVPCID_TYPE := value of register operand;  // must be in the range of 0-3
INVPCID_DESC := value of memory operand;
CASE INVPCID_TYPE OF
  0:  // individual-address invalidation
      PCID := INVPCID_DESC[11:0];
      L_ADDR := INVPCID_DESC[127:64];
      Invalidate mappings for L_ADDR associated with PCID except global translations;
      BREAK;
  1:  // single PCID invalidation
      PCID := INVPCID_DESC[11:0];
      Invalidate all mappings associated with PCID except global translations;
      BREAK;
  2:  // all PCID invalidation including global translations
      Invalidate all mappings for all PCIDs, including global translations;
      BREAK;
  3:  // all PCID invalidation retaining global translations
      Invalidate all mappings for all PCIDs except global translations;
      BREAK;
ESAC;

**Intel C/C++ Compiler Intrinsic Equivalent**

INVPCID void _invpcid(unsigned __int32 type, void * descriptor);

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#GP(0) If the current privilege level is not 0.
If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the source operand is located in an execute-only code segment.
If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
If bits 63:12 of INVPCID_DESC are not all zero.
If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#PF(fault-code) If a page fault occurs in accessing the memory operand.
#SS(0) If the memory operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.
#UD If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID[bit 10] = 0.
If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
If bits 63:12 of INVPCID_DESC are not all zero.
If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#UD If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID[bit 10] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) The INVPCID instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the current privilege level is not 0.
If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
If bits 63:12 of INVPCID_DESC are not all zero.
If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.
If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#PF(fault-code) If a page fault occurs in accessing the memory operand.
#SS(0) If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID[bit 10] = 0.
IRET/IRETD/IRETQ—Interrupt Return

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>IRET</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Interrupt return (16-bit operand size).</td>
</tr>
<tr>
<td>CF</td>
<td>IRETD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Interrupt return (32-bit operand size).</td>
</tr>
<tr>
<td>REX.W + CF</td>
<td>IRETQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Interrupt return (64-bit operand size).</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction performs a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, “Handling Multiple NMIs” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A), execution of the IRET instruction unblocks NMIs.
This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 17, “Control-flow Enforcement Technology (CET)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for CET details.

**Instruction ordering.** IRET is a serializing instruction. See Section 9.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

```
IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
    FI;

REAL-ADDRESS-MODE:
    IF OperandSize = 32
        THEN
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS := Pop();
            EFLAGS := (tempEFLAGS AND 257FDH) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS := Pop(); (* 16-bit pop *)
            EFLAGS[15:0] := Pop();
        FI;
    END;

RETURN-FROM-VIRTUAL-8086-MODE:
    (* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP := Pop();
                CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS := Pop();
                (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS := Pop(); (* 16-bit pop *)
                EFLAGS[15:0] := Pop(); (* IOPL in EFLAGS not modified by pop *)
            FI;
        ELSE (* OperandSize = 16 *)
            EIP := Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS := Pop(); (* 16-bit pop *)
            EFLAGS[15:0] := Pop(); (* IOPL in EFLAGS not modified by pop *)
    FI;
```
IF EIP not within CS limit
    THEN #GP(0); Fl;
ELSE
    #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    Fl;
END;

PROTECTED-MODE:
IF NT = 1
    THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    Fl;
IF OperandSize = 32
THEN
    EIP := Pop();
    CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    tempEFLAGS := Pop();
ELSE (* OperandSize = 16 *)
    EIP := Pop(); (* 16-bit pop; clear upper bits *)
    CS := Pop(); (* 16-bit pop *)
    tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
    Fl;
IF tempEFLAGS(VM) = 1 and CPL = 0
    THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE GOTO PROTECTED-MODE-RETURN;
    Fl;
TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
Mark the task just abandoned as NOT BUSY;
IF EIP is not within CS limit
    THEN #GP(0); Fl;
END;
RETURN-TO-VIRTUAL-8086-MODE:
(* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
(* If shadow stack or indirect branch tracking at CPL3 then #GP(0) *)
IF CR4.CET AND (IA32_U_CET.ENDBR_EN OR IA32_U_CET.SHSTK_EN)
    THEN #GP(0); Fl;
shadowStackEnabled = ShadowStackEnabled(CPL)
IF EIP not within CS limit
    THEN #GP(0); Fl;
END;
EFLAGS := tempEFLAGS;
ESP := Pop(); (* Pop 2 words; throw away high-order word *)
ES := Pop(); (* Pop 2 words; throw away high-order word *)
DS := Pop(); (* Pop 2 words; throw away high-order word *)
FS := Pop(); (* Pop 2 words; throw away high-order word *)
GS := Pop(); (* Pop 2 words; throw away high-order word *)
IF shadowStackEnabled
    (* check if 8 byte aligned *)
    IF SSP AND 0x7 != 0
        THEN #CP(FAR-RET/IRET); Fl;
    Fl;
CPL := 3;
(* Resume execution in Virtual-8086 mode *)
tempOldSSP = SSP;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP *
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF shadowStackEnabled
   expected_token_value = tempOldSSP | BUSY_BIT   (* busy bit - bit position 0 - must be set *)
   new_token_value = tempOldSSP                 (* clear the busy bit *)
   shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
   IF CS(RPL) > CPL
      THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
      ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
   IF OperandSize = 32
      THEN
         tempESP := Pop();
         tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
      ELSE IF OperandSize = 16
         THEN
            tempESP := Pop(); (* 16-bit pop; clear upper bits *)
            tempSS := Pop(); (* 16-bit pop *)
         ELSE (* OperandSize = 64 *)
            tempRSP := Pop();
            tempSS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
         FI;
      IF new mode ≠ 64-Bit Mode
         THEN
            IF EIP is not within CS limit
               THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
               IF RIP is non-canonical
                  THEN #GP(0); FI;
            FI;
      FI;
   EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) := tempEFLAGS;
   IF OperandSize = 32 or OperandSize = 64
      THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
   IF CPL ≤ IOPL
      THEN EFLAGS(IF) := tempEFLAGS; FI;
   IF CPL = 0
      THEN
         EFLAGS(IOPL) := tempEFLAGS;
         IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
      FI;
   IF ShadowStackEnabled(CPL)
      (* check if 8 byte aligned *)
      IF SSP AND 0x7 != 0
THEN #CP(FAR-RET/IRET); Fi;
IF CS(RPL) != 3
THEN
    tempSsCS = shadow_stack_load 8 bytes from SSP+16;
    tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
    tempSSP = shadow_stack_load 8 bytes from SSP;
    SSP = SSP + 24;
    (* Do 64 bit compare to detect bits beyond 15 being set *)
    tempCS = CS; (* zero padded to 64 bit *)
    IF tempCS != tempSsCS
        THEN #CP(FAR-RET/IRET); Fi;
    (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
    IF CSBASE + RIP != tempSsEIP
        THEN #CP(FAR-RET/IRET); Fi;
    (* check if 4 byte aligned *)
    IF tempSSP AND 0x3 != 0
        THEN #CP(FAR-RET/IRET); Fi;
FI;
FI;
tempOldCPL = CPL;
CPL := CS(RPL);
IF OperandSize = 64
THEN
    RSP := tempRSP;
    SS := tempSS;
ELSE
    ESP := tempESP;
    SS := tempSS;
FI;
IF new mode != 64-Bit Mode
THEN
    IF EIP is not within CS limit
        THEN #GP(0); Fi;
ELSE (* new mode = 64-bit mode *)
    IF RIP is non-canonical
        THEN #GP(0); Fi;
FI;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP := IA32_PL3_SSP; Fi;
    IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
        ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
        THEN #GP(0); Fi;
    SSP := tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP *
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
FOR each SegReg in (ES, FS, GS, and DS)
   DO
      tempDesc := descriptor cache for SegReg (* hidden part of segment register *)
      IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code))
         THEN (* Segment register invalid *)
            SegmentSelector := 0; (*Segment selector becomes null*)
         FI;
   OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
   IF new mode ≠ 64-Bit Mode
      THEN
         IF EIP is not within CS limit
            THEN #GP(0); FI;
         ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
               THEN #GP(0); FI;
         FI;
   FI;
   EFLAGS (CF, PF, ZF, SF, TF, OF, NT) := tempEFLAGS;
   IF OperandSize = 32 or OperandSize = 64
      THEN EFLAGS(RF, AC, ID) := tempEFLAGS; FI;
   IF CPL ≤ IOPL
      THEN EFLAGS(IF) := tempEFLAGS; FI;
   IF CPL = 0
      THEN
         EFLAGS(IOPL) := tempEFLAGS;
         IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) := tempEFLAGS; FI;
      FI;
   FI;
   IF ShadowStackEnabled(CPL)
      IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
         THEN #CP(FAR-RET/IRET); FI;
      tempSsCS = shadow_stack_load 8 bytes from SSP+16;
      tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
      tempSSP = shadow_stack_load 8 bytes from SSP;
      SSP = SSP + 24;
      tempCS = CS; (* zero padded to 64 bit *)
      IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
         THEN #CP(FAR-RET/IRET); FI;
      IF CSBASE + RIP != tempSsLIP (* 64 bit compare; CSBASE+RIP zero padded to 64 bit for 32 bit LIP *)
         THEN #CP(FAR-RET/IRET); FI;
      IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
         THEN #CP(FAR-RET/IRET); FI;
      IF ((IA32_EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0) OR
         ((IA32_EFER.LMA AND CS.L) = 1 AND tempSSP is not canonical relative to the current paging mode)
         THEN #GP(0); FI;
      FI;
   FI;
   IF ShadowStackEnabled(CPL)
      IF IA32_EFER.LMA = 1
         (* In IA-32e-mode the IRET may be switching stacks if the interrupt/exception was delivered
            through an IDT with a non-zero IST *)
         (* In IA-32e mode for same CPL IRET there is always a stack switch. The below check verifies if the
            stack switch was to self stack and if so, do not try to free the token on this shadow stack. If the
            stack switch was to the other stack, do not try to free the token on this shadow stack. *)
         THEN #GP(0); FI;
      FI;
tempSSP was not to same stack then there was a stack switch so do attempt to free the token *)

IF tempSSP != SSP
    THEN
        expected_token_value = SSP | BUSY_BIT (* busy bit - bit position 0 - must be set *)
        new_token_value = SSP (* clear the busy bit *)
        shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)
    FI;
    FI;
    SSP := tempSSP
    FI;
END;

IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
        ELSE IF OperandSize = 32
            THEN
                EIP := Pop();
                CS := Pop();
                tempEFLAGS := Pop();
            ELSE IF OperandSize = 16
                THEN
                    EIP := Pop(); (* 16-bit pop; clear upper bits *)
                    CS := Pop(); (* 16-bit pop *)
                    tempEFLAGS := Pop(); (* 16-bit pop; clear upper bits *)
                FI;
            ELSE (* OperandSize = 64 *)
                THEN
                    RIP := Pop();
                    CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    tempRFLAGS := Pop();
                FI;
            IF CS.RPL > CPL
                THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
            ELSE
                IF instruction began in 64-Bit Mode
                    THEN
                        IF OperandSize = 32
                            THEN
                                ESP := Pop();
                                SS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                            ELSE IF OperandSize = 16
                                THEN
                                    ESP := Pop(); (* 16-bit pop; clear upper bits *)
                                    SS := Pop(); (* 16-bit pop *)
                                ELSE (* OperandSize = 64 *)
                                    RSP := Pop();
                                    SS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                                FI;
                            FI;
                        FI;
                    GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
                END;
Flags Affected
All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task’s TSS.

Protected Mode Exceptions
#GP(0)  If the return code or stack segment selector is NULL.
        If the return instruction pointer is not within the return code segment limit.
#GP(selector)  If a segment selector index is outside its descriptor table limits.
        If the return code segment selector RPL is less than the CPL.
        If the DPL of a conforming-code segment is greater than the return code segment selector
        RPL.
        If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment
        selector.
        If the stack segment descriptor DPL is not equal to the RPL of the return code segment
        selector.
        If the stack segment is not a writable data segment.
        If the stack segment selector RPL is not equal to the RPL of the return code segment
        selector.
        If the segment descriptor for a code segment does not indicate it is a code segment.
        If the segment selector for a TSS has its local/global bit set for local.
        If a TSS segment descriptor specifies that the TSS is not busy.
        If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)  If the top bytes of stack are not within stack limits.
        If the return stack segment is not present.
#NP (selector)  If the return code segment is not present.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If an unaligned memory reference occurs when the CPL is 3 and alignment checking is
        enabled.
#UD  If the LOCK prefix is used.
#CP (Far-RET/IRET)  If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP
        (returning to CPL 3) is not 4 byte aligned.
        If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when
        returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
        If return instruction pointer from stack and shadow stack do not match.

Real-Address Mode Exceptions
#GP  If the return instruction pointer is not within the return code segment limit.
#SS  If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions
#GP(0)  If the return instruction pointer is not within the return code segment limit.
        IF IOPL not equal to 3.
#PF(fault-code)  If a page fault occurs.
#SS(0)  If the top bytes of stack are not within stack limits.
#AC(0)  If an unaligned memory reference occurs and alignment checking is enabled.
#UD  If the LOCK prefix is used.
Compatibility Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.
Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If EFLAGS.NT[bit 14] = 1.
If the return code segment selector is NULL.
If the stack segment selector is NULL going back to compatibility mode.
If the stack segment selector is NULL going back to CPL3 64-bit mode.
If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
If the return instruction pointer is not within the return code segment limit.
If the return instruction pointer is non-canonical.

#GP(Selector) If a segment selector index is outside its descriptor table limits.
If a segment descriptor memory address is non-canonical.
If the segment descriptor for a code segment does not indicate it is a code segment.
If the proposed new code segment descriptor has both the D-bit and L-bit set.
If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
If CPL is greater than the RPL of the code segment selector.
If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
If the stack segment is not a writable data segment.
If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0) If an attempt to pop a value off the stack violates the SS limit.
If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
If the return stack segment is not present.

#NP (selector) If the return code segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

#UD If the LOCK prefix is used.

#CP (Far-RET/IRET) If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.
If return instruction pointer from stack and shadow stack do not match.
## Jcc—Jump if Condition Is Met

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>77 cb</td>
<td>JA rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>73 cb</td>
<td>JAE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if above or equal (CF=0).</td>
</tr>
<tr>
<td>72 cb</td>
<td>JB rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if below (CF=1).</td>
</tr>
<tr>
<td>76 cb</td>
<td>JBE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>72 cb</td>
<td>JC rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if carry (CF=1).</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JCXZ rel8</td>
<td>D</td>
<td>N.E.</td>
<td>Valid</td>
<td>Jump short if CX register is 0.</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JECXZ rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if ECX register is 0.</td>
</tr>
<tr>
<td>E3 cb</td>
<td>JRCXZ rel8</td>
<td>D</td>
<td>Valid</td>
<td>N.E.</td>
<td>Jump short if RCX register is 0.</td>
</tr>
<tr>
<td>74 cb</td>
<td>JE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if equal (ZF=1).</td>
</tr>
<tr>
<td>7F cb</td>
<td>JG rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>7D cb</td>
<td>JGE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>7C cb</td>
<td>JL rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if less (SF ≠ OF).</td>
</tr>
<tr>
<td>7E cb</td>
<td>JLE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if less or equal (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>76 cb</td>
<td>JNA rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>72 cb</td>
<td>JNBE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not above or equal (CF=1).</td>
</tr>
<tr>
<td>73 cb</td>
<td>JC rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not carry (CF=0).</td>
</tr>
<tr>
<td>7S cb</td>
<td>JNE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not equal (ZF=0).</td>
</tr>
<tr>
<td>7E cb</td>
<td>JNG rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not greater (ZF=1 or SF=OF).</td>
</tr>
<tr>
<td>7C cb</td>
<td>JNGE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not greater or equal (SF=OF).</td>
</tr>
<tr>
<td>7D cb</td>
<td>JNL rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not less (SF=OF).</td>
</tr>
<tr>
<td>7F cb</td>
<td>JNLE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>71 cb</td>
<td>JNO rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not overflow (OF=0).</td>
</tr>
<tr>
<td>7B cb</td>
<td>JNP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not parity (PF=0).</td>
</tr>
<tr>
<td>79 cb</td>
<td>JNS rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not sign (SF=0).</td>
</tr>
<tr>
<td>75 cb</td>
<td>JNZ rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if not zero (ZF=0).</td>
</tr>
<tr>
<td>70 cb</td>
<td>JO rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if overflow (OF=1).</td>
</tr>
<tr>
<td>7A cb</td>
<td>JP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if parity (PF=1).</td>
</tr>
<tr>
<td>7A cb</td>
<td>JPE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if parity even (PF=1).</td>
</tr>
<tr>
<td>7B cb</td>
<td>JPO rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if parity odd (PF=0).</td>
</tr>
<tr>
<td>7B cb</td>
<td>JS rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if sign (SF=1).</td>
</tr>
<tr>
<td>74 cb</td>
<td>JZ rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short if zero (ZF = 1).</td>
</tr>
<tr>
<td>0F 87 cw</td>
<td>JA rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 87 cd</td>
<td>JA rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if above (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 83 cw</td>
<td>JAE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if above or equal (CF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 83 cd</td>
<td>JAE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if above or equal (CF=0).</td>
</tr>
</tbody>
</table>
### Table of Conditional Jumps

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 82 cw</td>
<td>JB rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if below (CF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 82 cd</td>
<td>JB rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if below (CF=1).</td>
</tr>
<tr>
<td>0F 86 cw</td>
<td>JBE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 86 cd</td>
<td>JBE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if below or equal (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 82 cw</td>
<td>JC rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if carry (CF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 82 cd</td>
<td>JC rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if carry (CF=1).</td>
</tr>
<tr>
<td>0F 84 cw</td>
<td>JE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 84 cd</td>
<td>JE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1).</td>
</tr>
<tr>
<td>0F 84 cw</td>
<td>JZ rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 84 cd</td>
<td>JZ rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1).</td>
</tr>
<tr>
<td>0F 8F cw</td>
<td>JG rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8F cd</td>
<td>JG rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if greater (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 8D cw</td>
<td>JGE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8D cd</td>
<td>JGE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 8C cw</td>
<td>JL rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if less (SF≠OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8C cd</td>
<td>JL rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if less (SF≠OF).</td>
</tr>
<tr>
<td>0F 8E cw</td>
<td>JLE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if less or equal (ZF=1 or SF≠OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8E cd</td>
<td>JLE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if less or equal (ZF=1 or SF≠OF).</td>
</tr>
<tr>
<td>0F 86 cw</td>
<td>JNA rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 86 cd</td>
<td>JNA rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not above (CF=1 or ZF=1).</td>
</tr>
<tr>
<td>0F 82 cw</td>
<td>JNAE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not above or equal (CF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 82 cd</td>
<td>JNAE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 83 cw</td>
<td>JNB rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not below (CF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 83 cd</td>
<td>JNB rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not below (CF=0).</td>
</tr>
<tr>
<td>0F 87 cw</td>
<td>JNBE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 87 cd</td>
<td>JNBE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not below or equal (CF=0 and ZF=0).</td>
</tr>
<tr>
<td>0F 83 cw</td>
<td>JNC rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not carry (CF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 83 cd</td>
<td>JNC rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 85 cw</td>
<td>JNE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not equal (ZF=0). Not supported in 64-bit mode.</td>
</tr>
</tbody>
</table>

**Jcc—Jump if Condition Is Met**
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 85 cd</td>
<td>JNE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 8E cw</td>
<td>JNG rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8E cd</td>
<td>JNG rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not greater (ZF=1 or SF≠ OF).</td>
</tr>
<tr>
<td>0F 8C cw</td>
<td>JNGE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8C cd</td>
<td>JNGE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not greater or equal (SF≠ OF).</td>
</tr>
<tr>
<td>0F 8D cw</td>
<td>JNL rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not less (SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8D cd</td>
<td>JNL rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 8F cw</td>
<td>JNLE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8F cd</td>
<td>JNLE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not less or equal (ZF=0 and SF=OF).</td>
</tr>
<tr>
<td>0F 81 cw</td>
<td>JNO rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not overflow (OF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 81 cd</td>
<td>JNO rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not overflow (OF=0).</td>
</tr>
<tr>
<td>0F 8B cw</td>
<td>JNP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not parity (PF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8B cd</td>
<td>JNP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not parity (PF=0).</td>
</tr>
<tr>
<td>0F 89 cw</td>
<td>JNS rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not sign (SF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 89 cd</td>
<td>JNS rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not sign (SF=0).</td>
</tr>
<tr>
<td>0F 85 cw</td>
<td>JNZ rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if not zero (ZF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 85 cd</td>
<td>JNZ rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if not zero (ZF=0).</td>
</tr>
<tr>
<td>0F 80 cw</td>
<td>JO rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if overflow (OF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 80 cd</td>
<td>JO rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if overflow (OF=1).</td>
</tr>
<tr>
<td>0F 8A cw</td>
<td>JP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if parity (PF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8A cd</td>
<td>JP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if parity (PF=1).</td>
</tr>
<tr>
<td>0F 8A cw</td>
<td>JPE rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if parity even (PF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8A cd</td>
<td>JPE rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if parity even (PF=1).</td>
</tr>
<tr>
<td>0F 8B cw</td>
<td>JPO rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if parity odd (PF=0). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 8B cd</td>
<td>JPO rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if parity odd (PF=0).</td>
</tr>
<tr>
<td>0F 88 cw</td>
<td>JS rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if sign (SF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 88 cd</td>
<td>JS rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if sign (SF=1).</td>
</tr>
<tr>
<td>0F 84 cw</td>
<td>JZ rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1). Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>0F 84 cd</td>
<td>JZ rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near if 0 (ZF=1).</td>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Offset</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, andZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the Jcc instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of −128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each Jcc mnemonic are given in the “Description” column of the table on the preceding page. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The Jcc instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the Jcc instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
```

BEYOND:

The JRCXZ, JECXZ, and JCXZ instructions differ from other Jcc instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute $2^{64}$, $2^{32}$ or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32-bit offset sign extended to 64 bits.
**Operation**

IF condition

    THEN
        tempEIP := EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN tempEIP := tempEIP AND 0000FFFFH;
        FI;
        IF tempEIP is not within code segment limit
            THEN #GP(0);
        ELSE EIP := tempEIP
        FI;
    FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If the offset being jumped to is beyond the limits of the CS segment.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0) If the memory address is in a non-canonical form.

#UD If the LOCK prefix is used.
**JMP—Jump**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EB cb</td>
<td>JMP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits.</td>
</tr>
<tr>
<td>E9 cw</td>
<td>JMP rel16</td>
<td>D</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>E9 cd</td>
<td>JMP rel32</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits.</td>
</tr>
<tr>
<td>FF /4</td>
<td>JMP r/m16</td>
<td>M</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, absolute indirect, address = zero-extended r/m16. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>FF /4</td>
<td>JMP r/m32</td>
<td>M</td>
<td>N.S.</td>
<td>Valid</td>
<td>Jump near, absolute indirect, address given in r/m32. Not supported in 64-bit mode.</td>
</tr>
<tr>
<td>FF /4</td>
<td>JMP r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Jump near, absolute indirect, RIP = 64-Bit offset from register or memory.</td>
</tr>
<tr>
<td>EA cd</td>
<td>JMP ptr16:16</td>
<td>S</td>
<td>Inv.</td>
<td>Valid</td>
<td>Jump far, absolute, address given in operand.</td>
</tr>
<tr>
<td>EA cp</td>
<td>JMP ptr16:32</td>
<td>S</td>
<td>Inv.</td>
<td>Valid</td>
<td>Jump far, absolute, address given in operand.</td>
</tr>
<tr>
<td>FF /5</td>
<td>JMP m16:16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump far, absolute indirect, address given in m16:16.</td>
</tr>
<tr>
<td>FF /5</td>
<td>JMP m16:32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Jump far, absolute indirect, address given in m16:32.</td>
</tr>
<tr>
<td>REX.W</td>
<td>FF /5</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Jump far, absolute indirect, address given in m16:64.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Segment + Absolute Address</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Offset</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- **Near jump**—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- **Short jump**—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- **Far jump**—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- **Task switch**—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 8, in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information on performing task switches with the JMP instruction). **Near and Short Jumps.** When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current
value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (rel8) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (r/m16 or r/m32). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (rel8, rel16, or rel32) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

**Far Jumps in Real-Address or Virtual-8086 Mode.** When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Jumps in Protected Mode.** When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task’s code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 8 in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for detailed information on the mechanics of a task switch.
Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS’s previous task link field is not loaded with the old task’s TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions” and Chapter 17, “Control-flow Enforcement Technology (CET)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for CET details.

**In 64-Bit Mode.** The instruction’s operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

**Instruction ordering.** Instructions following a far jump may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far jump have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Instructions sequentially following a near indirect JMP instruction (i.e., those not at the target) may be executed speculatively. If software needs to prevent this (e.g., in order to prevent a speculative execution side channel), then an INT3 or LFENCE instruction opcode can be placed after the near indirect JMP in order to block speculative execution.

**Operation**

IF near jump
  IF 64-bit Mode
    THEN
      IF near relative jump
        THEN
          tempRIP := RIP + DEST; (* RIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempRIP := DEST;
        FI;
      ELSE
        IF near relative jump
          THEN
            tempEIP := EIP + DEST; (* EIP is instruction following JMP instruction*)
          ELSE (* Near absolute jump *)
            tempEIP := DEST;
          FI;
        FI;
    FI;
  FI;
  IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and tempEIP outside code segment limit
    THEN #GP(0); FI
  IF 64-bit mode and tempRIP is not canonical
    THEN #GP(0);
  FI;
  IF OperandSize = 32
    THEN
      EIP := tempEIP;
    ELSE
      IF OperandSize = 16
        THEN (* OperandSize = 16 *)
          EIP := tempEIP AND 0000FFFFH;
      ELSE (* OperandSize = 64 *)
        RIP := tempRIP;
IF (JMP near indirect, absolute indirect)
  IF EndbranchEnabledAndNotSuppressed(CPL)
    IF CPL = 3
      THEN
        IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
          THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
          FI;
        ELSE
          IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
            THEN
              IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            FI;
        FI;
    FI;
  FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
THEN
  tempEIP := DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
  IF tempEIP is beyond code segment limit
    THEN #GP(0); FI;
  CS := DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
  IF OperandSize = 32
    THEN
      EIP := tempEIP; (* DEST is ptr16:32 or [m16:32] *)
    ELSE (* OperandSize = 16 *)
      EIP := tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
    FI;
  FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
THEN
  IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
  or segment selector in target operand NULL
    THEN #GP(0); FI;
  IF segment selector index not within descriptor table limits
    THEN #GP(new selector); FI;
  Read type and access rights of segment descriptor;
  IF (IA32_EFER.LMA = 0)
    THEN
      IF segment type is not a conforming or nonconforming code
      segment, call gate, task gate, or TSS
        THEN #GP(segment selector); FI;
    ELSE
      IF segment type is not a conforming or nonconforming code segment
      call gate
        THEN #GP(segment selector); FI;
    FI;
  Depending on type and access rights:
    GO TO CONFORMING-CODE-SEGMENT;
    GO TO NONCONFORMING-CODE-SEGMENT;
GO TO CALL-GATE;
GO TO TASK-GATE;
GO TO TASK-STATE-SEGMENT;
ELSE
  #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
  IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
  IF DPL > CPL
    THEN #GP(segment selector); FI;
  IF segment not present
    THEN #NP(segment selector); FI;
  IF tempEIP := DEST(Offset);
    IF OperandSize = 16
      THEN tempEIP := tempEIP AND 0000FFFFH;
    FI;
  IF IA32_EFER.LMA = 0 or target mode = Compatibility mode and tempEIP outside code segment limit
    THEN #GP(0); FI;
  IF tempEIP is non-canonical
    THEN #GP(0); FI;
  IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
      THEN #GP(0); FI;
  FI;
  CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
  CS(RPL) := CPL
  EIP := tempEIP;
  IF EndbranchEnabled(CPL)
    IF CPL = 3
      THEN
        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_U_CET.SUPPRESS = 0
      ELSE
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
      FI;
    FI;
  FI;
END;
NONCONFORMING-CODE-SEGMENT:
  IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
    THEN GP(new code segment selector); FI;
  IF (RPL > CPL) OR (DPL ≠ CPL)
    THEN #GP(code segment selector); FI;
  IF segment not present
    THEN #NP(segment selector); FI;
  IF tempEIP := DEST(Offset);
    IF OperandSize = 16
      THEN tempEIP := tempEIP AND 0000FFFFH; FI;
  IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
and tempEIP outside code segment limit
THEN #GP(0); Fl
IF tempEIP is non-canonical THEN #GP(0); Fl;
IF ShadowStackEnabled(CPL)
   IF (IA32_EFER.LMA and DEST(segment selector).L) = 0
      (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
      IF (SSP & 0xFFFFFFFF00000000 != 0)
         THEN #GP(0); Fl;
   FI;
Fl;
CS := DEST[segment selector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
IF EndbranchEnabled(CPL)
   IF CPL = 3
      THEN
         IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
         IA32_U_CET.SUPPRESS = 0
      ELSE
         IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
         IA32_S_CET.SUPPRESS = 0
      FI;
   Fl;
END;

CALL-GATE:
IF call gate DPL < CPL
or call gate DPL < call gate segment-selector RPL
   THEN #GP(call gate selector); Fl;
IF call gate not present
   THEN #NP(call gate selector); FI;
IF call gate code-segment-selector is NULL
   THEN #GP(0); Fl;
IF call gate code-segment selector index outside descriptor table limits
   THEN #GP(code segment selector); Fl;
Read code segment descriptor;
IF code-segment segment descriptor does not indicate a code segment
or code-segment segment descriptor is conforming and DPL > CPL
or code-segment segment descriptor is non-conforming and DPL ≠ CPL
   THEN #GP(code segment selector); Fl;
IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
or code-segment segment descriptor has both L-Bit and D-bit set)
   THEN #GP(code segment selector); Fl;
IF code segment is not present
   THEN #NP(code-segment selector); Fl;
tempEIP := DEST(Offset);
IF GateSize = 16
   THEN tempEIP := tempEIP AND 0000FFFFH; Fl;
IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
outside code segment limit
   THEN #GP(0); Fl
CS := DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
CS(RPL) := CPL;
EIP := tempEIP;
IF EndbranchEnabled(CPL)
  IF CPL = 3
    THEN
      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_U_CET.SUPPRESS = 0
    ELSE
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
  FI;
END;

END;

TASK-GATE:
IF task gate DPL < CPL
  or task gate DPL < task gate segment-selector RPL
    THEN #GP(task gate selector); Fl;
IF task gate not present
    THEN #NP(gate selector); Fl;
Read the TSS segment selector in the task-gate descriptor;
IF TSS segment selector local/global bit is set to local
  or index not within GDT limits
  or descriptor is not a TSS segment
  or TSS descriptor specifies that the TSS is busy
    THEN #GP(TSS selector); Fl;
IF TSS not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;

TASK-STATE-SEGMENT:
IF TSS DPL < CPL
  or TSS DPL < TSS segment-selector RPL
  or TSS descriptor indicates TSS not available
    THEN #GP(TSS selector); Fl;
IF TSS is not present
    THEN #NP(TSS selector); Fl;
SWITCH-TASKS to TSS;
IF EIP not within code segment limit
    THEN #GP(0); Fl;
END;

Flags Affected
All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions
#GP(0) If offset in target operand, call gate, or TSS is beyond the code segment limits.
If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If target mode is compatibility mode and SSP is not in low 4GB.
#GP(selector) If the segment selector index is outside descriptor table limits.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
If the DPL for a nonconforming-code segment is not equal to the CPL
(When not using a call gate.) If the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS’s segment selector.
If the segment descriptor for selector in a call gate does not indicate it is a code segment.
If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.
If the segment selector for a TSS has its local/global bit set for local.
If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP (selector) If the code segment being accessed is not present.
If call gate, task gate, or TSS not present.

#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the target operand is beyond the code segment limits.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same as 64-bit mode exceptions.

64-Bit Mode Exceptions
#GP(0) If a memory address is non-canonical.
If target offset in destination operand is non-canonical.
If target offset in destination operand is beyond the new code segment limit.
If the segment selector in the destination operand is NULL.
If the code segment selector in the 64-bit gate is NULL.
If transitioning to compatibility mode and the SSP is beyond 4GB.
#GP(selector) If the code segment or 64-bit call gate is outside descriptor table limits.
If the code segment or 64-bit call gate overlaps non-canonical space.
If the segment descriptor from a 64-bit call gate is in non-canonical space.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set.
If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment’s segment selector is greater than the CPL.
If the DPL for a conforming-code segment is greater than the CPL.
If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
If the upper type field of a 64-bit call gate is not 0x0.
If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
If the code segment descriptor pointed to by the selector in the 64-bit gate doesn’t have the L-bit set and the D-bit clear.
If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
If the code segment is non-conforming and CPL ≠ DPL.
If the code segment is confirming and CPL < DPL.

#NP(selector) If a code segment or 64-bit call gate is not present.

#UD (64-bit mode only) If a far jump is direct to an absolute address in memory.
If the LOCK prefix is used.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
### KADDW/KADDB/KADDQ/KADDD—ADD Two Masks

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.0F.W0 4A /r KADDW k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Add 16 bits masks in k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Add 8 bits masks in k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add 64 bits masks in k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 4A /r KADDD k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add 32 bits masks in k2 and k3 and place result in k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

#### Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

#### Operation

**KADDW**

DEST[15:0] := SRC1[15:0] + SRC2[15:0]  
DEST[MAX_KL-1:16] := 0

**KADDB**

DEST[7:0] := SRC1[7:0] + SRC2[7:0]  
DEST[MAX_KL-1:8] := 0

**KADDQ**

DEST[63:0] := SRC1[63:0] + SRC2[63:0]  
DEST[MAX_KL-1:64] := 0

**KADDD**

DEST[31:0] := SRC1[31:0] + SRC2[31:0]  
DEST[MAX_KL-1:32] := 0

#### Intel C/C++ Compiler Intrinsic Equivalent

KADDW __m128i _mmask16 _kadd_mask16 (__m128i a, __m128i b);  
KADDB __m128 _kadd_mask8 (__m128 a, __m128 b);  
KADDQ __m128i _kadd_mask64 (__m128i a, __m128i b);  
KADDD __m128 _kadd_mask32 (__m128 a, __m128 b);

#### Flags Affected

None.

#### SIMD Floating-Point Exceptions

None.
Other Exceptions
See Table 2-63, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."
KANDw/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

### Description
Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

### Operation

**KANDw**

\[
\text{DEST}[15:0] := \text{SRC1}[15:0] \text{ BITWISE AND SRC2}[15:0] \\
\text{DEST}[\text{MAX}_K1-1:16] := 0
\]

**KANDB**

\[
\text{DEST}[7:0] := \text{SRC1}[7:0] \text{ BITWISE AND SRC2}[7:0] \\
\text{DEST}[\text{MAX}_K1-1:8] := 0
\]

**KANDQ**

\[
\text{DEST}[63:0] := \text{SRC1}[63:0] \text{ BITWISE AND SRC2}[63:0] \\
\text{DEST}[\text{MAX}_K1-1:64] := 0
\]

**KANDD**

\[
\text{DEST}[31:0] := \text{SRC1}[31:0] \text{ BITWISE AND SRC2}[31:0] \\
\text{DEST}[\text{MAX}_K1-1:32] := 0
\]

### Intel C/C++ Compiler Intrinsic Equivalent

KANDW __mmask16 _mm512_kand(__mmask16 a, __mmask16 b);

### Flags Affected
None.

### SIMD Floating-Point Exceptions
None.

### Other Exceptions
See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.0F.W0 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDNW k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.</td>
</tr>
<tr>
<td>KANDNB k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.0F.W1 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDNQ k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 42 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KANDND k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM:[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

**KANDNW**

\[
\text{DEST}[15:0] := (\text{BITWISE NOT SRC1}[15:0]) \text{ BITWISE AND SRC2}[15:0]
\]

\[
\text{DEST}[^{\text{MAX}_K1 - 1}:16] := 0
\]

**KANDNB**

\[
\text{DEST}[7:0] := (\text{BITWISE NOT SRC1}[7:0]) \text{ BITWISE AND SRC2}[7:0]
\]

\[
\text{DEST}[^{\text{MAX}_K1 - 1}:8] := 0
\]

**KANDNQ**

\[
\text{DEST}[63:0] := (\text{BITWISE NOT SRC1}[63:0]) \text{ BITWISE AND SRC2}[63:0]
\]

\[
\text{DEST}[^{\text{MAX}_K1 - 1}:64] := 0
\]

**KANDND**

\[
\text{DEST}[31:0] := (\text{BITWISE NOT SRC1}[31:0]) \text{ BITWISE AND SRC2}[31:0]
\]

\[
\text{DEST}[^{\text{MAX}_K1 - 1}:32] := 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

KANDNW _mmask16 _mm512_kandn(_mmask16 a, _mmask16 b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
KMOVW/KMOVB/KMOVQ/KMOVD—Move From and to Mask Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from k2/m16 and store the result in k1.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from k2/m8 and store the result in k1.</td>
</tr>
<tr>
<td>VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from k2/m64 and store the result in k1.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from k2/m32 and store the result in k1.</td>
</tr>
<tr>
<td>VEX.L0.0F.W0 91 /r KMOVW m16, k1</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from k1 and store the result in m16.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 91 /r KMOVB m8, k1</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from k1 and store the result in m8.</td>
</tr>
<tr>
<td>VEX.L0.0F.W1 91 /r KMOVQ m64, k1</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from k1 and store the result in m64.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 91 /r KMOVD m32, k1</td>
<td>MR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from k1 and store the result in m32.</td>
</tr>
<tr>
<td>VEX.L0.0F.W0 92 /r KMOVW k1, r32</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from r32 to k1.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 92 /r KMOVB k1, r32</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from r32 to k1.</td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64</td>
<td>RR</td>
<td>V/I</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from r64 to k1.</td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from r32 to k1.</td>
</tr>
<tr>
<td>VEX.L0.0F.W0 93 /r KMOVW r32, k1</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 16 bits mask from k1 to r32.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 93 /r KMOVB r32, k1</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Move 8 bits mask from k1 to r32.</td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1</td>
<td>RR</td>
<td>V/I</td>
<td>AVX512BW</td>
<td>Move 64 bits mask from k1 to r64.</td>
</tr>
<tr>
<td>VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Move 32 bits mask from k1 to r32.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (w, ModRM[7:6] must not be 11b)</td>
<td>ModRM:reg (r)</td>
</tr>
<tr>
<td>RR</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.
When moving to a mask register, the result is zero extended to MAX_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination’s size is 32 bits. In 64-bit mode, the default GPR destination’s size is 64 bits. Note that VEX.W can only be used to modify the size of the GPR operand in 64b mode.

**Operation**

**KMOVW**
- If *destination is a memory location*
  \[ \text{DEST}[15:0] := \text{SRC}[15:0] \]
- If *destination is a mask register or a GPR*
  \[ \text{DEST} := \text{ZeroExtension}(\text{SRC}[15:0]) \]

**KMOVB**
- If *destination is a memory location*
  \[ \text{DEST}[7:0] := \text{SRC}[7:0] \]
- If *destination is a mask register or a GPR*
  \[ \text{DEST} := \text{ZeroExtension}(\text{SRC}[7:0]) \]

**KMOVQ**
- If *destination is a memory location or a GPR*
  \[ \text{DEST}[63:0] := \text{SRC}[63:0] \]
- If *destination is a mask register*
  \[ \text{DEST} := \text{ZeroExtension}(\text{SRC}[63:0]) \]

**KMOVD**
- If *destination is a memory location*
  \[ \text{DEST}[31:0] := \text{SRC}[31:0] \]
- If *destination is a mask register or a GPR*
  \[ \text{DEST} := \text{ZeroExtension}(\text{SRC}[31:0]) \]

**Intel C/C++ Compiler Intrinsic Equivalent**

KMOVW __mmask16 _mm512_kmov(__mmask16 a);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Instructions with RR operand encoding, see Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”

Instructions with RM or MR operand encoding, see Table 2-64, “TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory).”
KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.FW.0 44 /r  KNOTW k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise NOT of 16 bits mask k2.</td>
</tr>
<tr>
<td>VEX.L0.66.FW.0 44 /r  KNOTB k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise NOT of 8 bits mask k2.</td>
</tr>
<tr>
<td>VEX.L0.FW.1 44 /r  KNOTQ k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise NOT of 64 bits mask k2.</td>
</tr>
<tr>
<td>VEX.L0.66.FW.1 44 /r  KNOTD k1, k2</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise NOT of 32 bits mask k2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM:[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

**Operation**

**KNOTW**

DEST[15:0] := BITWISE NOT SRC[15:0]  
DEST[MAX_KL-1:16] := 0

**KNOTB**

DEST[7:0] := BITWISE NOT SRC[7:0]  
DEST[MAX_KL-1:8] := 0

**KNOTQ**

DEST[63:0] := BITWISE NOT SRC[63:0]  
DEST[MAX_KL-1:64] := 0

**KNOTD**

DEST[31:0] := BITWISE NOT SRC[31:0]  
DEST[MAX_KL-1:32] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KNOTW __mmask16 __mm512_knot(__mmask16 a);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.0F.W0 45 /r KORW k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise OR 16 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise OR 8 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.0F.W1 45 /r KORQ k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise OR 64 bits masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise OR 32 bits masks k2 and k3 and place result in k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM:[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description
Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

**KORW**
```
DEST[15:0] := SRC1[15:0] BITWISE OR SRC2[15:0]
DEST[MAX_KL-1:16] := 0
```

**KORB**
```
DEST[7:0] := SRC1[7:0] BITWISE OR SRC2[7:0]
DEST[MAX_KL-1:8] := 0
```

**KORQ**
```
DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAX_KL-1:64] := 0
```

**KORD**
```
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[MAX_KL-1:32] := 0
```

Intel C/C++ Compiler Intrinsic Equivalent
```
KORW __mmask16 __mm512_kor(__mmask16 a, __mmask16 b);
```

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks and Set Flags

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 98 /r KORTESTW k1, k2</td>
<td>RR V/V</td>
<td></td>
<td>AVX512F</td>
<td>Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2</td>
<td>RR V/V</td>
<td></td>
<td>AVX512DQ</td>
<td>Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>VEX.L0.0F.W1 98 /r KORTESTQ k1, k2</td>
<td>RR V/V</td>
<td></td>
<td>AVX512BW</td>
<td>Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2</td>
<td>RR V/V</td>
<td></td>
<td>AVX512BW</td>
<td>Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1’s.

**Operation**

**KORTESTW**


IF(TMP[15:0]=0)
    THEN ZF := 1
    ELSE ZF := 0

FI;

IF(TMP[15:0]=FFFFh)
    THEN CF := 1
    ELSE CF := 0

FI;

**KORTESTB**

TMP[7:0] := DEST[7:0] BITWISE OR SRC[7:0]

IF(TMP[7:0]=0)
    THEN ZF := 1
    ELSE ZF := 0

FI;

IF(TMP[7:0]=FFh)
    THEN CF := 1
    ELSE CF := 0

FI;
KORTESTQ
TMP[63:0] := DEST[63:0] BITWISE OR SRC[63:0]
IF(TMP[63:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[63:0]==FFFFFFFF_FFFFFFFFh)
    THEN CF := 1
    ELSE CF := 0
FI;

KORTESTD
TMP[31:0] := DEST[31:0] BITWISE OR SRC[31:0]
IF(TMP[31:0]=0)
    THEN ZF := 1
    ELSE ZF := 0
FI;
IF(TMP[31:0]=FFFFFFFFh)
    THEN CF := 1
    ELSE CF := 0
FI;

Intel C/C++ Compiler Intrinsic Equivalent
KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);

Flags Affected
The ZF flag is set if the result of OR-ing both sources is all 0s.
The CF flag is set if the result of OR-ing both sources is all 1s.
The OF, SF, AF, and PF flags are set to 0.

Other Exceptions
See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.66.0F3A.W1 32/r KSHIFTLW k1, k2, imm8</td>
<td>RRI V/V</td>
<td>AVX512F</td>
<td>Shift left 16 bits in k2 by immediate and write result in k1.</td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W0 32/r KSHIFTLB k1, k2, imm8</td>
<td>RRI V/V</td>
<td>AVX512DQ</td>
<td>Shift left 8 bits in k2 by immediate and write result in k1.</td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W1 32/r KSHIFTLQ k1, k2, imm8</td>
<td>RRI V/V</td>
<td>AVX512BW</td>
<td>Shift left 64 bits in k2 by immediate and write result in k1.</td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W0 32/r KSHIFTLD k1, k2, imm8</td>
<td>RRI V/V</td>
<td>AVX512BW</td>
<td>Shift left 32 bits in k2 by immediate and write result in k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**
Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

**Operation**

**KSHIFTLW**
COUNT := imm8[7:0]  
DEST[MAX_KL-1:0] := 0  
IF COUNT <=15  
THEN DEST[15:0] := SRC1[15:0] << COUNT;  
FI;

**KSHIFTLB**
COUNT := imm8[7:0]  
DEST[MAX_KL-1:0] := 0  
IF COUNT <=7  
THEN DEST[7:0] := SRC1[7:0] << COUNT;  
FI;

**KSHIFTLQ**
COUNT := imm8[7:0]  
DEST[MAX_KL-1:0] := 0  
IF COUNT <=63  
THEN DEST[63:0] := SRC1[63:0] << COUNT;  
FI;
KSHIFTLD
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=31
    THEN DEST[31:0] := SRC1[31:0] << COUNT;
FI;

Intel C/C++ Compiler Intrinsic Equivalent
Compiler auto generates KSHIFTLW when needed.

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
**KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.66.0F3A.W1 30 /r</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift right 16 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>KSHIFTRW k1, k2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W0 30 /r</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Shift right 8 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>KSHIFTRB k1, k2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W1 31 /r</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Shift right 64 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>KSHIFTRQ k1, k2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F3A.W0 31 /r</td>
<td>RRI</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Shift right 32 bits in k2 by immediate and write result in k1.</td>
</tr>
<tr>
<td>KSHIFTRD k1, k2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r, ModRM[7:6] must be 11b)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

**Operation**

**KSHIFTRW**

COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=15
    THEN DEST[15:0] := SRC1[15:0] >> COUNT;
FI;

**KSHIFTRB**

COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=7
    THEN DEST[7:0] := SRC1[7:0] >> COUNT;
FI;

**KSHIFTRQ**

COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=63
    THEN DEST[63:0] := SRC1[63:0] >> COUNT;
FI;
KSHIFTRD
COUNT := imm8[7:0]
DEST[MAX_KL-1:0] := 0
IF COUNT <=31
    THEN DEST[31:0] := SRC1[31:0] >> COUNT;
FI;

Intel C/C++ Compiler Intrinsic Equivalent
Compiler auto generates KSHIFTRW when needed.

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-63, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."
KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L0.0F.W0 99 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.</td>
</tr>
<tr>
<td>KTESTW k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W0 99 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.</td>
</tr>
<tr>
<td>KTESTB k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.0F.W1 99 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.</td>
</tr>
<tr>
<td>KTESTQ k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L0.66.0F.W1 99 /r</td>
<td>RR</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.</td>
</tr>
<tr>
<td>KTESTD k1, k2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r, ModRM:7:6 must be 11b)</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**KTESTW**

IF (TEMP[15:0] = = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
FI;
IF (TEMP[15:0] = = 0)
    THEN CF := 1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

**KTESTB**

TEMP[7:0] := SRC2[7:0] AND SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
FI;
TEMP[7:0] := SRC2[7:0] AND NOT SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN CF := 1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;
KTESTQ
TEMP[63:0] := SRC2[63:0] AND SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
FI;
TEMP[63:0] := SRC2[63:0] AND NOT SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN CF := 1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

KTESTD
TEMP[31:0] := SRC2[31:0] AND SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
FI;
TEMP[31:0] := SRC2[31:0] AND NOT SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN CF := 1;
    ELSE CF := 0;
FI;
AF := OF := PF := SF := 0;

Intel C/C++ Compiler Intrinsic Equivalent

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
**KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpack 8-bit masks in k2 and k3 and write word result in k1.</td>
</tr>
<tr>
<td>VEX.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Unpack 16-bit masks in k2 and k3 and write doubleword result in k1.</td>
</tr>
<tr>
<td>VEX.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Unpack 32-bit masks in k2 and k3 and write quadword result in k1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RVR</td>
</tr>
</tbody>
</table>

**Description**

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

**Operation**

**KUNPCKBW**

DEST[7:0] := SRC2[7:0]
DEST[15:8] := SRC1[7:0]
DEST[MAX_KL-1:16] := 0

**KUNPCKWD**

DEST[15:0] := SRC2[15:0]
DEST[31:16] := SRC1[15:0]
DEST[MAX_KL-1:32] := 0

**KUNPCKDQ**

DEST[31:0] := SRC2[31:0]
DEST[63:32] := SRC1[31:0]
DEST[MAX_KL-1:64] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KUNPCKBW __m128i _mm512_kunpackb(__m128i a, __m128i b);
KUNPCKDQ __m64 _mm512_kunpackd(__m64 a, __m64 b);
KUNP CKwD __m512 _mm512_kun packw(__m512 a, __m512 b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-63, "TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)."
KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.0F.W0 46 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XNOR 16-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXNORW k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 46 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise XNOR 8-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXNORB k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.0F.W1 46 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise XNOR 64-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXNORQ k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 46 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise XNOR 32-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXNORD k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM:[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

**KXNORW**

\[
\text{DEST}[15:0] := \text{NOT} (\text{SRC1}[15:0] \text{BITWISE XOR SRC2}[15:0])
\]

\[
\text{DEST}[\text{MAX}_\text{KL}-1:16] := 0
\]

**KXNORB**

\[
\text{DEST}[7:0] := \text{NOT} (\text{SRC1}[7:0] \text{BITWISE XOR SRC2}[7:0])
\]

\[
\text{DEST}[\text{MAX}_\text{KL}-1:8] := 0
\]

**KXNORQ**

\[
\text{DEST}[63:0] := \text{NOT} (\text{SRC1}[63:0] \text{BITWISE XOR SRC2}[63:0])
\]

\[
\text{DEST}[\text{MAX}_\text{KL}-1:64] := 0
\]

**KXNORD**

\[
\text{DEST}[31:0] := \text{NOT} (\text{SRC1}[31:0] \text{BITWISE XOR SRC2}[31:0])
\]

\[
\text{DEST}[\text{MAX}_\text{KL}-1:32] := 0
\]

Intel C/C++ Compiler Intrinsic Equivalent

KXNORW __mmask16 _mm512_kxnor(__mmask16 a, __mmask16 b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.L1.0F.W0 47 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XOR 16-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXORW k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W0 47 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Bitwise XOR 8-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXORB k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.0F.W1 47 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise XOR 64-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXORQ k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.L1.66.0F.W1 47 /r</td>
<td>RVR</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise XOR 32-bit masks k2 and k3 and place result in k1.</td>
</tr>
<tr>
<td>KXORD k1, k2, k3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVR</td>
<td>ModRM:reg (w)</td>
<td>VEX.1vvv (r)</td>
<td>ModRM:r/m (r, ModRM:[7:6] must be 11b)</td>
</tr>
</tbody>
</table>

Description
Perform a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation
KXORW
DEST[15:0] := SRC1[15:0] BITWISE XOR SRC2[15:0]
DEST[MAX_KL-1:16] := 0

KXORB
DEST[7:0] := SRC1[7:0] BITWISE XOR SRC2[7:0]
DEST[MAX_KL-1:8] := 0

KXORQ
DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAX_KL-1:64] := 0

KXORD
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[MAX_KL-1:32] := 0

Intel C/C++ Compiler Intrinsic Equivalent
KXORW __mmask16 __mm512_kxor(__mmask16 a, __mmask16 b);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-63, “TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg).”
LAHF—Load Status Flags Into AH Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
</table>

NOTES:
1. Valid in specific stepping; see Description section.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation
IF 64-Bit Mode
  THEN
    IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
       THEN AH := RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
       ELSE #UD;
       ELSE
         AH := EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
     FI;
  ELSE
    FI;

Flags Affected
None. The state of the flags in the EFLAGS register is not affected.

Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.
If the LOCK prefix is used.
**LAR—Load Access Rights Byte**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 02 /r</td>
<td>LAR r16, r16/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 := access rights referenced by r16/m16</td>
</tr>
<tr>
<td>0F 02 /r</td>
<td>LAR reg, r32/m16</td>
<td>RM</td>
<td>Valid</td>
<td>reg := access rights referenced by r32/m16</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - Bits 19:16 are undefined.
  - Bit 20 returns the software-available bit in the descriptor.
  - Bit 21 returns the L flag.
  - Bit 22 returns the D/B flag.
  - Bit 23 returns the G flag.
  - Bits 31:24 are returned as 0.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-53.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.
The LAR instruction can only be executed in protected mode and IA-32e mode.

### Table 3-53. Segment and Gate Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Protected Mode</th>
<th>IA-32e Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Available 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>LDT</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Busy 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>16-bit call gate</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>16-bit/32-bit task gate</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>16-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>16-bit trap gate</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Available 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>A</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>Busy 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>32-bit call gate</td>
<td>Yes</td>
</tr>
<tr>
<td>D</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>32-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>F</td>
<td>32-bit trap gate</td>
<td>No</td>
</tr>
</tbody>
</table>

### Operation

If Offset(SRC) > descriptor table limit

THEN

ZF := 0;

ELSE

SegmentDescriptor := descriptor referenced by SRC;

IF SegmentDescriptor(Type) ≠ conforming code segment and (CPL > DPL) or (RPL > DPL)

or SegmentDescriptor(Type) is not valid for instruction

THEN

ZF := 0;

ELSE

DEST := access rights from SegmentDescriptor as given in Description section;

ZF := 1;

FI;

FI;

### Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while
the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The LAR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The LAR instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If the memory operand effective address referencing the SS segment is in a non-canonical
form.

#GP(0) If the memory operand effective address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while
the current privilege level is 3.

#UD If the LOCK prefix is used.
**LDDQU—Load Unaligned Integer 128 Bits**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F F0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE3</td>
<td>Load unaligned data from mem and return double quadword in xmm1.</td>
</tr>
<tr>
<td>LDDQU xmm1, mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.F2.0F.WIG F0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Load unaligned packed integer values from mem to xmm1.</td>
</tr>
<tr>
<td>VLDDQU xmm1, m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.F2.0F.WIG F0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Load unaligned packed integer values from mem to ymm1.</td>
</tr>
<tr>
<td>VLDDQU ymm1, m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The instruction is *functionally similar* to (V)MOVDQU ymm/xmm, m256/m128 for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

**Implementation Notes**

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Note:** In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**LDDQU (128-bit Legacy SSE Version)**

DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
VLDDQU (VEX.128 Encoded Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

VLDDQU (VEX.256 Encoded Version)
DEST[255:0] := SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent
LDDQU __m128i _mm_lddqu_si128 (__m128i * p);
VLDDQU __m256i _mm256_lddqu_si256 (__m256i * p);

Numeric Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions."
Note treatment of #AC varies.
LDMXCSR—Load MXCSR Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /2 LDMXCSR m32</td>
<td>M</td>
<td>V/V</td>
<td>SSE</td>
<td>Load MXCSR register from m32.</td>
</tr>
<tr>
<td>VEX.LZ.0F.W1G AE /2 VLDMXCSR m32</td>
<td>M</td>
<td>V/V</td>
<td>AVX</td>
<td>Load MXCSR register from m32.</td>
</tr>
</tbody>
</table>

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See “MXCSR Control and Status Register” in Chapter 10, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

• the instruction must operate on an XMM or YMM register operand,
• the instruction causes that particular SIMD floating-point exception to be reported.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

MXCSR := m32;

C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

Numeric Exceptions

None.

Other Exceptions

See Table 2-22, "Type 5 Class Exception Conditions,” additionally:

#GP For an attempt to set reserved bits in MXCSR.
#UD If VEX.vvvv ≠ 1111B.
LDS/LES/LFS/LGS/LSS—Load Far Pointer

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5 /r</td>
<td>LDS r16,m16:16</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Load DS:r16 with far pointer from memory.</td>
</tr>
<tr>
<td>C5 /r</td>
<td>LDS r32,m16:32</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Load DS:r32 with far pointer from memory.</td>
</tr>
<tr>
<td>0F B2 /r</td>
<td>LSS r16,m16:16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load SS:r16 with far pointer from memory.</td>
</tr>
<tr>
<td>0F B2 /r</td>
<td>LSS r32,m16:32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load SS:r32 with far pointer from memory.</td>
</tr>
<tr>
<td>REX + 0F B2 /r</td>
<td>LSS r64,m16:64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load SS:r64 with far pointer from memory.</td>
</tr>
<tr>
<td>C4 /r</td>
<td>LES r16,m16:16</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Load ES:r16 with far pointer from memory.</td>
</tr>
<tr>
<td>C4 /r</td>
<td>LES r32,m16:32</td>
<td>RM</td>
<td>Invalid</td>
<td>Valid</td>
<td>Load ES:r32 with far pointer from memory.</td>
</tr>
<tr>
<td>0F B4 /r</td>
<td>LFS r16,m16:16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FS:r16 with far pointer from memory.</td>
</tr>
<tr>
<td>0F B4 /r</td>
<td>LFS r32,m16:32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load FS:r32 with far pointer from memory.</td>
</tr>
<tr>
<td>REX + 0F B4 /r</td>
<td>LFS r64,m16:64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load FS:r64 with far pointer from memory.</td>
</tr>
<tr>
<td>0F B5 /r</td>
<td>LGS r16,m16:16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load GS:r16 with far pointer from memory.</td>
</tr>
<tr>
<td>0F B5 /r</td>
<td>LGS r32,m16:32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load GS:r32 with far pointer from memory.</td>
</tr>
<tr>
<td>REX + 0F B5 /r</td>
<td>LGS r64,m16:64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load GS:r64 with far pointer from memory.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

Operation

64-BIT_MODE

IF SS is loaded
THEN
  IF SegmentSelector = NULL and ( (RPL = 3) or (RPL ≠ 3 and RPL ≠ CPL) )
  THEN #GP(0);
  ELSE IF descriptor is in non-canonical space

3588 Vol. 2A LDS/LES/LFS/LGS/LSS—Load Far Pointer
THEN #GP(selector); FI;
ELSE IF Segment selector index is not within descriptor table limits
    or segment selector RPL ≠ CPL
    or access rights indicate nonwritable data segment
    or DPL ≠ CPL
    THEN #GP(selector); FI;
ELSE IF Segment marked not present
    THEN #SS(selector); FI;
FI;
SS := SegmentSelector(SRC);
SS := SegmentDescriptor([SRC]);
ELSE IF attempt to load DS, or ES
    THEN #UD;
ELSE IF FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and (RPL > DPL or CPL > DPL)
        THEN #GP(selector); FI;
    ELSE IF Segment marked not present
        THEN #NP(selector); FI;
    FI;
    SegmentRegister := SegmentSelector(SRC);
    SegmentRegister := SegmentDescriptor([SRC]);
FI;
ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister := NULLSelector;
        SegmentRegister(DescriptorValidBit) := 0; FI (* Hidden flag;
        not accessible by software *)
FI;
DEST := Offset(SRC);
PARENTED MODE OR COMPATIBILITY MODE;
IF SS is loaded
    THEN
        IF SegmentSelector = NULL
            THEN #GP(0);
        ELSE IF Segment selector index is not within descriptor table limits
            or segment selector RPL ≠ CPL
            or access rights indicate nonwritable data segment
            or DPL ≠ CPL
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS := SegmentSelector(SRC);
        SS := SegmentDescriptor([SRC]);
ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and (RPL > DPL or CPL > DPL)
        THEN #GP(selector); FI;
    ELSE IF Segment marked not present

THEN #NP(selector); Fl;
  Fl;
  SegmentRegister := SegmentSelector(SRC) AND RPL;
  SegmentRegister := SegmentDescriptor([SRC]);
  Fl;
ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
  THEN
  SegmentRegister := NULLSelector;
  SegmentRegister(DescriptorValidBit) := 0; Fl; (* Hidden flag: not accessible by software *)
  Fl;
  DEST := Offset(SRC); 
Real-Address or Virtual-8086 Mode
  SegmentRegister := SegmentSelector(SRC); Fl;
  DEST := Offset(SRC);

Flags Affected

None.

Protected Mode Exceptions

#UD If source operand is not a memory location.
If the LOCK prefix is used.
#GP(0) If a NULL selector is loaded into the SS register.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector) If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL.
If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#SS(selector) If the SS register is being loaded and the segment is marked not present.
#NP(selector) If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If source operand is not a memory location.
If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#UD If source operand is not a memory location.
   If the LOCK prefix is used.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
   If a NULL selector is attempted to be loaded into the SS register in compatibility mode.
   If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode.
   If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.

#GP(Selector) If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
   If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.

#SS(0) If a memory operand effective address is non-canonical
#SS(Selector) If the SS register is being loaded and the segment is marked not present.
#NP(selector) If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment fault occurs.
#UD If source operand is not a memory location.
   If the LOCK prefix is used.


## LDTILECFG—Load Tile Configuration

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.NP.0F38.W0 49 (11):000:bbb LDTILECFG m512</td>
<td>A</td>
<td>V/N.E.</td>
<td>AMX-TILE</td>
<td>Load tile configuration as specified in m512.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The LDTILECFG instruction takes an operand containing a pointer to a 64-byte memory location containing the description of the tiles to be supported. In order to configure the tiles, the AMX-TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

The memory area contains the palette and describes how many tiles are being used and defines each tile in terms of rows and column bytes. Requests must be compatible with the restrictions provided by CPUID; see Table 3-10 below.

### Table 3-10. Memory Area Layout

<table>
<thead>
<tr>
<th>Byte(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>palette</td>
<td>Palette selects the supported configuration of the tiles that will be used.</td>
</tr>
<tr>
<td>1</td>
<td>start_row</td>
<td>start_row is used for storing the restart values for interrupted operations.</td>
</tr>
<tr>
<td>2-15</td>
<td>reserved, must be zero</td>
<td></td>
</tr>
<tr>
<td>16-17</td>
<td>tile0.colsb</td>
<td>Tile 0 bytes per row.</td>
</tr>
<tr>
<td>18-19</td>
<td>tile1.colsb</td>
<td>Tile 1 bytes per row.</td>
</tr>
<tr>
<td>20-21</td>
<td>tile2.colsb</td>
<td>Tile 2 bytes per row.</td>
</tr>
<tr>
<td>...</td>
<td>(sequence continues)</td>
<td></td>
</tr>
<tr>
<td>30-31</td>
<td>tile7.colsb</td>
<td>Tile 7 bytes per row.</td>
</tr>
<tr>
<td>32-47</td>
<td>reserved, must be zero</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>tile0.rows</td>
<td>Tile 0 rows.</td>
</tr>
<tr>
<td>49</td>
<td>tile1.rows</td>
<td>Tile 1 rows.</td>
</tr>
<tr>
<td>50</td>
<td>tile2.rows</td>
<td>Tile 2 rows.</td>
</tr>
<tr>
<td>...</td>
<td>(sequence continues)</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>tile7.rows</td>
<td>Tile 7 rows.</td>
</tr>
<tr>
<td>56-63</td>
<td>reserved, must be zero</td>
<td></td>
</tr>
</tbody>
</table>

If a tile row and column pair is not used to specify tile parameters, they must have the value zero. All enabled tiles (based on the palette) must be configured. Specifying tile parameters for more tiles than the implementation limit or the palette limit results in a #GP fault.

If the palette_id is zero, that signifies the INIT state for both TILECFG and TILEDATA. Tiles are zeroed in the INIT state. The only legal non-INIT value for palette_id is 1.

Any attempt to execute the LDTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.
Operation
LDTILECFG mem
error := False
buf := read_memory(mem, 64)
temp_tilecfg.palette_id := buf.byte[0]
if temp_tilecfg.palette_id > max_palette:
    error := True
if not xcr0_supports_palette(temp_tilecfg.palette_id):
    error := True
if temp_tilecfg.palette_id != 0:
    temp_tilecfg.start_row := buf.byte[1]
    if buf.byte[2..15] is nonzero:
        error := True

p := 16
# configure columns
for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
    temp_tilecfg.t[n].colsb := buf.word[p/2]
    p := p + 2
    if temp_tilecfg.t[n].colsb > palette_table[temp_tilecfg.palette_id].bytes_per_row:
        error := True
if nonzero(buf[p..47]):
    error := True

# configure rows
p := 48
for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
    temp_tilecfg.t[n].rows := buf.byte[p]
    if temp_tilecfg.t[n].rows > palette_table[temp_tilecfg.palette_id].max_rows:
        error := True
    p := p + 1
if nonzero(buf[p..63]):
    error := True

# validate each tile's row & col configs are reasonable and enable the valid tiles
for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
    if temp_tilecfg.t[n].rows != 0 and temp_tilecfg.t[n].colsb != 0:
        temp_tilecfg.t[n].valid := 1
    elif temp_tilecfg.t[n].rows == 0 and temp_tilecfg.t[n].colsb == 0:
        temp_tilecfg.t[n].valid := 0
    else:
        error := True
        // one of rows or colsbwas 0 but not both.

if error:
    #GP
    elif temp_tilecfg.palette_id == 0:
        TILES_CONFIGURED := 0 // init state
tilecfg := 0 // equivalent to 64B of zeros
        zero_all_tile_data()
    else:
        tilecfg := temp_tilecfg
        zero_all_tile_data()
        TILES_CONFIGURED := 1
Intel C/C++ Compiler Intrinsic Equivalent
LDTILECFG void _tile_loadconfig(const void *);

Flags Affected
None.

Exceptions
AMX-E1; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.
LEA—Load Effective Address

**Op/En** Instruction **Op**/ **64-Bit** **Compat/** **Description**
8D /r LEA r16,m RM Valid Valid Store effective address for m in register r16.
8D /r LEA r32,m RM Valid Valid Store effective address for m in register r32.
REX.W + 8D /r LEA r64,m RM Valid N.E. Store effective address for m in register r64.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

**Table 3-54. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes**

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Address Size</th>
<th>Action Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>16-bit effective address is calculated and stored in requested 16-bit register destination.</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32-bit effective address is calculated and stored in the requested 32-bit register destination.</td>
</tr>
</tbody>
</table>

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction’s destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-55.

**Table 3-55. 64-bit Mode LEA Operation with Address and Operand Size Attributes**

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Address Size</th>
<th>Action Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>32</td>
<td>32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).</td>
</tr>
</tbody>
</table>
Operation
IF OperandSize = 16 and AddressSize = 16
    THEN
        DEST := EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 and AddressSize = 32
        THEN
            temp := EffectiveAddress(SRC); (* 32-bit address *)
            DEST := temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 16
        THEN
            temp := EffectiveAddress(SRC); (* 16-bit address *)
            DEST := ZeroExtend(temp); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 32
        THEN
            DEST := EffectiveAddress(SRC); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 16 and AddressSize = 64
        THEN
            temp := EffectiveAddress(SRC); (* 64-bit address *)
            DEST := temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 64
        THEN
            temp := EffectiveAddress(SRC); (* 64-bit address *)
            DEST := temp[0:31]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 64 and AddressSize = 64
        THEN
            DEST := EffectiveAddress(SRC); (* 64-bit address *)
        FI;
    FI;

Flags Affected
None.

Protected Mode Exceptions
#UD If source operand is not a memory location.
    If the LOCK prefix is used.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
LEAVE—High Level Procedure Exit

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C9</td>
<td>LEAVE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set SP to BP, then pop BP.</td>
</tr>
<tr>
<td>C9</td>
<td>LEAVE</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Set ESP to EBP, then pop EBP.</td>
</tr>
<tr>
<td>C9</td>
<td>LEAVE</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set RSP to RBP, then pop RBP.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure’s stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See “Procedure Calls for Block-Structured Languages” in Chapter 7 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction’s default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF StackAddressSize = 32
THEN
    ESP := EBP;
ELSE IF StackAddressSize = 64
    THEN RSP := RBP; FI;
ELSE IF StackAddressSize = 16
    THEN SP := BP; FI;
FI;

IF OperandSize = 32
THEN EBP := Pop();
ELSE IF OperandSize = 64
    THEN RBP := Pop(); FI;
ELSE IF OperandSize = 16
    THEN BP := Pop(); FI;
FI;

**Flags Affected**

None.
Protected Mode Exceptions

#SS(0) If the EBP register points to a location that is not within the limits of the current stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If the stack address is in a non-canonical form.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
LFENCE—Load Fence

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE E8 LFENCE</td>
<td>ZO</td>
<td>V/V</td>
<td>SSE2</td>
<td>Serializes load operations.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete before the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute (even speculatively) until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

**Operation**

Wait_On_Following_Instructions_Until(preceding_instructions_complete);

**Intel C/C++ Compiler Intrinsic Equivalent**

void _mm_lfence(void)

**Exceptions (All Modes of Operation)**

#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.

If the LOCK prefix is used.
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /2</td>
<td>LGDT m16&amp;32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Load m into GDTR.</td>
</tr>
<tr>
<td>0F 01 /3</td>
<td>LIDT m16&amp;32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Load m into IDTR.</td>
</tr>
<tr>
<td>0F 01 /2</td>
<td>LGDT m16&amp;64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load m into GDTR.</td>
</tr>
<tr>
<td>0F 01 /3</td>
<td>LIDT m16&amp;64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load m into IDTR.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction’s operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See “SGDT—Store Global Descriptor Table Register” in Chapter 4, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for information on storing the contents of the GDTR and IDTR.
Operation
IF Instruction is LIDT THEN
  IF OperandSize = 16 THEN
    IDTR(Limit) := SRC[0:15];
    IDTR(Base) := SRC[16:47] AND 00FFFFFFH;
  ELSE IF 32-bit Operand Size THEN
    IDTR(Limit) := SRC[0:15];
    IDTR(Base) := SRC[16:47];
  FI;
ELSE IF 64-bit Operand Size (* In 64-Bit Mode *) THEN
  IDTR(Limit) := SRC[0:15];
  IDTR(Base) := SRC[16:79];
FI;
ELSE (* Instruction is LGDT *)
  IF OperandSize = 16 THEN
    GDTR(Limit) := SRC[0:15];
    GDTR(Base) := SRC[16:47] AND 00FFFFFFH;
  ELSE IF 32-bit Operand Size THEN
    GDTR(Limit) := SRC[0:15];
    GDTR(Base) := SRC[16:47];
  FI;
ELSE IF 64-bit Operand Size (* In 64-Bit Mode *) THEN
  GDTR(Limit) := SRC[0:15];
  GDTR(Base) := SRC[16:79];
FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#UD If the LOCK prefix is used.
#GP(0) If the current privilege level is not 0.
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.
#GP If the current privilege level is not 0.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the current privilege level is not 0.
#PF(fault-code) If a page fault occurs.
If the memory address is in a non-canonical form.
#UD If the LOCK prefix is used.
LLDT—Load Local Descriptor Table Register

**Description**
Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

**Operation**
- IF SRC(Offset) > descriptor table limit  
  THEN #GP(segment selector); Fl;

- IF segment selector is valid  
  Read segment descriptor;
  IF SegmentDescriptor(Type) ≠ LDT  
  THEN #GP(segment selector); Fl;
  IF segment descriptor is not present  
  THEN #NP(segment selector); Fl;
  LDTR(SegmentSelector) := SRC;
  LDTR(SegmentDescriptor) := GDTSegmentDescriptor;
  ELSE LDTR := INVALID
  Fl;

**Flags Affected**
None.
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register contains a NULL segment selector.

#GP(selector) If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT
   is not a Local Descriptor Table.
   Segment selector is beyond GDT limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP(selector) If the LDT descriptor is not present.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The LLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The LLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the current privilege level is not 0.
   If the memory address is in a non-canonical form.

#GP(selector) If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT
   is not a Local Descriptor Table.
   Segment selector is beyond GDT limit.

#NP(selector) If the LDT descriptor is not present.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
LMSW—Load Machine Status Word

**Description**

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on IA-32 and Intel 64 processors beginning with Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

\[
CR0[0:3] := SRC[0:3];
\]

**Flags Affected**

None.

**Protected Mode Exceptions**

- **#GP(0)**  
  If the current privilege level is not 0.
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

- **#SS(0)**  
  If a memory operand effective address is outside the SS segment limit.

- **#PF(fault-code)**  
  If a page fault occurs.

- **#UD**  
  If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP**  
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

- **#UD**  
  If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
#GP(0)  The LMSW instruction is not recognized in virtual-8086 mode.
#UD    If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the current privilege level is not 0.
        If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#UD    If the LOCK prefix is used.
LOADIWKEY—Load Internal Wrapping Key With Key Locker

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 38 DC 11:rrr:bbb LOADIWKEY xmm1, xmm2, &lt;EAX&gt;, &lt;XMM0&gt;</td>
<td>A</td>
<td>V/V</td>
<td>KL</td>
<td>Load internal wrapping key from xmm1, xmm2, and XMM0.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM[reg (r)]</td>
<td>ModRM[r/m (r)]</td>
<td>Implicit EAX (r)</td>
<td>Implicit XMMO (r)</td>
</tr>
</tbody>
</table>

**Description**

The LOADIWKEY instruction writes the Key Locker internal wrapping key, which is called IWKey. This IWKey is used by the ENCODEKEY* instructions to wrap keys into handles. Conversely, the AESENC/DEC*KL instructions use IWKey to unwrap those keys from the handles and help verify the handle integrity. For security reasons, no instruction is designed to allow software to directly read the IWKey value.

IWKey includes two cryptographic keys as well as metadata. The two cryptographic keys are loaded from register sources so that LOADIWKEY can be executed without the keys ever being in memory.

The key input operands are:

- The 256-bit encryption key is loaded from the two explicit operands.
- The 128-bit integrity key is loaded from the implicit operand XMM0.

The implicit operand EAX specifies the KeySource and whether backing up the key is permitted:

- EAX[0] – When set, the wrapping key being initialized is not permitted to be backed up to platform-scoped storage.
- EAX[4:1] – This specifies the KeySource, which is the type of key. Currently only two encodings are supported. A KeySource of 0 indicates that the key input operands described above should be directly stored as the internal wrapping keys. LOADIWKEY with a KeySource of 1 will have random numbers from the on-chip random number generator XORed with the source registers (including XMM0) so that the software that executes the LOADIWKEY does not know the actual IWKey encryption and integrity keys. Software can choose to put additional random data into the source registers so that other sources of random data are combined with the hardware random number generator supplied value. Software should always check ZF after executing LOADIWKEY with KeySource of 1 as this operation may fail due to it being unable to get sufficient full-entropy data from the on-chip random number generator. Both KeySource of 0 and 1 specify that IWKey be used with the AES-GCM-SIV algorithm. CPUID.19H.ECX[1] enumerates support for KeySource of 1. All other KeySource encodings are reserved.

1. Further details on Key Locker and usage of this instruction can be found here: [https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html](https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html)
Operation
LOADIWKEY
IF CPL > 0 // LOADIWKEY only allowed at ring 0 (supervisor mode)
    THEN #GP (0); FI;
IF EAX[4:1] > 1 // Reserved KeySource encoding used
    THEN #GP (0); FI;
IF EAX[31:5] != 0 // Reserved bit in EAX is set
    THEN #GP (0); FI;
IF EAX[0] AND (CPUID.19H.ECX[0] == 0) // NoBackup is not supported on this part
    THEN #GP (0); FI;
IF (EAX[4:1] == 1) AND (CPUID.19H.ECX[1] == 0) // KeySource of 1 is not supported on this part
    THEN #GP (0); FI;
IF (EAX[4:1] == 0) // KeySource of 0
    THEN
        IWKey.Encryption Key[127:0] := SRC2[127:0];
        IWKey.Encryption Key[255:128] := SRC1[127:0];
        IWKey.IntegrityKey[127:0] := XMM0[127:0];
        IWKey.NoBackup = EAX[0];
        IWKey.KeySource = EAX[4:1];
        RFLAGS.ZF := 0;
    ELSE // KeySource of 1. See RDSEED definition for details of randomness
        IF HW_NRND_GEN.ready == 1 // Full-entropy random data from RDSEED hardware block was received
            THEN
                IWKey.Encryption Key[127:0] := SRC2[127:0] XOR HW_NRND_GEN.data[127:0];
                IWKey.IntegrityKey[127:0] := XMM0[127:0] XOR HW_NRND_GEN.data[383:256];
                IWKey.NoBackup = EAX[0];
                IWKey.KeySource = EAX[4:1];
                RFLAGS.ZF := 0;
            ELSE // Random data was not returned from RDSEED hardware block. IWKey was not loaded
                RFLAGS.ZF := 1;
        FI;
    FI;
    RFLAGS.OF, SF, AF, PF, CF := 0;

Flags Affected
ZF is set to 0 if the operation succeeded and set to 1 if the operation failed due to full-entropy random data not being received from RDSEED. The other arithmetic flags (OF, SF, AF, PF, CF) are cleared to 0.

Intel C/C++ Compiler Intrinsic Equivalent
LOADIWKEY void _mm_loadiwkey(unsigned int ctl, __m128i intkey, __m128i enkey_lo, __m128i enkey_hi);
Exceptions (All Operating Modes)

#GP          If CPL > 0. (Does not apply in real-address mode.)
             If EAX[31:5] != 0.
             If (EAX[0] == 1) AND (CPUID.19H.ECX[0] == 0).
#UD          If the LOCK prefix is used.
             If CPUID.07H:ECX.KL[bit 23] = 0.
             If CR4.KL = 0.
             If CR0.EM = 1.
             If CR4.OSFXSR = 0.
#NM          If CR0.TS = 1.
LOCK—Assert LOCK# Signal Prefix

<table>
<thead>
<tr>
<th>Opcode¹</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/A Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>LOCK</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Asserts LOCK# signal for duration of the accompanying instruction.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**
Causes the processor’s LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the “IA-32 Architecture Compatibility” section below for more details.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**
Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor’s cache is locked. Here, the processor’s cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See “Effects of a Locked Operation on Internal Processor Caches” in Chapter 9 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on locking of caches.

**Operation**
AssertLOCK#(DurationOfAccompanyingInstruction);

**Flags Affected**
None.

**Protected Mode Exceptions**

#UD If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.

Other exceptions can be generated by the instruction when the LOCK prefix is applied.
Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
LODS/LODSB/LODSW/LODSD/LODSQ—Load String

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>LODS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (RS)I into AL.</td>
</tr>
<tr>
<td>AD</td>
<td>LODS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (RS)I into AX.</td>
</tr>
<tr>
<td>AD</td>
<td>LODS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (RS)I into EAX.</td>
</tr>
<tr>
<td>REX.W + AD</td>
<td>LODS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load qword at address (RS)I into RAX.</td>
</tr>
<tr>
<td>AC</td>
<td>LODSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (RS)I into AL.</td>
</tr>
<tr>
<td>AD</td>
<td>LODSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (RS)I into AX.</td>
</tr>
<tr>
<td>AD</td>
<td>LODSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (RS)I into EAX.</td>
</tr>
<tr>
<td>REX.W + AD</td>
<td>LODSQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load qword at address (RS)I into RAX.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.
In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for a description of the REP prefix.

### Operation

If AL := SRC; (* Byte load *)

THEN AL := SRC; (* Byte load *)

IF DF = 0

THEN (E)SI := (E)SI + 1;
ELSE (E)SI := (E)SI - 1;

FI;

ELSE IF AX := SRC; (* Word load *)

THEN IF DF = 0

THEN (E)SI := (E)SI + 2;
ELSE (E)SI := (E)SI - 2;

IF;

FI;

ELSE IF EAX := SRC; (* Doubleword load *)

THEN IF DF = 0

THEN (E)SI := (E)SI + 4;
ELSE (E)SI := (E)SI - 4;

FI;

FI;

ELSE IF RAX := SRC; (* Quadword load *)

THEN IF DF = 0

THEN (R)SI := (R)SI + 8;
ELSE (R)SI := (R)SI - 8;

FI;

FI;

FI;

### Flags Affected
None.

### Protected Mode Exceptions

- **#GP(0)**  
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)**  
  If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**  
  If a page fault occurs.
- **#AC(0)**  
  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**  
  If the LOCK prefix is used.

### Real-Address Mode Exceptions

- **#GP**  
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS**  
  If a memory operand effective address is outside the SS segment limit.
- **#UD**  
  If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
### LOOP/LOOPcc—Loop According to ECX Counter

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2 cb</td>
<td>LOOP rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement count; jump short if count ≠ 0.</td>
</tr>
<tr>
<td>E1 cb</td>
<td>LOOPE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement count; jump short if count ≠ 0 and ZF = 1.</td>
</tr>
<tr>
<td>E0 cb</td>
<td>LOOPNE rel8</td>
<td>D</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement count; jump short if count ≠ 0 and ZF = 0.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Offset</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOPcc) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (cc) is associated with each instruction to indicate the condition being tested for. Here, the LOOPcc instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

#### Operation

IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;
Count := Count – 1;
IF Instruction is not LOOP
    THEN
        IF (Instruction := LOOPE) or (Instruction := LOOPZ)
            THEN IF (ZF = 1) and (Count ≠ 0)
                THEN BranchCond := 1;
                ELSE BranchCond := 0;
            FI;
            ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
                IF (ZF = 0 ) and (Count ≠ 0)
                    THEN BranchCond := 1;
                    ELSE BranchCond := 0;
                FI;
        FI;
FI;
ELSE (* Instruction = LOOP *)
    IF (Count ≠ 0)
        THEN BranchCond := 1;
        ELSE BranchCond := 0;
        FI;
FI;
FI;
IF BranchCond = 1
    THEN
        IF in 64-bit mode (* OperandSize = 64 *)
            THEN
                tempRIP := RIP + SignExtend(DEST);
                IF tempRIP is not canonical
                    THEN #GP(0);
                    ELSE RIP := tempRIP;
                    FI;
                ELSE
                    tempEIP := EIP + SignExtend(DEST);
                    IF OperandSize ≠ 16
                        THEN tempEIP := tempEIP AND 0000FFFFH;
                        FI;
                    IF tempEIP is not within code segment limit
                        THEN #GP(0);
                        ELSE EIP := tempEIP;
                        FI;
                    FI;
        ELSE
            Terminate loop and continue program execution at (R/E)IP;
        FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the offset being jumped to is beyond the limits of the CS segment.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the offset being jumped to is in a non-canonical form.
#UD If the LOCK prefix is used.
**LSL—Load Segment Limit**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 03 /r</td>
<td>LSL r16, r16/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load: r16 := segment limit, selector r16/m16.</td>
</tr>
<tr>
<td>0F 03 /r</td>
<td>LSL r32, r32/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load: r32 := segment limit, selector r32/m16.</td>
</tr>
<tr>
<td>REX.W + 0F 03 /r</td>
<td>LSL r64, r32/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Load: r64 := segment limit, selector r32/m16</td>
</tr>
</tbody>
</table>

**NOTES:**
1. For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit "raw" limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed.
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.
Table 3-56. Segment and Gate Descriptor Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Protected Mode</th>
<th>IA-32e Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Available 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>LDT</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Busy 16-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>16-bit call gate</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>16-bit/32-bit task gate</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>16-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>16-bit trap gate</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>Available 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>A</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>Busy 32-bit TSS</td>
<td>Yes</td>
</tr>
<tr>
<td>C</td>
<td>32-bit call gate</td>
<td>No</td>
</tr>
<tr>
<td>D</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>E</td>
<td>32-bit interrupt gate</td>
<td>No</td>
</tr>
<tr>
<td>F</td>
<td>32-bit trap gate</td>
<td>No</td>
</tr>
</tbody>
</table>

NOTES:
1. In this case, the descriptor comprises 16 bytes; bits 12:8 of the upper 4 bytes must be 0.

Operation
IF SRC(Offset) > descriptor table limit
THEN ZF := 0; Fi;
Read segment descriptor;
IF SegmentDescriptor(Type) ≠ conforming code segment
and (CPL > DPL) OR (RPL > DPL)
or Segment type is not valid for instruction
THEN
    ZF := 0;
ELSE
    temp := SegmentLimit([SRC]);
    IF (SegmentDescriptor(G) = 1)
        THEN temp := (temp << 12) OR 00000FFFH;
    ELSE IF OperandSize = 32
        THEN DEST := temp; Fi;
    ELSE IF OperandSize = 64 (* REX.W used *)
        THEN DEST := temp(* Zero-extended *); Fi;
    ELSE (* OperandSize = 16 *)
        DEST := temp AND FFFFH;
    Fi;
Fi;

Flags Affected
The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0.
Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while
the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD The LSL instruction cannot be executed in real-address mode.

Virtual-8086 Mode Exceptions
#UD The LSL instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If the memory operand effective address referencing the SS segment is in a non-canonical
form.
#GP(0) If the memory operand effective address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and the memory operand effective address is unaligned while
the current privilege level is 3.
#UD If the LOCK prefix is used.
LTR—Load Task Register

Description
Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

Operation
IF SRC is a NULL selector
    THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
    THEN #GP(segment selector); Fl;

Read segment descriptor;

IF segment descriptor is not for an available TSS
    THEN #GP(segment selector); Fl;

IF segment descriptor is not present
    THEN #NP(segment selector); Fl;

TSSSegmentDescriptor(busy) := 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)

TaskRegister(SegmentSelector) := SRC;
TaskRegister(SegmentDescriptor) := TSSSegmentDescriptor;

Flags Affected
None.
Protected Mode Exceptions

#GP(0)  If the current privilege level is not 0.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the source operand contains a NULL segment selector.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#GP(selector)  If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
If the selector points to LDT or is beyond the GDT limit.

#NP(selector)  If the TSS descriptor is marked not present.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)  If a page fault occurs.

#UD  If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD  The LTR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD  The LTR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode, as well as the following:

#GP(selector)  If the source selector points to a 16-bit TSS.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)  If the current privilege level is not 0.
If the memory address is in a non-canonical form.
If the source operand contains a NULL segment selector.

#GP(selector)  If the source selector points to a segment that is not a TSS, to a 16-bit TSS, or to a TSS for a task that is already busy.
If the selector points to LDT or is beyond the GDT limit.
If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.

#NP(selector)  If the TSS descriptor is marked not present.

#PF(fault-code)  If a page fault occurs.

#UD  If the LOCK prefix is used.
**LZCNT—Count the Number of Leading Zero Bits**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F BD /r</td>
<td>RM</td>
<td>V/V</td>
<td>LZCNT</td>
<td>Count the number of leading zero bits in r/m16, return result in r16.</td>
</tr>
<tr>
<td>LZCNT r16, r/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F BD /r</td>
<td>RM</td>
<td>V/V</td>
<td>LZCNT</td>
<td>Count the number of leading zero bits in r/m32, return result in r32.</td>
</tr>
<tr>
<td>LZCNT r32, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 REX.W 0F BD /r</td>
<td>RM</td>
<td>V/N.E.</td>
<td>LZCNT</td>
<td>Count the number of leading zero bits in r/m64, return result in r64.</td>
</tr>
<tr>
<td>LZCNT r64, r/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR. In 64-bit mode 64-bit operand size requires REX.W=1.

**Operation**

\[
\begin{align*}
temp & := \text{OperandSize} - 1 \\
\text{DEST} & := 0 \\
\text{WHILE} (temp > = 0) \text{ AND (Bit(SRC, temp) = 0)} \\
\text{DO} & \\
& \quad temp := temp - 1 \\
& \quad \text{DEST} := \text{DEST} + 1 \\
\text{OD} & \\
\text{IF} \ \text{DEST} = \text{OperandSize} \\
& \quad \text{CF} := 1 \\
\text{ELSE} & \\
& \quad \text{CF} := 0 \\
\text{FI} & \\
\text{IF} \ \text{DEST} = 0 \\
& \quad \text{ZF} := 1 \\
\text{ELSE} & \\
& \quad \text{ZF} := 0 \\
\text{FI} & \\
\end{align*}
\]

**Flags Affected**

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF, and AF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

LZCNT unsigned __int32 _lzcnt_u32(unsigned __int32 src);
LZCNT unsigned __int64 _lzcnt_u64(unsigned __int64 src);
Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
    If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) For an illegal address in the SS segment.
#UD If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If LOCK prefix is used.
4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. The operation of the immediate control byte is common to these four string text processing instructions of SSE4.2. This section describes the common operations.

4.1.1 General Description

The operation of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM is defined by the combination of the respective opcode and the interpretation of an immediate control byte that is part of the instruction encoding.

The opcode controls the relationship of input bytes/words to each other (determines whether the inputs terminated strings or whether lengths are expressed explicitly) as well as the desired output (index or mask).

The imm8 control byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

The arithmetic flags (ZF, CF, SF, OF, AF, PF) are set as a result of these instructions. However, the meanings of the flags have been overloaded from their typical meanings in order to provide additional information regarding the relationships of the two inputs.

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The imm8 control byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the imm8 Control Byte consists of bit fields that control the following attributes:

- **Source data format** — Byte/word data element granularity, signed or unsigned elements.
- **Aggregation operation** — Encodes the mode of per-element comparison operation and the aggregation of per-element comparisons into an intermediate result.
- **Polarity** — Specifies intermediate processing to be performed on the intermediate result.
- **Output selection** — Specifies final operation to produce the output (depending on index or mask) from the intermediate result.
4.1.2 Source Data Format

Table 4-1. Source Data Format

<table>
<thead>
<tr>
<th>Imm8[1:0]</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Unsigned bytes</td>
<td>Both 128-bit sources are treated as packed, unsigned bytes.</td>
</tr>
<tr>
<td>01b</td>
<td>Unsigned words</td>
<td>Both 128-bit sources are treated as packed, unsigned words.</td>
</tr>
<tr>
<td>10b</td>
<td>Signed bytes</td>
<td>Both 128-bit sources are treated as packed, signed bytes.</td>
</tr>
<tr>
<td>11b</td>
<td>Signed words</td>
<td>Both 128-bit sources are treated as packed, signed words.</td>
</tr>
</tbody>
</table>

If the imm8 control byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the imm8 control byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

<table>
<thead>
<tr>
<th>Imm8[3:2]</th>
<th>Mode</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Equal any</td>
<td>The arithmetic comparison is “equal.”</td>
</tr>
<tr>
<td>01b</td>
<td>Ranges</td>
<td>Arithmetic comparison is “greater than or equal” between even indexed bytes/words of reg and each byte/word of reg/mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arithmetic comparison is “less than or equal” between odd indexed bytes/words of reg and each byte/word of reg/mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(reg/mem[m] &gt;= reg[n] for n = even, reg/mem[m] &lt;= reg[n] for n = odd)</td>
</tr>
<tr>
<td>10b</td>
<td>Equal each</td>
<td>The arithmetic comparison is “equal.”</td>
</tr>
<tr>
<td>11b</td>
<td>Equal ordered</td>
<td>The arithmetic comparison is “equal.”</td>
</tr>
</tbody>
</table>

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by “BoolRes[Reg/Mem element index, Reg element index].” Comparisons evaluating to “True” are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by imm8 control byte bits[3:2].
See Section 4.1.6 for a description of the overrideIfDataInvalid() function used in Table 4-3.

### Table 4-3. Aggregation Operation

<table>
<thead>
<tr>
<th>Mode</th>
<th>Pseudocode</th>
</tr>
</thead>
</table>
| Equal any                   | **UpperBound = imm8[0] ? 7 : 15;**  
|                             | **IntRes1 = 0;**  
|                             | **For j = 0 to UpperBound, j++**  
|                             | **For i = 0 to UpperBound, i++**  
|                             | **IntRes1[i] OR= overrideIfDataInvalid(BoolRes[j,i])**                                                                                                                                                     |
| Ranges                      | **UpperBound = imm8[0] ? 7 : 15;**  
|                             | **IntRes1 = 0;**  
|                             | **For j = 0 to UpperBound, j++**  
|                             | **For i = 0 to UpperBound, i+=2**  
|                             | **IntRes1[i] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))**                                                                                                  |
| Equal each                  | **UpperBound = imm8[0] ? 7 : 15;**  
|                             | **IntRes1 = 0;**  
|                             | **For i = 0 to UpperBound, i++**  
|                             | **IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])**                                                                                             |
| Equal ordered               | **UpperBound = imm8[0] ? 7 :15;**  
|                             | **IntRes1 = imm8[0] ? FFH : FFFFH**  
|                             | **For j = 0 to UpperBound, j++**  
|                             | **For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++**  
|                             | **IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])**                                                                                          |

### 4.1.4 Polarity

IntRes1 may then be further modified by performing a 1’s complement, according to the value of the imm8 control byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to “valid” reg/mem input elements are complemented (note that the definition of a valid input element is dependant on the specific opcode and is defined in each opcode’s description). The result of the possible negation is referred to as IntRes2.

### Table 4-4. Polarity

<table>
<thead>
<tr>
<th>Imm8[5:4]</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>Positive Polarity (+)</td>
<td>IntRes2 = IntRes1</td>
</tr>
<tr>
<td>01b</td>
<td>Negative Polarity (-)</td>
<td>IntRes2 = -1 XOR IntRes1</td>
</tr>
<tr>
<td>10b</td>
<td>Masked (+)</td>
<td>IntRes2 = IntRes1</td>
</tr>
</tbody>
</table>
4.1.5 Output Selection

For PCMPESTRI/PCMPISTRI, the imm8 control byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

Specifically for PCMPESTRM/PCMPISTRM, the imm8 control byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

4.1.6 Valid/Invalid Override of Comparisons

PCMPxSTRx instructions allow for the possibility that an end-of-string (EOS) situation may occur within the 128-bit packed data value (see the instruction descriptions below for details). Any data elements on either source that are determined to be past the EOS are considered to be invalid, and the treatment of invalid data within a comparison pair varies depending on the aggregation function being performed.

In general, the individual comparison result for each element pair BoolRes[i,j] can be forced true or false if one or more elements in the pair are invalid. See Table 4-7.

<table>
<thead>
<tr>
<th>xmm1 byte/word</th>
<th>xmm2/ m128 byte/word</th>
<th>Imm8[3:2] = 00b (equal any)</th>
<th>Imm8[3:2] = 01b (ranges)</th>
<th>Imm8[3:2] = 10b (equal each)</th>
<th>Imm8[3:2] = 11b (equal ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Force false</td>
<td>Force false</td>
<td>Force true</td>
<td>Force true</td>
</tr>
<tr>
<td>Invalid</td>
<td>Valid</td>
<td>Force false</td>
<td>Force false</td>
<td>Force false</td>
<td>Force true</td>
</tr>
<tr>
<td>Valid</td>
<td>Invalid</td>
<td>Force false</td>
<td>Force false</td>
<td>Force false</td>
<td>Force false</td>
</tr>
<tr>
<td>Valid</td>
<td>Valid</td>
<td>Do not force</td>
<td>Do not force</td>
<td>Do not force</td>
<td>Do not force</td>
</tr>
</tbody>
</table>
### 4.1.7 Summary of Imm8 Control byte

**Table 4-8. Summary of Imm8 Control Byte**

<table>
<thead>
<tr>
<th>Imm8</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>------0b</td>
<td>128-bit sources treated as 16 packed bytes.</td>
</tr>
<tr>
<td>------1b</td>
<td>128-bit sources treated as 8 packed words.</td>
</tr>
<tr>
<td>------0-b</td>
<td>Packed bytes/words are unsigned.</td>
</tr>
<tr>
<td>------1-b</td>
<td>Packed bytes/words are signed.</td>
</tr>
<tr>
<td>-----00-b</td>
<td>Mode is equal any.</td>
</tr>
<tr>
<td>-----01-b</td>
<td>Mode is ranges.</td>
</tr>
<tr>
<td>-----10-b</td>
<td>Mode is equal each.</td>
</tr>
<tr>
<td>-----11-b</td>
<td>Mode is equal ordered.</td>
</tr>
<tr>
<td>------0-b</td>
<td>IntRes1 is unmodified.</td>
</tr>
<tr>
<td>------1-b</td>
<td>IntRes1 is negated (1’s complement).</td>
</tr>
<tr>
<td>------0-b</td>
<td>Negation of IntRes1 is for all 16 (8) bits.</td>
</tr>
<tr>
<td>------1-b</td>
<td>Negation of IntRes1 is masked by reg/mem validity.</td>
</tr>
<tr>
<td>-----00-b</td>
<td>Index of the least significant, set, bit is used (regardless of corresponding input element validity). IntRes2 is returned in least significant bits of XMM0.</td>
</tr>
<tr>
<td>-----1-b</td>
<td>Index of the most significant, set, bit is used (regardless of corresponding input element validity). Each bit of IntRes2 is expanded to byte/word.</td>
</tr>
<tr>
<td>------0-b</td>
<td>This bit currently has no defined effect, should be 0.</td>
</tr>
<tr>
<td>------1-b</td>
<td>This bit currently has no defined effect, should be 0.</td>
</tr>
</tbody>
</table>
4.1.8 Diagram Comparison and Aggregation Process

![Diagram Comparison and Aggregation Process](image)

4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1, and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- **f0()**: A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.

  \[ f0(B, C, D) = (B \text{ AND } C) \text{ XOR } ((\text{NOT}(B) \text{ AND } D) \text{ XOR } (B \text{ AND } C) \text{ XOR } (B \text{ AND } D) \text{ XOR } (C \text{ AND } D)) \]

- **f1()**: A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.

  \[ f1(B, C, D) = B \text{ XOR } C \text{ XOR } D \]

- **f2()**: A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.

  \[ f2(B, C, D) = (B \text{ AND } C) \text{ XOR } (B \text{ AND } D) \text{ XOR } (C \text{ AND } D) \]

- **f3()**: A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as f1().

  \[ f3(B, C, D) = B \text{ XOR } C \text{ XOR } D \]

- **Ch()**: A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
INSTRUCTION SET REFERENCE, M-U

Ch(E,F,G) := (E AND F) XOR ((NOT E) AND G)

• Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
  \[ \text{Maj}(A,B,C) := (A \text{ AND } B) \text{ XOR } (A \text{ AND } C) \text{ XOR } (B \text{ AND } C) \]

ROR is rotate right operation
\[ (A \text{ ROR } N) := A[N-1:0] \text{ } || \text{ } A[\text{Width}-1:N] \]

ROL is rotate left operation
\[ (A \text{ ROL } N) := A \text{ ROR } (\text{Width}-N) \]

SHR is the right shift operation
\[ (A \text{ SHR } N) := \text{ZEROES}[N-1:0] \text{ } || \text{ } A[\text{Width}-1:N] \]

• \( \Sigma_0() \): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  \[ \Sigma_0(A) := (A \text{ ROR } 2) \text{ XOR } (A \text{ ROR } 13) \text{ XOR } (A \text{ ROR } 22) \]

• \( \Sigma_1() \): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  \[ \Sigma_1(E) := (E \text{ ROR } 6) \text{ XOR } (E \text{ ROR } 11) \text{ XOR } (E \text{ ROR } 25) \]

• \( \sigma_0() \): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  \[ \sigma_0(W) := (W \text{ ROR } 7) \text{ XOR } (W \text{ ROR } 18) \text{ XOR } (W \text{ SHR } 3) \]

• \( \sigma_1() \): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  \[ \sigma_1(W) := (W \text{ ROR } 17) \text{ XOR } (W \text{ ROR } 19) \text{ XOR } (W \text{ SHR } 10) \]

• \( K_i \): SHA1 Constants dependent on immediate \( i \).
  \[ K0 = 0x5A827999 \]
  \[ K1 = 0x6ED9EBA1 \]
  \[ K2 = 0x8F1BBCDC \]
  \[ K3 = 0xCA62C1D6 \]

4.3 INSTRUCTIONS (M-U)

MASKMOVQDU—Store Selected Bytes of Double Quadword

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F F7 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Selectively write bytes from xmm1 to memory location using the byte mask in xmm2. The default memory location is specified by DS:DI/EDI/RDI.</td>
</tr>
<tr>
<td>VEX.128.66.0F:WIG F7 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Selectively write bytes from xmm1 to memory location using the byte mask in xmm2. The default memory location is specified by DS:DI/EDI/RDI.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

1. ModRM.MOD = 011B required

**Description**

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQDU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQDU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQDU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVQDU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If VMASKMOVQDU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

---

1. ModRM.MOD = 011B required
Operation

IF (MASK[7] = 1)
  THEN DEST[DI/EDI] := SRC[7:0] ELSE (* Memory location unchanged *); FI;

IF (MASK[15] = 1)
  THEN DEST[DI/EDI+1] := SRC[15:8] ELSE (* Memory location unchanged *); FI;
  (* Repeat operation for 3rd through 14th bytes in source operand *)

IF (MASK[127] = 1)

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD If VEX.L = 1
  If VEX.vvvv ≠ 1111B.
MASKMOVQ—Store Selected Bytes of Quadword

**Opcode/Instruction** | **Op/En** | **64-Bit Mode** | **Compat/Leg Mode** | **Description**
--- | --- | --- | --- | ---
NP 0F F7 /r | RM | Valid | Valid | Selectively write bytes from mm1 to memory location using the byte mask in mm2. The default memory location is specified by DS:DI/EDI/RDI.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

**Operation**

(* Repeat operation for 3rd through 6th bytes in source operand *)
Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmove_si64(__m64d, __m64n, char * p)

Other Exceptions

See Table 23-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
### MAXPD—Maximum of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5F /r MAXPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the maximum double precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum double precision floating-point values between xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the maximum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Return the maximum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the maximum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed double precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

\[
\text{MAX}(\text{SRC1}, \text{SRC2}) \\
\begin{cases}
\text{IF } ((\text{SRC1} = 0.0) \text{ and } (\text{SRC2} = 0.0)) \text{ THEN } \text{DEST} := \text{SRC2}; \\
\text{ELSE IF } (\text{SRC1} = \text{NaN}) \text{ THEN } \text{DEST} := \text{SRC2}; \text{FI}; \\
\text{ELSE IF } (\text{SRC2} = \text{NaN}) \text{ THEN } \text{DEST} := \text{SRC2}; \text{FI}; \\
\text{ELSE IF } (\text{SRC1} > \text{SRC2}) \text{ THEN } \text{DEST} := \text{SRC1}; \\
\text{ELSE } \text{DEST} := \text{SRC2}; \\
\text{FI}; \\
\end{cases}
\]

VMAXPD (EVEX Encoded Versions)

\[(\text{KL}, \text{VL}) = (2, 128), (4, 256), (8, 512)\]

\[\text{FOR } j := 0 \text{ TO } \text{KL}-1 \]
\[i := j \times 64 \]
\[\text{IF } k1[j] \text{ OR } *\text{no writemask}*$
\[\text{THEN} \]
\[\text{IF } (\text{EVEX}.b = 1) \text{ AND } (\text{SRC2} \text{ *is memory*}) \]
\[\text{THEN} \]
\[\text{DEST}[i+63:i] := \text{MAX}(\text{SRC1}[i+63:i], \text{SRC2}[63:0]) \]
\[\text{ELSE} \]
\[\text{DEST}[i+63:i] := \text{MAX}(\text{SRC1}[i+63:i], \text{SRC2}[i+63:i]) \]
\[\text{FI}; \]
\[\text{ELSE} \]
\[\text{IF } *\text{merging-masking}* \]
\[\text{THEN } *\text{DEST}[i+63:i] \text{ remains unchanged*} \]
\[\text{ELSE } \text{DEST}[i+63:i] := 0 \quad \text{; zeroing-masking} \]
\[\text{FI} \]
\[\text{FI} \]
\[\text{ENDFOR} \]
\[\text{DEST}[\text{MAXVL-1:VL}] := 0 \]

VMAXPD (VEX.256 Encoded Version)

\[\text{DEST}[63:0] := \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0]) \]
\[\text{DEST}[127:64] := \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64]) \]
\[\text{DEST}[191:128] := \text{MAX}(\text{SRC1}[191:128], \text{SRC2}[191:128]) \]
\[\text{DEST}[255:192] := \text{MAX}(\text{SRC1}[255:192], \text{SRC2}[255:192]) \]
\[\text{DEST}[\text{MAXVL-1:256}] := 0 \]

VMAXPD (VEX.128 Encoded Version)

\[\text{DEST}[63:0] := \text{MAX}(\text{SRC1}[63:0], \text{SRC2}[63:0]) \]
\[\text{DEST}[127:64] := \text{MAX}(\text{SRC1}[127:64], \text{SRC2}[127:64]) \]
\[\text{DEST}[\text{MAXVL-1:128}] := 0 \]

MAXPD (128-bit Legacy SSE Version)

\[\text{DEST}[63:0] := \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0]) \]
\[\text{DEST}[127:64] := \text{MAX}(\text{DEST}[127:64], \text{SRC}[127:64]) \]
\[\text{DEST}[\text{MAXVL-1:128}] \text{ (Unmodified)} \]
Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD __m512d_mm512_max_pd(__m512d a, __m512d b);
VMAXPD __m512d_mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMAXPD __m512d_mm512_maskz_max_pd(__mmask8 k, __m512d a, __m512d b);
VMAXPD __m512d_mm512_max_round_pd(__m512d a, __m512d b, int);
VMAXPD __m512d_mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m256d_mm256_max_pd(__m256d a, __m256d b);
VMAXPD __m256d_mm256_mask_max_pd(__mmask8 k, __m256d a, __m256d b);
VMAXPD __m256d_mm256_maskz_max_pd(__mmask8 k, __m256d a, __m256d b);
VMAXPD __m128d_mm128_max_pd(__m128d a, __m128d b);
(V)MAXPD __m128d_mm_max_pd(__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
MAXPS—Maximum of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the maximum single precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum single precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 5F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum single precision floating-point values between ymm1 and ymm2, ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Return the maximum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Return the maximum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 5F /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the maximum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed single precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

\[
\text{MAX}(\text{SRC1}, \text{SRC2})
\]

\[
\begin{align*}
\text{IF } &((\text{SRC1} = 0.0) \text{ and } (\text{SRC2} = 0.0)) \text{ THEN } \text{DEST} := \text{SRC2}; \\
&\text{ELSE IF } (\text{SRC1} = \text{NaN}) \text{ THEN } \text{DEST} := \text{SRC2}; \text{FI}; \\
&\text{ELSE IF } (\text{SRC2} = \text{NaN}) \text{ THEN } \text{DEST} := \text{SRC2}; \text{FI}; \\
&\text{ELSE IF } (\text{SRC1} > \text{SRC2}) \text{ THEN } \text{DEST} := \text{SRC1}; \\
&\text{ELSE } \text{DEST} := \text{SRC2}; \\
&\text{FI};
\end{align*}
\]

VMAXPS (EVEX Encoded Versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]
FOR \(j := 0\) TO \(KL-1\)
\[
i := j \cdot 32
\]
IF \(k1[j]\) OR \(*\text{no writemask}\*\) THEN
\[
\text{IF } (\text{EVEX.b} = 1) \text{ AND } (\text{SRC2 }\text{is memory}) \text{ THEN }
\]
\[
\begin{align*}
\text{DEST}[i+31:i] &:= \text{MAX}(\text{SRC1}[i+31:i], \text{SRC2}[31:0]) \\
\text{ELSE } &\text{DEST}[i+31:i] := \text{MAX}(\text{SRC1}[i+31:i], \text{SRC2}[i+31:i]) \\
&\text{FI}; \\
\text{ELSE } &\text{DEST}[i+31:i] \text{ remains unchanged};
\end{align*}
\]
ELSE
\[
\text{DEST}[i+31:i] := 0 ; \text{zeroing-masking}
\]
FI
ENDFOR
\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VMAXPS (VEX.256 Encoded Version)

\[
\begin{align*}
\text{DEST}[31:0] &:= \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0]) \\
\text{DEST}[63:32] &:= \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32]) \\
\text{DEST}[95:64] &:= \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64]) \\
\text{DEST}[127:96] &:= \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96]) \\
\text{DEST}[159:128] &:= \text{MAX}(\text{SRC1}[159:128], \text{SRC2}[159:128]) \\
\text{DEST}[191:160] &:= \text{MAX}(\text{SRC1}[191:160], \text{SRC2}[191:160]) \\
\text{DEST}[223:192] &:= \text{MAX}(\text{SRC1}[223:192], \text{SRC2}[223:192]) \\
\text{DEST}[255:224] &:= \text{MAX}(\text{SRC1}[255:224], \text{SRC2}[255:224]) \\
\text{DEST}[\text{MAXVL}-1:256] &:= 0
\end{align*}
\]

VMAXPS (VEX.128 Encoded Version)

\[
\begin{align*}
\text{DEST}[31:0] &:= \text{MAX}(\text{SRC1}[31:0], \text{SRC2}[31:0]) \\
\text{DEST}[63:32] &:= \text{MAX}(\text{SRC1}[63:32], \text{SRC2}[63:32]) \\
\text{DEST}[95:64] &:= \text{MAX}(\text{SRC1}[95:64], \text{SRC2}[95:64]) \\
\text{DEST}[127:96] &:= \text{MAX}(\text{SRC1}[127:96], \text{SRC2}[127:96]) \\
\text{DEST}[\text{MAXVL}-1:128] &:= 0
\end{align*}
\]
MAXPS (128-bit Legacy SSE Version)
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[95:64] := MAX(DEST[95:64], SRC[95:64])
DEST[127:96] := MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMAXPS __m512 _mm512_max_ps( __m512 a, __m512 b);
VMAXPS __m512 _mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_maskz_max_ps( __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_max_round_ps( __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_maskz_max_round_ps( __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);
VMAXPS __m256 _mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 _mm256_maskz_max_ps( __mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 _mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 _mm_maskz_max_ps( __mmask8 k, __m128 a, __m128 b);
MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Invalid (including QNaN Source Operand), Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
## MAXSD—Return Maximum Scalar Double Precision Floating-Point Value

### Description

Compares the low double precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
    ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
    ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
    ELSE DEST := SRC2;
    FI;
}
```

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5F /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Return the maximum scalar double precision floating-point value between xmm2/m64 and xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 5F /r</td>
<td>B V/V</td>
<td>AVX</td>
<td>Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.F2.0F.W1 5F /r</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Return the maximum scalar double precision floating-point value between xmm3/m64 and xmm2.</td>
<td></td>
</tr>
</tbody>
</table>
VMAXSD (EVEX Encoded Version)

IF k1[0] or *no writemask*
    THEN DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
ELSE
    IF *merging-masking* ; merging-masking
       THEN *DEST[63:0] remains unchanged*
       ELSE ; zeroing-masking
           DEST[63:0] := 0
    FI;

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VMAXSD (VEX.128 Encoded Version)

DEST[63:0] := MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

MAXSD (128-bit Legacy SSE Version)

DEST[63:0] := MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXSD __m128d _mm_max_round_sd(__m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
MAXSS—Return Maximum Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5F /r MAXSS xmm1, xmm2/m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the maximum scalar single precision floating-point value between xmm2/m32 and xmm1.</td>
</tr>
<tr>
<td>VEX.LIGF3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
<tr>
<td>EVEX.LLG.F3.0F.W0 5F /r VMAXSS xmm1 (k1)[z], xmm2, xmm3/m32{sae}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the maximum scalar single precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Compares the low single precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```
MAX(SRC1, SRC2)
{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
  ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
  ELSE IF (SRC1 > SRC2) THEN DEST := SRC1;
  ELSE DEST := SRC2;
  FI;
```
VMAXSS (EVEX Encoded Version)
IF k1[0] or *no writemask*
    THEN DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[31:0] := 0
    FI;
FI;
DEST[MAXVL-1:128] := 0

VMAXSS (VEX.128 Encoded Version)
DEST[31:0] := MAX(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] := 0

MAXSS (128-bit Legacy SSE Version)
DEST[31:0] := MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMAXSS __m128 _mm_max_round_ss( __m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions
Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
**MFENCE—Memory Fence**

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE F0 MFENCE</td>
<td>ZO</td>
<td>V/V</td>
<td>SSE2</td>
<td>Serializes load and store operations.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction. The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

**Operation**

`Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);`

**Intel C/C++ Compiler Intrinsic Equivalent**

`void _mm_mfence(void)`

**Exceptions (All Modes of Operation)**

- **#UD** If CPUID.01H:EDX.SSE2[bit 26] = 0.
  
- If the LOCK prefix is used.

---

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.
## MINPD—Minimum of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>Mod Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5D /r MINPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the minimum double precision floating-point values between xmm1 and xmm2/mem</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum double precision floating-point values between xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum packed double precision floating-point values between ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 5D /r VMINPD xmm1 <a href="z">k1</a>, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Return the minimum packed double precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 5D /r VMINPD ymm1 <a href="z">k1</a>, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Return the minimum packed double precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 5D /r VMINPD zmm1 <a href="z">k1</a>, zmm2, zmm3/m512/m64bcst{sae}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum packed double precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvv (r)</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvv (r)</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed double precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

\[
\text{MIN}(\text{SRC1}, \text{SRC2})
\]

\[
\begin{align*}
\text{IF } (\text{SRC1} = 0.0) \text{ and } (\text{SRC2} = 0.0) & \text{ THEN } \text{DEST} := \text{SRC2}; \\
\text{ELSE IF } (\text{SRC1} = \text{NaN}) & \text{ THEN } \text{DEST} := \text{SRC2}; \\
\text{ELSE IF } (\text{SRC2} = \text{NaN}) & \text{ THEN } \text{DEST} := \text{SRC2}; \\
\text{ELSE IF } (\text{SRC1} < \text{SRC2}) & \text{ THEN } \text{DEST} := \text{SRC1}; \\
\text{ELSE DEST} & := \text{SRC2}; \\
\text{FI};
\end{align*}
\]

**VMINPD (EVEX Encoded Version)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j := 0 \text{ TO } KL-1\)

\[
i := j \times 64
\]

IF \(k1[j] \text{ OR *no writemask* \ THEN}
\]

\[
\text{IF } (\text{EVEX.b} = 1) \text{ AND } (\text{SRC2 *is memory*}) \\
\text{THEN}
\]

\[
\text{DEST}[i+63:i] := \text{MIN}(\text{SRC1}[i+63:i], \text{SRC2}[63:0])
\]

\[
\text{ELSE}
\]

\[
\text{DEST}[i+63:i] := \text{MIN}(\text{SRC1}[i+63:i], \text{SRC2}[i+63:i])
\]

\[
\text{FI};
\]

\[
\text{ELSE}
\]

\[
\text{IF *merging-masking* \ THEN *DEST}[i+63:i] \text{ remains unchanged*}
\]

\[
\text{ELSE} \text{ DEST}[i+63:i] := 0 \text{ \ ; zeroing-masking}
\]

\[
\text{FI}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]

**VMINPD (VEX.256 Encoded Version)**

\[
\text{DEST}[63:0] := \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] := \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])
\]

\[
\text{DEST}[191:128] := \text{MIN}(\text{SRC1}[191:128], \text{SRC2}[191:128])
\]

\[
\text{DEST}[255:192] := \text{MIN}(\text{SRC1}[255:192], \text{SRC2}[255:192])
\]

**VMINPD (VEX.128 Encoded Version)**

\[
\text{DEST}[63:0] := \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] := \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])
\]

\[
\text{DEST}[\text{MAXVL-1:128}] := 0
\]

**MINPD (128-bit Legacy SSE Version)**

\[
\text{DEST}[63:0] := \text{MIN}(\text{SRC1}[63:0], \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] := \text{MIN}(\text{SRC1}[127:64], \text{SRC2}[127:64])
\]

\[
\text{DEST}[\text{MAXVL-1:128}] \text{ (Unmodified)}
\]
Intel C/C++ Compiler Intrinsic Equivalent

VMINPD __m512d _mm512_min_pd( __m512d a, __m512d b);  
VMINPD __m512d _mm512_mask_min_pd( __m512d s, __mmask8 k, __m512d a, __m512d b);  
VMINPD __m512d _mm512_maskz_min_pd( __mmask8 k, __m512d a, __m512d b);  
VMINPD __m512d _mm512_min_round_pd( __m512d a, __m512d b, int);  
VMINPD __m512d _mm512_mask_min_round_pd( __m512d s, __mmask8 k, __m512d a, __m512d b, int);  
VMINPD __m256d _mm256_mask_min_pd( __m256d s, __mmask8 k, __m256d a, __m256d b);  
VMINPD __m256d _mm256_maskz_min_pd( __mmask8 k, __m256d a, __m256d b);  
VMINPD __m128d _mm_mask_min_pd( __m128d s, __mmask8 k, __m128d a, __m128d b);  
VMINPD __m128d _mm_maskz_min_pd( __mmask8 k, __m128d a, __m128d b);  
VMINPD __m256d _mm256_min_pd( __m256d a, __m256d b);  
MINPD __m128d _mm_min_pd( __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
MINPS—Minimum of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the minimum single precision floating-point values between xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.0F.W0 5D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum single precision floating-point values between xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.256.0F.W0 5D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum single precision floating-point values between ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Return the minimum packed single precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum packed single precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum packed single precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed single precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNAN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.
128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

MIN(SRC1, SRC2)

```plaintext
IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FI;
ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FI;
ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
ELSE DEST := SRC2;
FI;
```

**VMINPS (EVEX Encoded Version)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times 32\]

IF \(k1[j]\) OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

DEST\([i+31:i]\) := MIN(SRC1\([i+31:i]\), SRC2\([31:0]\))

ELSE

DEST\([i+31:i]\) := MIN(SRC1\([i+31:i]\), SRC2\([i+31:i]\))

FI;

ELSE

IF *merging-mask* ; merging-mask

THEN *DEST\([i+31:i]\) remains unchanged*

ELSE DEST\([i+31:i]\) := 0 ; zeroing-mask

FI

ENDIF

ENDFOR

DEST[MAXVL-1:VL] := 0

**VMINPS (VEX.256 Encoded Version)**

DEST\([31:0]\) := MIN(SRC1\([31:0]\), SRC2\([31:0]\))

DEST\([63:32]\) := MIN(SRC1\([63:32]\), SRC2\([63:32]\))

DEST\([95:64]\) := MIN(SRC1\([95:64]\), SRC2\([95:64]\))

DEST\([127:96]\) := MIN(SRC1\([127:96]\), SRC2\([127:96]\))

DEST\([159:128]\) := MIN(SRC1\([159:128]\), SRC2\([159:128]\))

DEST\([191:160]\) := MIN(SRC1\([191:160]\), SRC2\([191:160]\))

DEST\([223:192]\) := MIN(SRC1\([223:192]\), SRC2\([223:192]\))

DEST\([255:224]\) := MIN(SRC1\([255:224]\), SRC2\([255:224]\))

**VMINPS (VEX.128 Encoded Version)**

DEST\([31:0]\) := MIN(SRC1\([31:0]\), SRC2\([31:0]\))

DEST\([63:32]\) := MIN(SRC1\([63:32]\), SRC2\([63:32]\))

DEST\([95:64]\) := MIN(SRC1\([95:64]\), SRC2\([95:64]\))

DEST\([127:96]\) := MIN(SRC1\([127:96]\), SRC2\([127:96]\))

DEST[MAXVL-1:128] := 0
MINPS (128-bit Legacy SSE Version)
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] := MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] := MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] := MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMINPS __m512 __mm512_min_ps(__m512 a, __m512 b);
VMINPS __m512 __mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 __mm512_maskz_min_ps(__mmask16 k, __m512 a, __m512 b);
VMINPS __m512 __mm512_min_round_ps(__m512 a, __m512 b, int);
VMINPS __m512 __mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 __mm512_maskz_min_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 __mm256_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 __mm256_mask_min_ps(__mmask8 k, __m256 a, __m256 b);
VMINPS __m256 __mm256_maskz_min_ps(__mmask8 k, __m256 a, __m256 b);
VMINPS __m128 __mm128_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 __mm128_mask_min_ps(__mmask8 k, __m128 a, __m128 b);
VMINPS __m128 __mm128_maskz_min_ps(__mmask8 k, __m128 a, __m128 b);
MINPS __m128 __mm128_min_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions
Invalid (including QNaN Source Operand), Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions."
MINSD—Return Minimum Scalar Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the minimum scalar double precision floating-point value between xmm2/m64 and xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 5D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.</td>
</tr>
<tr>
<td>EVEX.LLIG.F2.0F.W1 5D /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum scalar double precision floating-point value between xmm3/m64 and xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Compares the low double precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSD is encoded with VEX.L=0. Encoding VMINSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) AND (SRC2 = 0.0)) THEN DEST := SRC2;
    ELSE IF (SRC1 = NaN) THEN DEST := SRC2; FL;
    ELSE IF (SRC2 = NaN) THEN DEST := SRC2; FL;
    ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
    ELSE DEST := SRC2;
    FL;
}
```
INSTRUCTION SET REFERENCE, M-U

MINSD (EVEX Encoded Version)
IF k1[0] or *no writemask*
    THEN DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged* 
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
    FI;
    fi;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

MINSD (VEX.128 Encoded Version)
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

MINSD (128-bit Legacy SSE Version)
DEST[63:0] := MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions
Invalid (including QNaN Source Operand), Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
MINSS—Return Minimum Scalar Single Precision Floating-Point Value

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the minimum scalar single precision floating-point value between xmm2/m32 and xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.W1G 5D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 5D /r  VMINSS xmm1 (k1)[r], xmm2, xmm3/m32{sae}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Return the minimum scalar single precision floating-point value between xmm3/m32 and xmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Compares the low single precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

```c
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST := SRC2;
    ELSE IF (SRC1 = NaN) THEN DEST := SRC2; Fl;
    ELSE IF (SRC2 = NaN) THEN DEST := SRC2; Fl;
    ELSE IF (SRC1 < SRC2) THEN DEST := SRC1;
    ELSE DEST := SRC2;
    Fl;
```
MINSS (EVEX Encoded Version)
IF k1[0] or *no writemask*
    THEN DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*;
    ELSE ; zeroing-masking
        THEN DEST[31:0] := 0
    FI;
FI;
DEST[MAXVL-1:128] := 0

VMINSS (VEX.128 Encoded Version)
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] := 0

MINSS (128-bit Legacy SSE Version)
DEST[31:0] := MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions
Invalid (Including QNaN Source Operand), Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
MONITOR—Set Up Monitor Address

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 C8</td>
<td>MONITOR</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:RAX/EAX/AX.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The address is specified in RAX/EAX/AX and the size is based on the effective address size of the encoded instruction. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 9, “Multiple-Processor Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

MONITOR void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

Numeric Exceptions

None.
Protected Mode Exceptions

#GP(0)  If the value in EAX is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
   If ECX ≠ 0.
#SS(0)  If the value in EAX is outside the SS segment limit.
#PF(fault-code)  For a page fault.
#UD  If CPUID.01H:ECX.MONITOR[bit 3] = 0.
   If current privilege level is not 0.

Real Address Mode Exceptions

#GP  If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
   If ECX ≠ 0.
#SS  If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD  If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD  The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.
   If RCX ≠ 0.
#SS(0)  If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)  For a page fault.
#UD  If the current privilege level is not 0.
   If CPUID.01H:ECX.MONITOR[bit 3] = 0.
# MOV—Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/A Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>88 /r</td>
<td>MOV r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>REX + 88 /r</td>
<td>MOV r/m81, r81</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>89 /r</td>
<td>MOV r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r16 to r/m16.</td>
</tr>
<tr>
<td>89 /r</td>
<td>MOV r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 89 /r</td>
<td>MOV r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to r/m64.</td>
</tr>
<tr>
<td>8A /r</td>
<td>MOV r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>REX + 8A /r</td>
<td>MOV r81, r/m81</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r/m8 to r8.</td>
</tr>
<tr>
<td>8B /r</td>
<td>MOV r/m16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to r/m16.</td>
</tr>
<tr>
<td>8B /r</td>
<td>MOV r/m32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 8B /r</td>
<td>MOV r/m64, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to r/m64.</td>
</tr>
<tr>
<td>8C /r</td>
<td>MOV r/m16, Sreg2</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move segment register to r/m16.</td>
</tr>
<tr>
<td>8C /r</td>
<td>MOV r16/r32/m16, Sreg2</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move zero extended 16-bit segment register to r16/r32/m16.</td>
</tr>
<tr>
<td>REX.W + 8C /r</td>
<td>MOV r/m64, Sreg2</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move zero extended 16-bit segment register to r/m64/m16.</td>
</tr>
<tr>
<td>8E /r</td>
<td>MOV Sreg, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r/m16 to segment register.</td>
</tr>
<tr>
<td>REX.W + 8E /r</td>
<td>MOV Sreg, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move lower 16 bits of r/m64 to segment register.</td>
</tr>
<tr>
<td>A0</td>
<td>MOV AL, moffs83</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte at (seg:offset) to AL.</td>
</tr>
<tr>
<td>REX.W + A0</td>
<td>MOV AL, moffs83</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte at (offset) to AL.</td>
</tr>
<tr>
<td>A1</td>
<td>MOV AX, moffs163</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move word at (seg:offset) to AX.</td>
</tr>
<tr>
<td>A1</td>
<td>MOV EAX, moffs323</td>
<td>FD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move doubleword at (seg:offset) to EAX.</td>
</tr>
<tr>
<td>REX.W + A1</td>
<td>MOV RAX, moffs643</td>
<td>FD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move quadword at (offset) to RAX.</td>
</tr>
<tr>
<td>A2</td>
<td>MOV moffs8, AL</td>
<td>TD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move AL to (seg:offset).</td>
</tr>
<tr>
<td>REX.W + A2</td>
<td>MOV moffs81, AL</td>
<td>TD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move AL to (offset).</td>
</tr>
<tr>
<td>A3</td>
<td>MOV moffs163, AX</td>
<td>TD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move AX to (seg:offset).</td>
</tr>
<tr>
<td>A3</td>
<td>MOV moffs323, EAX</td>
<td>TD</td>
<td>Valid</td>
<td>Valid</td>
<td>Move EAX to (seg:offset).</td>
</tr>
<tr>
<td>REX.W + A3</td>
<td>MOV moffs643, RAX</td>
<td>TD</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move RAX to (offset).</td>
</tr>
<tr>
<td>B0 + rb ib</td>
<td>MOV r8, imm8</td>
<td>OI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm8 to r8.</td>
</tr>
<tr>
<td>REX + B0 + rb ib</td>
<td>MOV r81, imm8</td>
<td>OI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm8 to r8.</td>
</tr>
<tr>
<td>B8 + rw iw</td>
<td>MOV r16, imm16</td>
<td>OI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm16 to r16.</td>
</tr>
<tr>
<td>B8 + rd id</td>
<td>MOV r32, imm32</td>
<td>OI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm32 to r32.</td>
</tr>
<tr>
<td>REX.W + B8 + rd io</td>
<td>MOV r/m64, imm64</td>
<td>OI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm64 to r64.</td>
</tr>
<tr>
<td>C6 /0 ib</td>
<td>MOV r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm8 to r/m8.</td>
</tr>
<tr>
<td>REX + C6 /0 ib</td>
<td>MOV r/m81, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm8 to r/m8.</td>
</tr>
<tr>
<td>C7 /0 iw</td>
<td>MOV r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm16 to r/m16.</td>
</tr>
<tr>
<td>C7 /0 id</td>
<td>MOV r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Move imm32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + C7 /0 id</td>
<td>MOV r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move imm32 sign extended to 64-bits to r/m64.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
MOV—Move

2. In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following “Description” section for further information).

3. The moffs8, moffs16, moffs32, and moffs64 operands specify a simple offset relative to the segment base, where 8, 16, 32, and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32, or 64 bits.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRMreg/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>FD</td>
<td>AL/AX/EAX/RAX</td>
<td>MoffS</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TD</td>
<td>MoffS (w)</td>
<td>AL/AX/EAX/RAX</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OI</td>
<td>opcode + rd (w)</td>
<td>imm8/16/32/64</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMreg/m (w)</td>
<td>imm8/16/32/64</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the “Operation” algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, stack-pointer value) before an event can be delivered. See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium, and earlier processors.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST := SRC;

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

IF SS is loaded
THEN
  IF segment selector is NULL
    THEN #GP(0); FI;
  IF segment selector index is outside descriptor table limits
    OR segment selector’s RPL ≠ CPL
    OR segment is not a writable data segment
    OR DPL ≠ CPL
    THEN #GP(selector); FI;
  IF segment not marked present
    THEN #SS(selector);
    ELSE
      SS := segment selector;
      SS := segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector
THEN
  IF segment selector index is outside descriptor table limits
    OR segment is not a data or readable code segment
    OR ((segment is a data or nonconforming code segment) AND ((RPL > DPL) or (CPL > DPL)))
    THEN #GP(selector); FI;
  IF segment not marked present
    THEN #NP(selector);
    ELSE
      SegmentRegister := segment selector;
      SegmentRegister := segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with NULL selector
THEN
  SegmentRegister := segment selector;
  SegmentRegister := segment descriptor;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If attempt is made to load SS register with NULL segment selector.
If the destination operand is in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#GP(selector) If segment selector index is outside descriptor table limits.
If the SS register is being loaded and the segment selector’s RPL and the segment descriptor’s DPL are not equal to the CPL.
If the SS register is being loaded and the segment pointed to is a non-writable data segment.
If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, and either the RPL or the CPL is greater than the DPL.

#SS(0) If a memory operand effective address is outside the SS segment limit.
#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.
#NP If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If attempt is made to load the CS register.
If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If attempt is made to load the CS register.
If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If attempt is made to load the CS register.
If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0) If the memory address is in a non-canonical form.
If an attempt is made to load SS register with NULL segment selector when CPL = 3.
If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.

#GP(selector) If segment selector index is outside descriptor table limits.
If the memory access to the descriptor table is non-canonical.
If the SS register is being loaded and the segment selector’s RPL and the segment descriptor’s DPL are not equal to the CPL.
If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.

#SS(0) If the stack address is in a non-canonical form.

#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If attempt is made to load the CS register.
If the LOCK prefix is used.
MOV—Move to/from Control Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 20/r MOV r32, CR0-CR7</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move control register to r32.</td>
</tr>
<tr>
<td>0F 20/r MOV r64, CR0-CR7</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move extended control register to r64.</td>
</tr>
<tr>
<td>REX.R + 0F 20 /r MOV r64, CR8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move extended CR8 to r64.¹</td>
</tr>
<tr>
<td>0F 22 /r MOV CR0-CR7, r32</td>
<td>RM</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move r32 to control register.</td>
</tr>
<tr>
<td>0F 22 /r MOV CR0-CR7, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to extended control register.</td>
</tr>
<tr>
<td>REX.R + 0F 22 /r MOV CR8, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to extended CR8.¹</td>
</tr>
</tbody>
</table>

**NOTES:**
1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 9 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM/r/m (w)</td>
<td>ModRM/reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM/reg (w)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general-purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. On a 64-bit capable processor, an execution of MOV to CR outside of 64-bit mode zeros the upper 32 bits of the control register. (See "Control Registers" in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Some of the bits in CR0, CR3, and CR4 are reserved and must be written with zeros. Attempting to set any reserved bits in CR0[31:0] is ignored. Attempting to set any reserved bits in CR0[63:32] results in a general-protection exception, #GP(0). When PCIDs are not enabled, bits 2:0 and bits 11:5 of CR3 are not used and attempts to set them are ignored. Attempting to set any reserved bits in CR3[63:MAXPHYADDR] results in #GP(0). Attempting to set any reserved bits in CR4 results in #GP(0). On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction’s default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. Use
of the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

\[
\text{DEST} := \text{SRC};
\]

**Flags Affected**

The OF, SF, ZF, AF, PF, and CF flags are undefined.

**Protected Mode Exceptions**

- **#GP(0)** If the current privilege level is not 0.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
- If an attempt is made to write a 1 to any reserved bit in CR4.
- If an attempt is made to write 1 to CR4.PCIDE.
- If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
- If an attempt is made to activate IA-32e mode and either the current CS has the L-bit set or the TR references a 16-bit TSS.
- **#UD** If the LOCK prefix is used.
- If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

**Real-Address Mode Exceptions**

- **#GP** If an attempt is made to write a 1 to any reserved bit in CR4.
- If an attempt is made to write 1 to CR4.PCIDE.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
- If an attempt is made to activate IA-32e mode and either the current CS has the L-bit set or the TR references a 16-bit TSS.
- **#UD** If the LOCK prefix is used.
- If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** These instructions cannot be executed in virtual-8086 mode.

**Compatibility Mode Exceptions**

- **#GP(0)** If the current privilege level is not 0.
- If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
- If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
- If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
- If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD  If the LOCK prefix is used.
If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.

64-Bit Mode Exceptions

#GP(0)  If the current privilege level is not 0.
If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
If an attempt is made to clear CR0.PG[bit 31].
If an attempt is made to write a 1 to any reserved bit in CR4.
If an attempt is made to write a 1 to any reserved bit in CR8.
If an attempt is made to write a 1 to any reserved bit in CR3[63:MAXPHYADDR].
If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].

#UD  If the LOCK prefix is used.
If an attempt is made to access CR1, CR5, CR6, CR7, or CR9–CR15.
If the REX.R prefix is used to specify a register other than CR8.
MOV—Move to/from Debug Registers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 21/r MOV r32, DR0–DR7</td>
<td>MR</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move debug register to r32.</td>
</tr>
<tr>
<td>0F 21/r MOV r64, DR0–DR7</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move extended debug register to r64.</td>
</tr>
<tr>
<td>0F 23 /r MOV DR0–DR7, r32</td>
<td>RM</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move r32 to debug register.</td>
</tr>
<tr>
<td>0F 23 /r MOV DR0–DR7, r64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r64 to extended debug register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 18.2, “Debug Registers”, of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the reg field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the mod field are ignored. The r/m field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8–R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST := SRC;
    FI;

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.
**Protected Mode Exceptions**

- **#GP(0)**  
  If the current privilege level is not 0.
- **#UD**  
  If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
  - If the LOCK prefix is used.
- **#DB**  
  If any debug register is accessed while the DR7.GD[bit 13] = 1.

**Real-Address Mode Exceptions**

- **#UD**  
  If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
  - If the LOCK prefix is used.
- **#DB**  
  If any debug register is accessed while the DR7.GD[bit 13] = 1.

**Virtual-8086 Mode Exceptions**

- **#GP(0)**  
  The debug registers cannot be loaded or read when in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#GP(0)**  
  If the current privilege level is not 0.
  - If an attempt is made to write a 1 to any of bits 63:32 in DR6.
  - If an attempt is made to write a 1 to any of bits 63:32 in DR7.
- **#UD**  
  If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
  - If the LOCK prefix is used.
  - If the REX.R prefix is used.
- **#DB**  
  If any debug register is accessed while the DR7.GD[bit 13] = 1.
### MOVAPD—Move Aligned Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg/r/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRMreg/r/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Moves 2, 4 or 8 double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection fault will occur.
exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVAE.512 encoded version:
Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:
Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:
Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX Encoded Versions, Register-Copy Form)

\[ (KL, VL) = (2, 128), (4, 256), (8, 512) \]

FOR \( j = 0 \) TO \( KL - 1 \)

\[ i := j * 64 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+63:i] := \text{SRC}[i+63:i] \)

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+63:i] \) remains unchanged*

ELSE \( \text{DEST}[i+63:i] := 0 \) ; zeroing-masking

FI

FI;
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)
VMOVPD (EVEX Encoded Versions, Store-Form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[i+63:i]
    ELSE
      ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;

VMOVPD (EVEX Encoded Versions, Load-Form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] := 0 ; zeroing-masking
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVPD (VEX.256 Encoded Version, Load - and Register Copy)
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

VMOVPD (VEX.256 Encoded Version, Store-Form)
DEST[255:0] := SRC[255:0]

VMOVPD (VEX.128 Encoded Version, Load - and Register Copy)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

MOVAPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVAPD (128-bit Store-Form Version)
DEST[127:0] := SRC[127:0]
Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m512d__m512_load_pd( void * m);
VMOVAPD __m512d__m512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d__m512_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void __m512_store_pd( void * d, __m512d a);
VMOVAPD void __m512_mask_store_pd( void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d__m256_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d__m256_mask_load_pd(__mmask8 k, void * m);
VMOVAPD void __m256_mask_store_pd( void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d__m128_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d__m128_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void __m128_mask_store_pd( void * d, __mmask8 k, __m128d a);
MOVAPD __m256d__m256_load_pd (double * p);
MOVAPD void __m256_store_pd (double * p, __m256d a);
MOVAPD __m128d__m128_load_pd (double * p);
MOVAPD void __m128_store_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-44, “Type E1 Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
MOVAPS—Move Aligned Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 28 /r MOVAPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move aligned packed single precision floating-point values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>NP 0F 29 /r MOVAPS xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE</td>
<td>Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single precision floating-point values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single precision floating-point values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single precision floating-point values from ymm2/mem to ymm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed single precision floating-point values from ymm1 to ymm2/mem.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Move aligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Move aligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Move aligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Move aligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Moves 4, 8 or 16 single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
EVEX.512 encoded version:
Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:
Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:
Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

**Operation**

**VMOVAPS (EVEX Encoded Versions, Register-Copy Form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[i+31:i]
    ELSE IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] := 0 ; zeroing-masking
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VMOVAPS (EVEX Encoded Versions, Store Form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
    ELSE DEST[i+31:i] := 0 ; zeroing-masking
  FI;
ENDFOR;
**INSTRUCTION SET REFERENCE, M-U**

**VMOVAPS (EVEX Encoded Versions, Load Form)**

\( (KL, VL) = (4, 128), (8, 256), (16, 512) \)

FOR \( j := 0 \) TO \( KL-1 \)

\[ i := j \times 32 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+31:i] := \text{SRC}[i+31:i] \)

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+31:i] \) remains unchanged*

ELSE \( \text{DEST}[i+31:i] := 0 \) ; zeroing-masking

Fi

Fi;
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:\text{VL}] := 0 \)

**VMOVAPS (VEX.256 Encoded Version, Load - and Register Copy)**

\( \text{DEST}[255:0] := \text{SRC}[255:0] \)

\( \text{DEST}[\text{MAXVL}-1:256] := 0 \)

**VMOVAPS (VEX.256 Encoded Version, Store-Form)**

\( \text{DEST}[255:0] := \text{SRC}[255:0] \)

**VMOVAPS (VEX.128 Encoded Version, Load - and Register Copy)**

\( \text{DEST}[127:0] := \text{SRC}[127:0] \)

\( \text{DEST}[\text{MAXVL}-1:128] := 0 \)

**MOVAPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)**

\( \text{DEST}[127:0] := \text{SRC}[127:0] \)

\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified)

**(V)MOVAPS (128-bit Store-Form Version)**

\( \text{DEST}[127:0] := \text{SRC}[127:0] \)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVAPS __m512 _mm512_load_ps( void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps( __mmask16 k, void * m);
VMOVAPS void _mm512_store_ps( void * d, __m512 a);
VMOVAPS __m128 _mm128_load_ps( float * p);
VMOVAPS __m128 _mm128_mask_load_ps( __m128 s, __mmask8 k, void * p);
VMOVAPS void _mm128_store_ps( float * p, __m128 a);

**SIMD Floating-Point Exceptions**

None.
Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, "Type 1 Class Exception Conditions," additionally:

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-44, "Type E1 Class Exception Conditions."
**MOVBE—Move Data After Swapping Bytes**

**Op/En Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 38 F0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>MOVBE</td>
<td>Reverse byte order in m16 and move to r16.</td>
</tr>
<tr>
<td>MOVBE r16, m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F 38 F0 /r</td>
<td>RM</td>
<td>V/V</td>
<td>MOVBE</td>
<td>Reverse byte order in m32 and move to r32.</td>
</tr>
<tr>
<td>MOVBE r32, m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REX.W + 0F 38 F0 /r</td>
<td>RM</td>
<td>V/N.E.</td>
<td>MOVBE</td>
<td>Reverse byte order in m64 and move to r64.</td>
</tr>
<tr>
<td>MOVBE r64, m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F 38 F1 /r</td>
<td>MR</td>
<td>V/V</td>
<td>MOVBE</td>
<td>Reverse byte order in r16 and move to m16.</td>
</tr>
<tr>
<td>MOVBE m16, r16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F 38 F1 /r</td>
<td>MR</td>
<td>V/V</td>
<td>MOVBE</td>
<td>Reverse byte order in r32 and move to m32.</td>
</tr>
<tr>
<td>MOVBE m32, r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REX.W + 0F 38 F1 /r</td>
<td>MR</td>
<td>V/N.E.</td>
<td>MOVBE</td>
<td>Reverse byte order in r64 and move to m64.</td>
</tr>
<tr>
<td>MOVBE m64, r64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```plaintext
TEMP := SRC
IF (OperandSize = 16)
    THEN
        DEST[7:0] := TEMP[15:8];
        DEST[15:8] := TEMP[7:0];
ELSE IF (OperandSize = 32)
        DEST[7:0] := TEMP[31:24];
        DEST[23:16] := TEMP[15:8];
        DEST[31:23] := TEMP[7:0];
ELSE IF (OperandSize = 64)
        DEST[7:0] := TEMP[63:56];
```
DEST[23:16] := TEMP[47:40];
DEST[47:40] := TEMP[23:16];
DEST[63:56] := TEMP[7:0];

FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the destination operand is in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.MOVBE[bit 22] = 0.
If the LOCK prefix is used.
If REP (F3H) prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If CPUID.01H:ECX.MOVBE[bit 22] = 0.
If the LOCK prefix is used.
If REP (F3H) prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.MOVBE[bit 22] = 0.
If the LOCK prefix is used.
If REP (F3H) prefix is used.
If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0)    If the memory address is in a non-canonical form.
#SS(0)    If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD      If CPUID.01H:ECX.MOVBE[bit 22] = 0.
          If the LOCK prefix is used.
          If REP (F3H) prefix is used.
## MOVD/MOVQ—Move Doubleword/Move Quadword

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 6E /r</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Move doubleword from r/m32 to mm.</td>
</tr>
<tr>
<td>MOVQ mm, r/m64</td>
<td>NP REX.W + 0F 6E /r</td>
<td>A</td>
<td>V/N.E.</td>
<td>MMX</td>
</tr>
<tr>
<td>NP 0F 7E /r</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Move doubleword from mm to r/m32.</td>
</tr>
<tr>
<td>MOVQ r/m64, mm</td>
<td>NP REX.W + 0F 7E /r</td>
<td>B</td>
<td>V/N.E.</td>
<td>MMX</td>
</tr>
<tr>
<td>66 0F 6E /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move doubleword from r/m32 to xmm.</td>
</tr>
<tr>
<td>MOVQ xmm, r/m32</td>
<td>66 REX.W 0F 6E /r</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE2</td>
</tr>
<tr>
<td>66 0F 7E /r</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move doubleword from xmm register to r/m32.</td>
</tr>
<tr>
<td>MOVQ r/m64, xmm</td>
<td>66 REX.W 0F 7E /r</td>
<td>B</td>
<td>V/N.E.</td>
<td>SSE2</td>
</tr>
<tr>
<td>VEX.128.66.6F.W0 6E /</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move doubleword from r/m32 to xmm1.</td>
</tr>
<tr>
<td>VMOVQ xmm1, r32/m32</td>
<td>VEX.128.66.6F.W1 6E /r</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX</td>
</tr>
<tr>
<td>VEX.128.66.6F.W0 7E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move doubleword from xmm1 register to r/m32.</td>
</tr>
<tr>
<td>VMOVQ r32/m32, xmm1</td>
<td>VEX.128.66.6F.W1 7E /r</td>
<td>B</td>
<td>V/N.E.¹</td>
<td>AVX</td>
</tr>
<tr>
<td>VEX.128.66.6F.W0 6E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move doubleword from r/m32 to xmm1.</td>
</tr>
<tr>
<td>VMOVQ xmm1, r32/m32</td>
<td>EVEX.128.66.6F.W1 6E /r</td>
<td>C</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.128.66.6F.W0 7E /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move doubleword from xmm1 register to r/m32.</td>
</tr>
<tr>
<td>VMOVQ r32/m32, xmm1</td>
<td>EVEX.128.66.6F.W1 7E /r</td>
<td>D</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

### NOTES:
1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

MOVD/MOVQ—Move Doubleword/Move Quadword
Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOV or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (When Destination Operand is an MMX Technology Register)

\[
\text{DEST}[31:0] := \text{SRC}; \quad \text{DEST}[63:32] := 00000000\text{H};
\]

MOVD (When Destination Operand is an XMM Register)

\[
\text{DEST}[31:0] := \text{SRC}; \quad \text{DEST}[127:32] := 000000000000000000000000\text{H}; \quad \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}
\]

MOVD (When Source Operand is an MMX Technology or XMM Register)

\[
\text{DEST} := \text{SRC}[31:0];
\]

VMOVQ (When Destination is an XMM Register)

\[
\text{DEST}[31:0] := \text{SRC}[31:0] \quad \text{DEST}[\text{MAXVL}-1:32] := 0
\]

MOVD (When Source Operand is an MMX Technology or XMM Register)

\[
\text{DEST}[63:0] := \text{SRC}[63:0]; \quad \text{DEST}[127:64] := 0000000000000000\text{H}; \quad \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}
\]
MOVQ (When Destination Operand is r/m64)
\[ \text{DEST}[63:0] := \text{SRC}[63:0]; \]

MOVQ (When Source Operand is an XMM Register or r/m64)
\[ \text{DEST} := \text{SRC}[63:0]; \]

VMOVQ (VEX-Encoded Version When Destination is an XMM Register)
\[ \text{DEST}[63:0] := \text{SRC}[63:0]; \]
\[ \text{DEST}[\text{MAXVL}-1:64] := 0 \]

VMOVD (EVEX-Encoded Version When Destination is an XMM Register)
\[ \text{DEST}[31:0] := \text{SRC}[31:0]; \]
\[ \text{DEST}[\text{MAXVL}-1:32] := 0 \]

VMOVQ (EVEX-Encoded Version When Destination is an XMM Register)
\[ \text{DEST}[63:0] := \text{SRC}[63:0]; \]
\[ \text{DEST}[\text{MAXVL}-1:64] := 0 \]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVD __m64 _mm_cvtsi32_si64 (int i )
MOVD int _mm_cvtsi64_si32 ( __m64m )
MOVD __m128i _mm_cvtsi32_si128 (int a)
MOVD int _mm_cvtsi128_si32 ( __m128i a)
MOVD __int64 _mm_cvtsi128_si64( __m128i );
MOVD _m128i _mm_cvtsi128_si128( __int64 );
VMOVD __m128i _mm_cvtsi32_si128( int);
VMOVD int _mm_cvtsi128_si32( __m128i );
VMOVD __m128i _mm_cvtsi64_si128( __int64 );
VMOVQ __int64 _mm_cvtsi128_si64( __m128i );
VMOVQ __m128i _mm_loadl_epi64( __m128i * s);
VMOVQ void _mm_storel_epi64( __m128i * d, __m128i s);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
Additionally:

#UD If VEX.L = 1.
If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
MOVDDUP—Replicate Double Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 12 /r MOVDDUP xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE3</td>
<td>Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.</td>
</tr>
<tr>
<td>VEX.128.F2.0F:W1 12 /r VMOVDDUP xmm1, xmm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move double precision floating-point value from xmm2/m64 and duplicate into xmm1.</td>
</tr>
<tr>
<td>VEX.256.F2.0F:W1 12 /r VMOVDDUP ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move even index double precision floating-point values from ymm2/mem and duplicate each element into ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F2.0F:W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move double precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move even index double precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move even index double precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>MOVDDUP</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

For 256-bit or higher versions: Duplicates even-indexed double precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

VMOVDDUP (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_SRC[63:0] := SRC[63:0]
TMP_SRC[127:64] := SRC[63:0]
IF VL >= 256
FI;
IF VL >= 512
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_SRC[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE
      ; zeroing-masking
      DEST[i+63:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVDDUP (VEX.256 Encoded Version)
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:256] := 0

VMOVDDUP (VEX.128 Encoded Version)
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] := 0

Figure 4-2. VMOVDDUP Operation
MOVDDUP (128-bit Legacy SSE Version)
DEST[63:0] := SRC[63:0]
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMOVDDUP __m512d _mm512_movedup_pd(__m512d a);
VMOVDDUP __m512d __m512_mask_movedup_pd(__m512d s, __mmask8 k, __m512d a);
VMOVDDUP __m512d __m512_maskz_movedup_pd(__mmask8 k, __m512d a);
VMOVDDUP __m256d __m256_mask_movedup_pd(__m256d s, __mmask8 k, __m256d a);
VMOVDDUP __m256d __m256_maskz_movedup_pd(__mmask8 k, __m256d a);
VMOVDDUP __m128d __m128_mask_movedup_pd(__m128d s, __mmask8 k, __m128d a);
VMOVDDUP __m128d __m128_maskz_movedup_pd(__mmask8 k, __m128d a);
MOVDDUP __m256d __m256_movedup_pd (__m256d a);
MOVDDUP __m128d __m128_movedup_pd (__m128d a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-52, “Type E5NF Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
MOVDIRI—Move Doubleword as Direct Store

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 F9 /r MOVDIRI m32, r32</td>
<td>A</td>
<td>V/V</td>
<td>MOVDIRI</td>
<td>Move doubleword from r32 to m32 using direct store.</td>
</tr>
<tr>
<td>NP REX.W + 0F 38 F9 /r MOVDIRI m64, r64</td>
<td>A</td>
<td>V/N.E.</td>
<td>MOVDIRI</td>
<td>Move quadword from r64 to m64 using direct store.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

1. The Mod field of the ModR/M byte cannot have value 11B.

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a direct-store operation. The source operand is a general purpose register. The destination operand is a 32-bit memory location. In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See summary chart at the beginning of this section for encoding data and limits.

The direct-store is implemented by using write combining (WC) memory type protocol for writing data. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store. Unlike stores with non-temporal hint that allow uncached (UC) and write-protected (WP) memory-type for the destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of the destination address memory type (including UC and WP types).

Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Direct stores are weakly ordered relative to other stores. Software that desires stronger ordering should use a fencing instruction (MFENCE or SFENCE) before or after a direct store to enforce the ordering desired.

Direct-stores issued by MOVDIRI to a destination aligned to a 4-byte boundary (8-byte boundary if used with REX.W prefix) guarantee 4-byte (8-byte with REX.W prefix) write-completion atomicity. This means that the data arrives at the destination in a single undivided 4-byte (or 8-byte) write transaction. If the destination is not aligned for the write size, the direct-stores issued by MOVDIRI are split and arrive at the destination in two parts. Each part of such split direct-store will not merge with younger stores but can arrive at the destination in either order. Availability of the MOVDIRI instruction is indicated by the presence of the CPUID feature flag MOVDIRI (bit 27 of the ECX register in leaf 07H, see “CPUID—CPU Identification” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

Operation

DEST := SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIRI void _directstore_u32(void *dst, uint32_t val)
MOVDIRI void _directstore_u64(void *dst, uint64_t val)

1. The Mod field of the ModR/M byte cannot have value 11B.
Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#UD If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
#AC If alignment checking is enabled and an unaligned memory reference made while in current
privilege level 3.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
#AC If alignment checking is enabled and an unaligned memory reference made while in current
privilege level 3.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF (fault-code) For a page fault.
#AC If alignment checking is enabled and an unaligned memory reference made while in current
privilege level 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If memory address referencing the SS segment is in non-canonical form.
#GP(0) If the memory address is in non-canonical form.
#PF (fault-code) For a page fault.
#UD If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
#AC If alignment checking is enabled and an unaligned memory reference made while in current
privilege level 3.
MOVDIR64B—Move 64 Bytes as Direct Store

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 F8 /r</td>
<td>A</td>
<td>V/V</td>
<td>MOVDIR64B</td>
<td>Move 64-bytes as direct-store with guaranteed 64-byte write atomicity from the source memory operand address to destination memory address specified as offset to ES segment in the register operand.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Moves 64-bytes as direct-store with 64-byte write atomicity from source memory address to destination memory address. The source operand is a normal memory operand. The destination operand is a memory location specified in a general-purpose register. The register content is interpreted as an offset into ES segment without any segment override. In 64-bit mode, the register operand width is 64-bits (32-bits with 67H prefix). Outside of 64-bit mode, the register width is 32-bits when CS.D=1 (16-bits with 67H prefix), and 16-bits when CS.D=0 (32-bits with 67H prefix). MOVDIR64B requires the destination address to be 64-byte aligned. No alignment restriction is enforced for source operand.

MOVDIR64B first reads 64-bytes from the source memory address. It then performs a 64-byte direct-store operation to the destination address. The load operation follows normal read ordering based on source address memory-type. The direct-store is implemented by using the write combining (WC) memory type protocol for writing data. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store.

Unlike stores with non-temporal hint which allow UC/WP memory-type for destination to override the non-temporal hint, direct-stores always follow WC memory-type protocol irrespective of destination address memory type (including UC/WP types). Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combining buffer may be combined with younger direct stores to the same address. Direct stores are weakly ordered relative to other stores. Software that desires stronger ordering should use a fencing instruction (MFENCE or SFENCE) before or after a direct store to enforce the ordering desired.

There is no atomicity guarantee provided for the 64-byte load operation from source address, and processor implementations may use multiple load operations to read the 64-bytes. The 64-byte direct-store issued by MOVDIR64B guarantees 64-byte write-completion atomicity. This means that the data arrives at the destination in a single undivided 64-byte write transaction.

Availability of the MOVDIR64B instruction is indicated by the presence of the CPUID feature flag MOVDIR64B (bit 28 of the ECX register in leaf 07H, see “CPUID—CPU Identification” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

Operation

DEST := SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIR64B void _movdir64b(void *dst, const void* src)

1. The Mod field of the ModR/M byte cannot have value 11B.
Protected Mode Exceptions
#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
    If address in destination (register) operand is not aligned to a 64-byte boundary.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#UD If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
    If LOCK prefix is used.

Real-Address Mode Exceptions
#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
    If address in destination (register) operand is not aligned to a 64-byte boundary.
#UD If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
    If LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.
#PF (fault-code) For a page fault.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If memory address referencing the SS segment is in non-canonical form.
#GP(0) If the memory address is in non-canonical form.
    If address in destination (register) operand is not aligned to a 64-byte boundary.
#PF (fault-code) For a page fault.
#UD If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
    If LOCK prefix is used.
## MOVDQA, VMOVQDA32/64—Move Aligned Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 6F /r MOVDQA xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move aligned packed integer values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>66 0F 7F /r MOVDQA xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move aligned packed integer values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 6F /r VMOVQDA xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 7F /r VMOVQDA xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 6F /r VMOVQDA ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from ymm2/mem to ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 7F /r VMOVQDA ymm2/m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move aligned packed integer values from ymm1 to ymm2/mem.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 6F /r VMOVQDA32 xmm1[k1]{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 6F /r VMOVQDA32 ymm1[k1]{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 6F /r VMOVQDA32 zmm1[k1]{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 7F /r VMOVQDA32 xmm2/m128[k1]{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7F /r VMOVQDA32 ymm2/m256[k1]{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7F /r VMOVQDA32 zmm2/m512[k1]{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 6F /r VMOVQDA64 xmm1[k1]{z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 6F /r VMOVQDA64 ymm1[k1]{z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 6F /r VMOVQDA64 zmm1[k1]{z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 7F /r VMOVQDA64 xmm2/m128[k1]{z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7F /r VMOVQDA64 ymm2/m256[k1]{z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7F /r VMOVQDA64 zmm2/m512[k1]{z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
</tbody>
</table>
Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQA32 (EVEX Encoded Versions, Register-Copy Form)

(KL, VL) = (4, 128), (8, 256), (16, 512)

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]

\[
i := j * 32
\]

\[
\text{IF } k1[j] \text{ OR *no writemask*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] := \text{SRC}[i+31:i]
\]

\[
\text{ELSE}
\]

\[
\text{IF *merging-masking* ; merging-masking}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged*}
\]

\[
\text{ELSE } \text{DEST}[i+31:i] := 0 ; \text{zeroing-masking}
\]

\[
\text{FI}
\]

\[
\text{FI};
\]
MOVDQA, VMOVDBQA32/64—Move Aligned Packed Integer Values

ENDFOR

DEST[MAXVL-1:VL] := 0

VMOVDBQA32 (EVEX Encoded Versions, Store-Form)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := SRC[i+31:i]
      ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
      FI;
  ENDFOR;

VMOVDBQA32 (EVEX Encoded Versions, Load-Form)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := SRC[i+31:i]
      ELSE IF *merging-masking* ; merging-masking
           THEN *DEST[i+31:i] remains unchanged*
           ELSE DEST[i+31:i] := 0 ; zeroing-masking
           FI
      FI;
  ENDFOR

DEST[MAXVL-1:VL] := 0

VMOVDBQA64 (EVEX Encoded Versions, Register-Copy Form)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := SRC[i+63:i]
      ELSE IF *merging-masking* ; merging-masking
           THEN *DEST[i+63:i] remains unchanged*
           ELSE DEST[i+63:i] := 0 ; zeroing-masking
           FI
      FI;
  ENDFOR

DEST[MAXVL-1:VL] := 0

VMOVDBQA64 (EVEX Encoded Versions, Store-Form)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := SRC[i+63:i]
      ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
      FI;
  ENDFOR;
VMOVDAQ64 (EVEK Encoded Versions, Load-Form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] := 0 ; zeroing-masking
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVDAQ (VEX.256 Encoded Version, Load and Register Copy)
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

VMOVDAQ (VEX.256 Encoded Version, Store-Form)
DEST[255:0] := SRC[255:0]

VMOVDAQ (VEX.128 Encoded Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

VMOVDAQ (128-bit Load- and Register-Copy-Form Legacy SSE Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVDQA (128-bit Store-Form Version)
DEST[127:0] := SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent
VMOVQDA32 __m512i _mm512_load_epi32( void * sa);
VMOVQDA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVQDA32 __m512i _mm512_maskz_load_epi32( __mmask16 k, void * sa);
VMOVQDA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVQDA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVQDA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVQDA32 __m256i _mm256_maskz_load_epi32( __mmask8 k, void * sa);
VMOVQDA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVQDA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVQDA32 __m128i _mm128_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVQDA32 __m128i _mm128_maskz_load_epi32( __mmask8 k, void * sa);
VMOVQDA32 void _mm128_store_epi32(void * d, __m128i a);
VMOVQDA32 void _mm_mask_load_epi32(void * d, __mmask8 k, __m128i a);
VMOVQDA64 __m512i _mm512_load_epi64( void * sa);
VMOVQDA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVQDA64 __m512i _mm512_maskz_load_epi64( __mmask8 k, void * sa);
VMOVQDA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVQDA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVQDA64 __m256i _mm256_mask_load_epi64(void * d, __mmask8 k, __m512i a);
VMOVQDA64 __m256i _mm256_maskz_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVQAD64 __m256i _mm256_maskz_load_epi64( __mmask8 k, void * sa);
VMOVQAD64 void _mm256_store_epi64( void * d, __m256i a);
VMOVQAD64 void _mm256_mask_store_epi64( void * d, __mmask8 k, __m256i a);
VMOVQAD64 __m128i _mm_mask_load_epi64( __m128i s, __mmask8 k, void * sa);
VMOVQAD64 __m128i _mm_maskz_load_epi64( __mmask8 k, void * sa);
VMOVQAD64 void _mm_store_epi64( void * d, __m128i a);
VMOVQAD64 void _mm_mask_store_epi64( void * d, __mmask8 k, __m128i a);
MOVQAD __m256i _mm256_load_si256( __m256i * p);
MOVQAD _mm256_store_si256( __m256i * p, __m256i a);
MOVQAD __m128i _mm_load_si128( __m128i * p);
MOVQAD void _mm_store_si128( __m128i * p, __m128i a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, ”Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-44, ”Type E1 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
## MOV_DQU, MOV_QDQU8, MOV_QDQU16, MOV_QDQU32—Move Unaligned Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 6F /r MOV_DQU xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move unaligned packed integer values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>F3 0F 7F /r MOV_QDQU xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move unaligned packed integer values from xmm1 to xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.F3 0F.WI G 6F /r MOV_QDQU xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from xmm2/m128 to xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3 0F.WI G 7F /r MOV_QDQU xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from xmm1 to xmm2/m128.</td>
</tr>
<tr>
<td>VEX.256.F3 0F.WI G 6F /r MOV_QDQU ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from ymm2/m256 to ymm1.</td>
</tr>
<tr>
<td>VEX.256.F3 0F.WI G 7F /r MOV_QDQU ymm2/m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed integer values from ymm1 to ymm2/m256.</td>
</tr>
<tr>
<td>EVEX.128.F2 0F.W0 6F /r MOV_QDQU8 xmm1[k1][z], xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2 0F.W0 6F /r MOV_QDQU8 ymm1[k1][z], ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2 0F.W0 6F /r MOV_QDQU8 zmm1[k1][z], zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F2 0F.W0 7F /r MOV_QDQU8 xmm1[k1][z], xmm2/m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2 0F.W0 7F /r MOV_QDQU8 ymm1[k1][z], ymm2/m256</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2 0F.W0 7F /r MOV_QDQU8 zmm1[k1][z], zmm2/m512</td>
<td>D</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F2 0F.W1 6F /r MOV_QDQU16 xmm1[k1][z], xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2 0F.W1 6F /r MOV_QDQU16 ymm1[k1][z], ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2 0F.W1 6F /r MOV_QDQU16 zmm1[k1][z], zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F2 0F.W1 7F /r MOV_QDQU16 xmm1[k1][z], xmm2/m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2 0F.W1 7F /r MOV_QDQU16 ymm1[k1][z], ymm2/m256</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2 0F.W1 7F /r MOV_QDQU16 zmm1[k1][z], zmm2/m512</td>
<td>D</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3 0F.W0 6F /r MOV_QDQU32 xmm1[k1][z], xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3 0F.W0 6F /r MOV_QDQU32 ymm1[k1][z], ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3 0F.W0 6F /r MOV_QDQU32 zmm1[k1][z], zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
**Description**

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**EVEX encoded versions:**

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

**VEX.256 encoded version:**

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.
128-bit versions:
Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.
When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

**VMOVQDQU (EVEX Encoded Versions, Register-Copy Form)**

$$(KL, VL) = (16, 128), (32, 256), (64, 512)$$

FOR $j := 0$ TO $KL-1$
  $i := j \times 8$
  IF $k1[j] \text{ OR *no writemask*}$$
    THEN $\text{DEST}[i+7:i] := \text{SRC}[i+7:i]$\n    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST*[i+7:i] remains unchanged*
        ELSE  DEST*[i+7:i] := 0 ; zeroing-masking
      FI
  FI;
ENDFOR

$\text{DEST}[\text{MAXVL}-1:VL] := 0$

**VMOVQDQU (EVEX Encoded Versions, Store-Form)**

$$(KL, VL) = (16, 128), (32, 256), (64, 512)$$

FOR $j := 0$ TO $KL-1$
  $i := j \times 8$
  IF $k1[j] \text{ OR *no writemask*}$$
    THEN $\text{DEST}[i+7:i] := \text{SRC}[i+7:i]$\n  ELSE *DEST*[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;

**VMOVQDQU (EVEX Encoded Versions, Load-Form)**

$$(KL, VL) = (16, 128), (32, 256), (64, 512)$$

FOR $j := 0$ TO $KL-1$
  $i := j \times 8$
  IF $k1[j] \text{ OR *no writemask*}$$
    THEN $\text{DEST}[i+7:i] := \text{SRC}[i+7:i]$\n    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST*[i+7:i] remains unchanged*
        ELSE  DEST*[i+7:i] := 0 ; zeroing-masking
      FI
  FI;
ENDFOR

$\text{DEST}[\text{MAXVL}-1:VL] := 0$
VMOVDQU16 (EVEX Encoded Versions, Register-Copy Form)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \(j := 0 \) TO \(KL-1\)
\[
i := j \times 16
\]
IF \(k1[j] \) OR *no writemask*
\[
\text{THEN } \text{DEST}[i+15:i] := \text{SRC}[i+15:i]
\]
ELSE
\[
\text{IF } *\text{merging-masking}* ; \text{merging-masking} \\
\text{THEN } *\text{DEST}[i+15:i] \text{ remains unchanged}* \\
\text{ELSE } \text{DEST}[i+15:i] := 0 ; \text{zeroing-masking}
\]
\]
\FI
\]
\FI
ENDFOR

\text{DEST}[\text{MAXVL-1:VL}] := 0

VMOVDQU16 (EVEX Encoded Versions, Store-Form)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \(j := 0 \) TO \(KL-1\)
\[
i := j \times 16
\]
IF \(k1[j] \) OR *no writemask*
\[
\text{THEN } \text{DEST}[i+15:i] := \text{SRC}[i+15:i]
\]
ELSE
\[
\text{FI}
\]
ENDFOR

VMOVDQU16 (EVEX Encoded Versions, Load-Form)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \(j := 0 \) TO \(KL-1\)
\[
i := j \times 16
\]
IF \(k1[j] \) OR *no writemask*
\[
\text{THEN } \text{DEST}[i+15:i] := \text{SRC}[i+15:i]
\]
ELSE
\[
\text{IF } *\text{merging-masking}* ; \text{merging-masking} \\
\text{THEN } *\text{DEST}[i+15:i] \text{ remains unchanged}* \\
\text{ELSE } \text{DEST}[i+15:i] := 0 ; \text{zeroing-masking}
\]
\]
\FI
\]
ENDFOR

\text{DEST}[\text{MAXVL-1:VL}] := 0

VMOVDQU32 (EVEX Encoded Versions, Register-Copy Form)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0 \) TO \(KL-1\)
\[
i := j \times 32
\]
IF \(k1[j] \) OR *no writemask*
\[
\text{THEN } \text{DEST}[i+31:i] := \text{SRC}[i+31:i]
\]
ELSE
\[
\text{IF } *\text{merging-masking}* ; \text{merging-masking} \\
\text{THEN } *\text{DEST}[i+31:i] \text{ remains unchanged}* \\
\text{ELSE } \text{DEST}[i+31:i] := 0 ; \text{zeroing-masking}
\]
\]
\FI
\]
ENDFOR
DEST[MAXVL-1:VL] := 0

**VMOVQDQU32 (EVEX Encoded Versions, Store-Form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
        SRC[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;

**VMOVQDQU32 (EVEX Encoded Versions, Load-Form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] := 0 ; zeroing-masking
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VMOVQDQU64 (EVEX Encoded Versions, Register-Copy Form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] := 0 ; zeroing-masking
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VMOVQDQU64 (EVEX Encoded Versions, Store-Form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR;
VMOVQDQU64 (EVEX Encoded Versions, Load-Form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] := SRC[i+63:i]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
  ELSE DEST[i+63:i] := 0 ; zeroing-masking
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVQDU (VEX.256 Encoded Version, Load- and Register Copy)
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

VMOVQDU (VEX.256 Encoded Version, Store-Form)
DEST[255:0] := SRC[255:0]

VMOVQDU (VEX.128 encoded version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

VMOVQDU (128-bit Load- and Register-Copy- Form Legacy SSE Version)
DEST[127:0] := SRC[127:0]
(UNMODIFIED)

(V)MOVQDU (128-bit Store-Form Version)
DEST[127:0] := SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQDU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVQDU16 __m512i _mm512_maskz_loadu_epi16( __mmask32 k, void * sa);
VMOVQDU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVQDU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVQDU16 __m256i _mm256_maskz_loadu_epi16( __mmask16 k, void * sa);
VMOVQDU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVQDU16 __m128i _mm128i_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVQDU16 __m128i _mm128i_maskz_loadu_epi16( __mmask8 k, void * sa);
VMOVQDU16 void _mm128_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVQDU32 __m512i _mm512_mask_loadu_epi32( void * sa);
VMOVQDU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVQDU32 __m512i _mm512_maskz_loadu_epi32(__mmask16 k, void * sa);
VMOVQDU32 void _mm512_mask_storeu_epi32(void * d, __m512i a);
VMOVQDU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVQDU32 __m256i _mm256_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVQDU32 void _mm256_mask_storeu_epi32(void * d, __m256i a);
VMOVQDU32 __m128i _mm128i_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);
VMOVQDU32 __m128i _mm128i_maskz_loadu_epi32(__mmask8 k, void * sa);
INSTRUCTION SET REFERENCE, M-U

VMOVDQU32 __m128i _mm_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVDQU32 void __m128i _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void __m128i _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void __m512i _mm_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void __m512i _mm_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8(__mmask64 k, void * sa);
VMOVDQU8 void _mm512i _mm_mask_storeu_epi8(__mmask64 k, void * sa);
VMOVDQU8 void __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8(__mmask32 k, void * sa);
VMOVDQU8 void _mm256i _mm_mask_storeu_epi8(__mmask32 k, void * sa);
VMOVDQU8 __m128i _mm128_maskz_loadu_epi8(__mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
VMOVQU __m256i _mm256_loadu_si256(__m256i * p);
VMOVQU __m256i _mm256_storeu_si256(__m256i *p, __m256i a);
VMOVQU __m128i _mm128_loadu_si128(__m128i * p);
VMOVQU __m128i _mm128_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F D6 /r</td>
<td>MOVDQ2Q mm, xmm</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move low quadword from xmm to mmx register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST := SRC[63:0];

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVDQ2Q __m64 _mm_movepi64_pi64 ( __m128i a)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM  If CR0.TS[bit 3] = 1.

#UD  If CR0.EF[bit 2] = 1.

     If CR4.OSFXSR[bit 9] = 0.

     If CPUID.01H:EDX.SSE2[bit 26] = 0.

     If the LOCK prefix is used.

#MF  If there is a pending x87 FPU exception.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
### MOVHLPS—Move Packed Single Precision Floating-Point Values High to Low

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 12 /r MOVHLPS xmm1, xmm2</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single precision floating-point values from high quadword of xmm2 to low quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single precision floating-point values from high quadword of xmm3 and low quadword of xmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r) / EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form:

Moves two packed single precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

### Operation

**MOVHLPS (128-bit Two-Argument Form)**

DEST[63:0] := SRC[127:64]

DEST[MAXVL-1:64] (Unmodified)

**VMOVHLPS (128-bit Three-Argument Form - VEX & EVEX)**

DEST[63:0] := SRC2[127:64]

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

### SIMD Floating-Point Exceptions

None.

1. ModRM.MOD = 011B required.
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:

#UD  If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-55, “Type E7NM Class Exception Conditions.”
MOVHPD—Move High Packed Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 16 /r MOVHPD xmm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move double precision floating-point value from m64 to high quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G 16 /r VMOVHPD xmm2, xmm1, m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge double precision floating-point value from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge double precision floating-point value from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>66 0F 17 /r MOVHPD m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move double precision floating-point value from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G 17 /r VMOVHPD m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move double precision floating-point value from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move double precision floating-point value from high quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:
Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:
Loads a double precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:
Stores a double precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD. If VMOVHPD is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.
**Operation**

**MOVHPD (128-bit Legacy SSE Load)**
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

**VMOVHPD (VEX.128 & EVEX Encoded Load)**
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0

**VMOVHPD (Store)**
DEST[63:0] := SRC[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**
MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:
#UD If VEX.L = 1.
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
MOVHPS—Move High Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 16 /r MOVHPS xmm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single precision floating-point values from m64 to high quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single precision floating-point values from m64 and the low quadword of xmm1.</td>
</tr>
<tr>
<td>NP 0F 17 /r MOVHPS m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single precision floating-point values from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move two packed single precision floating-point values from high quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move two packed single precision floating-point values from high quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMtr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRMtr/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple2</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Tuple2</td>
<td>ModRMtr/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:
Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:
Loads two single precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:
Stores two packed single precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.
Operation

MOVHPS (128-bit Legacy SSE Load)
DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

VMOVHPS (VEX.128 and EVEX Encoded Load)
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0

VMOVHPS (Store)
DEST[63:0] := SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
MOVHPS __m128 _mm_loadh_pi ( __m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi ( __m64 *p, __m128 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:
#UD If VEX.L = 1.
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
**MOVLHPS—Move Packed Single Precision Floating-Point Values Low to High**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 16 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single precision floating-point values from low quadword of xmm2 to high quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 16 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 16 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single precision floating-point values from low quadword of xmm3 and low quadword of xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMreg(r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r) / EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM argument (first operand). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.

**Operation**

**MOVLHPS (128-bit Two-Argument Form)**

DEST[63:0] (Unmodified)
DEST[127:64] := SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

**VMOVLHPS (128-bit Three-Argument Form - VEX & EVEX)**

DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

**SIMD Floating-Point Exceptions**

None.

1. ModRM.MOD = 011b required
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions,” additionally:
#UD  If VEX.L = 1.
EVEX-encoded instruction, see Exceptions Type E7NM.128 in Table 2-55, “Type E7NM Class Exception Conditions.”
MOVLPD—Move Low Packed Double Precision Floating-Point Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 12 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move double precision floating-point value from m64 to low quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1 13 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge double precision floating-point value from m64 and the high quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 13 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge double precision floating-point value from m64 and the high quadword of xmm1.</td>
</tr>
<tr>
<td>66 0F 13 /r</td>
<td>C</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move double precision floating-point value from low quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1 13 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move double precision floating-point value from low quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:mr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:mr/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:mr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRM:mr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:mr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Tuple1 Scalar</td>
<td>ModRM:mr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:
Moves a double precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:
Loads a double precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:
Stores a double precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'= 1 will cause an #UD exception.

**Operation**

**MOVLPD (128-bit Legacy SSE Load)**

DEST[63:0] := SRC[63:0]

DEST[MAXVL-1:64] (Unmodified)
**VMOVLPD (VEX.128 & EVEX Encoded Load)**
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**VMOVLPD (Store)**
DEST[63:0] := SRC[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**
MOVLPD __m128d __mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD If VEX.L = 1.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
MOVLPS—Move Low Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 12 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single precision floating-point values from m64 to low quadword of xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 12 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.</td>
</tr>
<tr>
<td>EVEX.128.0F.WO 12 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge two packed single precision floating-point values from m64 and the high quadword of xmm1.</td>
</tr>
<tr>
<td>0F 13/r</td>
<td>C</td>
<td>V/V</td>
<td>SSE</td>
<td>Move two packed single precision floating-point values from low quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 13/r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move two packed single precision floating-point values from low quadword of xmm1 to m64.</td>
</tr>
<tr>
<td>EVEX.128.0F.WO 13/r</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move two packed single precision floating-point values from low quadword of xmm1 to m64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple2</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Tuple2</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:
Moves two packed single precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:
Loads two packed single precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:
Loads two packed single precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L’L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L’L= 1 will cause an #UD exception.
**Operation**

**MOVLPS (128-bit Legacy SSE Load)**

DEST[63:0] := SRC[63:0]

DEST[MAXVL-1:64] (Unmodified)

**VMOVLPS (VEX.128 & EVEX Encoded Load)**

DEST[63:0] := SRC2[63:0]

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0

**VMOVLPS (Store)**

DEST[63:0] := SRC[63:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVLPS __m128 _mm_loadl_pi ( __m128 a, __m64 *p)

MOVLPS void _mm_storl_pi ( __m64 *p, __m128 a)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:

#UD If VEX.L = 1.

EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
**Instruction**: MOVMSKPD—Extract Packed Double Precision Floating-Point Sign Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 50 /r MOVMSKPD reg, xmm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Extract 2-bit sign mask from xmm and store in reg. The upper bits of r32 or r64 are filled with zeros.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 50 /r VMOVMSKPD reg, xmm2</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 2-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 50 /r VMOVMSKPD reg, ymm2</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 4-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Extracts the sign bits from the packed double precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**(V)MOVMSKPD (128-bit Versions)**

\[
\text{DEST}[0] := \text{SRC}[63] \\
\text{DEST}[1] := \text{SRC}[127] \\
\text{IF DEST} = \text{r32} \\
\quad \text{THEN DEST}[31:2] := 0; \\
\quad \text{ELSE DEST}[63:2] := 0; \\
\text{FI}
\]

**(V)MOVSXKPD (VEX.256 Encoded Version)**

\[
\text{DEST}[0] := \text{SRC}[63] \\
\text{DEST}[1] := \text{SRC}[127] \\
\text{DEST}[2] := \text{SRC}[191] \\
\text{DEST}[3] := \text{SRC}[255] \\
\text{IF DEST} = \text{r32} \\
\quad \text{THEN DEST}[31:4] := 0; \\
\quad \text{ELSE DEST}[63:4] := 0; \\
\text{FI}
\]
Intel C/C++ Compiler Intrinsic Equivalent
MOVMSKPD int _mm_movemask_pd (__m128d a)
VMOVMSKPD __m256_movemask_pd(__m256d a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-24, "Type 7 Class Exception Conditions," additionally:
#UD If VEX.vvvv ≠ 1111B.
MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 50 /r MOVMSKPS reg, xmm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Extract 4-bit sign mask from xmm and store in reg. The upper bits of r32 or r64 are filled with zeros.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 50 /r VMOVMSKPS reg, xmm2</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 4-bit sign mask from xmm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 50 /r VMOVMSKPS reg, ymm2</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of r32 or r64 are zeroed.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

1. ModRM.MOD = 011B required

Description

Extracts the sign bits from the packed single precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

DEST[0] := SRC[31];
DEST[1] := SRC[63];
DEST[2] := SRC[95];
DEST[3] := SRC[127];
IF DEST = r32
Ft;

1. ModRM.MOD = 011B required
(V)MOVMSKPS (128-bit version)
DEST[0] := SRC[31]
DEST[1] := SRC[63]
DEST[2] := SRC[95]
IF DEST = r32
   THEN DEST[31:4] := 0;
   ELSE DEST[63:4] := 0;
FI

VMOVMSKPS (VEX.256 encoded version)
DEST[0] := SRC[31]
DEST[1] := SRC[63]
DEST[2] := SRC[95]
IF DEST = r32
   THEN DEST[31:8] := 0;
   ELSE DEST[63:8] := 0;
FI

Intel C/C++ Compiler Intrinsic Equivalent
int _mm_movemask_ps(__m128 a)
int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-24, "Type 7 Class Exception Conditions," additionally:
#UD If VEX.vvvv ≠ 1111B.
MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 2A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1G 2A /r VMOVNTDQA xmm1, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move double quadword from m128 to xmm using non-temporal hint if WC memory type.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1G 2A /r VMOVNTDQA ymm1, m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementa-

1. ModRM.MOD != 011B
tion may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.
The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.
The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

**Operation**

**MOVNTDQA (128bit- Legacy SSE Form)**

DEST := SRC
DEST[MAXVL-1:128] (Unmodified)

**VMOVNTDQA (VEX.128 and EVEX.128 Encoded Form)**

DEST := SRC
DEST[MAXVL-1:128] := 0

**VMOVNTDQA (VEX.256 and EVEX.256 Encoded Forms)**

DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

**VMOVNTDQA (EVEX.512 Encoded Form)**

DEST[511:0] := SRC[511:0]
DEST[MAXVL-1:512] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTDQA __m512i _mm512_stream_load_si512(__m512i const* p);
MOVNTDQA __m128i _mm_stream_load_si128 (const __m128i *p);
VMOVNTDQA __m256i _mm256_stream_load_si256 (__m256i const* p);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”
Additionally:

#UD  If VEX.vvvv != 1111B or EVEX.vvv != 1111B.
MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F E7 /r MOVNTDQ m128, xmm1</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move packed integer values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed integer values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed integer values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed integer values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed integer values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move packed integer values in zmm1 to m512 using non-temporal hint.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTDQ(EVEX Encoded Versions)

VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0

1. ModRM.MOD != 011B
MOVNTDQ (Legacy and VEX Versions)
DEST := SRC

Intel C/C++ Compiler Intrinsic Equivalent
VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);
VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);
MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B or EVEX.vvv != 111B.
MOVNTI—Store Doubleword Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C3 i r MOVNTI m32, r32</td>
<td>MR</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move doubleword from r32 to m32 using non-temporal hint.</td>
</tr>
<tr>
<td>NP REX.W + 0F C3 i r MOVNTI m64, r64</td>
<td>MR</td>
<td>V/NI.E.</td>
<td>SSE2</td>
<td>Move quadword from r64 to m64 using non-temporal hint.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncachable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST := SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI void _mm_stream_si32 (int *p, int a)
MOVNTI void _mm_stream_si64 (__int64 *p, __int64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0) For an illegal address in the SS segment.
#PF(fault-code) For a page fault.
#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
   If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in real address mode.
#PF(fault-code) For a page fault.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.
If the LOCK prefix is used.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
MOVNTPD—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move packed double precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>MOVNTPD m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed double precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPD m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed double precision values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPD m256, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed double precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPD m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed double precision values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPD m256, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move packed double precision values in zmm1 to m512 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPD m512, zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding¹**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Moves the packed double precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

**Operation**

**VMOVNTPD (EVEX Encoded Versions)**

VL = 128, 256, 512

DEST[VL-1:0] := SRC[VL-1:0]

DEST[MAXVL-1:VL] := 0

---

¹ ModRM.MOD != 011B
MOVNTPD (Legacy and VEX Versions)
DEST := SRC

Intel C/C++ Compiler Intrinsic Equivalent
VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE2 in Table 2-18, "Type 1 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-45, "Type E1NF Class Exception Conditions."
Additionally:
#UD If VEX.vvv != 111B or EVEX.vvv != 111B.
MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move packed single precision values xmm1 to mem using non-temporal hint.</td>
</tr>
<tr>
<td>MOVNTPS m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.0F.WIG 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed single precision values xmm1 to mem using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPS m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.0F.WIG 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move packed single precision values ymm1 to mem using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPS m256, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.0F.W0 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed single precision values in xmm1 to m128 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPS m128, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W0 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move packed single precision values in ymm1 to m256 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPS m256, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W0 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move packed single precision values in zmm1 to m512 using non-temporal hint.</td>
</tr>
<tr>
<td>VMOVNTPS m512, zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Description

Moves the packed single precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. Written to memory, the data type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VMOVNTPS (EVEX Encoded Versions)
VL = 128, 256, 512
DEST[VL-1:0] := SRC[VL-1:0]
DEST[MAXVL-1:VL] := 0

¹. ModRM.MOD !≡ 011B
MOVNTPS
DEST := SRC

Intel C/C++ Compiler Intrinsic Equivalent
VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type1.SSE in Table 2-18, “Type 1 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-45, “Type E1NF Class Exception Conditions.”
Additionally:
#UD If VEX.vvv != 1111B or EVEX.vvv != 1111B.
MOVNTQ—Store of Quadword Using Non-Temporal Hint

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E7 /r</td>
<td>MOVNTQ m64, mm</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Move quadword from mm to m64 using non-temporal hint.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

DEST := SRC;

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVNTQ void _mm_stream_pi(__m64 * p, __m64 a)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 23-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
MOVQ—Move Quadword

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 6F /r</td>
<td>MOVQ mm, mm/m64</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>NP 0F 7F /r</td>
<td>MOVQ mm/m64, mm</td>
<td>N/A</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>F3 0F 7E /r</td>
<td>MOVQ xmm1, xmm2/m64</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>VEX.128.F3.0F.W1 7E /r</td>
<td>VMOVQ xmm1, xmm2/m64</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 D6 /r</td>
<td>VMOVQ xmm1/m64, xmm2</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 D6 /r</td>
<td>VMOVQ xmm1/m64, xmm2</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

**Operation**

**MOVQ Instruction When Operating on MMX Technology Registers and Memory Locations**

```plaintext
DEST := SRC;
```
MOVQ Instruction When Source and Destination Operands are XMM Registers
  DEST[63:0] := SRC[63:0];
  DEST[127:64] := 0000000000000000H;

MOVQ Instruction When Source Operand is XMM Register and Destination operand is memory location:
  DEST := SRC[63:0];

MOVQ Instruction When Source Operand is Memory Location and Destination operand is XMM register:
  DEST[63:0] := SRC;
  DEST[127:64] := 0000000000000000H;

VMOVQ (VEX.128.F3.0F 7E) With XMM Register Source and Destination
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (VEX.128.66.0F 7E) With XMM Register Source and Destination
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (VEX.128.66.0F D6) With XMM Register Source and Destination
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (7E - EVEX Encoded Version) With XMM Register Source and Destination
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (D6 - EVEX Encoded Version) With XMM Register Source and Destination
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (7E) With Memory Source
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (7E - EVEX Encoded Version) With Memory Source
  DEST[63:0] := SRC[63:0]
  DEST[MAXVL-1:64] := 0

VMOVQ (D6) With Memory DEST
  DEST[63:0] := SRC2[63:0]

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
VMOVQ __m128i _mm_loadu_si64( void * s);
VMOVQ void _mm_storeu_si64( void * d, __m128i s);
MOVQ m128i _mm_move_epi64(__m128i a)

SIMD Floating-Point Exceptions
None.
Other Exceptions
See Table 23-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF D6 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move quadword from mm to low quadword of xmm.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

DEST[63:0] := SRC[63:0];
DEST[127:64] := 00000000000000000H;

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVQ2DQ _128i _mm_movpi64_epi64 ( __m64 a)

**SIMD Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM If CR0.TS[bit 3] = 1.

#UD If CR0.EM[bit 2] = 1.

#MF If there is a pending x87 FPU exception.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions

Same exceptions as in protected mode.
## Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct location. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incre-
mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

NOTE
To improve performance, more recent processors support modifications to the processor’s operation during the string store operations initiated with MOVS and MOVSB. See Section 7.3.9.3 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional information on fast-string operation.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction’s default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
DEST := SRC;
Non-64-bit Mode:
IF (Byte move)
 THEN IF DF = 0
 THEN
 (E)SI := (E)SI + 1;
 (E)DI := (E)DI + 1;
 ELSE
 (E)SI := (E)SI - 1;
 (E)DI := (E)DI - 1;
 FI;
 ELSE IF (Word move)
 THEN IF DF = 0
 THEN
 (E)SI := (E)SI + 2;
 (E)DI := (E)DI + 2;
 FI;
 ELSE
 (E)SI := (E)SI - 2;
 (E)DI := (E)DI - 2;
 FI;
 ELSE IF (Doubleword move)
 THEN IF DF = 0
 THEN
 (E)SI := (E)SI + 4;
 (E)DI := (E)DI + 4;
 FI;
 ELSE
 (E)SI := (E)SI - 4;
 (E)DI := (E)DI - 4;
 FI;
 FI;
64-bit Mode:
IF (Byte move)
 THEN IF DF = 0
 THEN
 (R(E)SI := (R(E)SI + 1;
 (R(E)DI := (R(E)DI + 1;
ELSE
    (R|E)SI := (R|E)SI - 1;
    (R|E)DI := (R|E)DI - 1;
    Fl;
ELSE IF (Word move)
    THEN IF DF = 0
        (R|E)SI := (R|E)SI + 2;
        (R|E)DI := (R|E)DI + 2;
        Fl;
    ELSE
        (R|E)SI := (R|E)SI - 2;
        (R|E)DI := (R|E)DI - 2;
        Fl;
    ELSE IF (Doubleword move)
    THEN IF DF = 0
        (R|E)SI := (R|E)SI + 4;
        (R|E)DI := (R|E)DI + 4;
        Fl;
    ELSE
        (R|E)SI := (R|E)SI - 4;
        (R|E)DI := (R|E)DI - 4;
        Fl;
    ELSE IF (Quadword move)
    THEN IF DF = 0
        (R|E)SI := (R|E)SI + 8;
        (R|E)DI := (R|E)DI + 8;
        Fl;
    ELSE
        (R|E)SI := (R|E)SI - 8;
        (R|E)DI := (R|E)DI - 8;
        Fl;
    Fl;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
MOVSD—Move or Merge Scalar Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move scalar double precision floating-point value from xmm2 to xmm1 register.</td>
</tr>
<tr>
<td>MOVSD xmm1, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Load scalar double precision floating-point value from m64 to xmm1 register.</td>
</tr>
<tr>
<td>MOVSD xmm1, m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2 0F 11 /r</td>
<td>C</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move scalar double precision floating-point value from xmm2 register to xmm1/m64.</td>
</tr>
<tr>
<td>MOVSD xmm1/m64, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 10 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge scalar double precision floating-point value from xmm2 and xmm3 to xmm1 register.</td>
</tr>
<tr>
<td>VMOVSD xmm1, xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 10 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Load scalar double precision floating-point value from m64 to xmm1 register.</td>
</tr>
<tr>
<td>VMOVSD xmm1, m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 11 /r</td>
<td>E</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1.</td>
</tr>
<tr>
<td>VMOVSD xmm1, xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 11 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Store scalar double precision floating-point value from xmm1 register to m64.</td>
</tr>
<tr>
<td>VMOVSD m64, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 10 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VMOVSD xmm1 (k1)[z], xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 10 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Load scalar double precision floating-point value from m64 to xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>VMOVSD xmm1 (k1)[z], m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 11 /r</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Merge scalar double precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VMOVSD xmm1 (k1)[z], xmm2, xmm3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 11 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Store scalar double precision floating-point value from xmm1 register to m64 under writemask k1.</td>
</tr>
<tr>
<td>VMOVSD m64 (k1), xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRMreg/r (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>N/A</td>
<td>ModRMreg/r (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>G</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg/r (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Moves a scalar double precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM...
registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation

VMOVSD (EVEX.LLIG.F2.0F 10 /r: VMOVSD xmm1, m64 With Support for 32 Registers)
IF k1[0] or *no writemask*
    THEN DEST[63:0] := SRC[63:0]
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
    FI;
FI;
DEST[MAXVL-1:64] := 0

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD m64, xmm1 With Support for 32 Registers)
IF k1[0] or *no writemask*
    THEN DEST[63:0] := SRC[63:0]
ELSE *DEST[63:0] remains unchanged* ; merging-masking
FI;

VMOVSD (EVEX.LLIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
IF k1[0] or *no writemask*
    THEN DEST[63:0] := SRC2[63:0]
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
    FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, xmm2)
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

VMOVSD (VEX.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

MOVSD—Move or Merge Scalar Double Precision Floating-Point Value
VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] := SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

VMOVSD (VEX.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] := SRC[63:0]
DEST[MAXVL-1:64] := 0

MOVSD/VMOVSD (128-bit Versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)
DEST[63:0] := SRC[63:0]

MOVSD (128-bit Legacy SSE Version: MOVSD xmm1, m64)
DEST[63:0] := SRC[63:0]
DEST[127:64] := 0
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd( __mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd( __mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd ( __m128d a, __m128d b)

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:
  #UD  IF VEX.vvvv != 1111B.
EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”
MOVSHDUP—Replicate Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE3</td>
<td>Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F.W1G 16 /r VMOVSHDUP xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move odd index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F.W1G 16 /r VMOVSHDUP ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move odd index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 [k1][z], xmm2/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move odd index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 [k1][z], ymm2/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move odd index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 [k1][z], zmm2/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move odd index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Duplicates odd-indexed single precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Figure 4-3. MOVSHDUP Operation
Operation

**VMOVSHDUP (EVEX Encoded Versions)**

$(KL, VL) = (4, 128), (8, 256), (16, 512)$

$\text{TMP\_SRC}[31:0] := \text{SRC}[63:32]$

$\text{TMP\_SRC}[95:64] := \text{SRC}[127:96]$
$\text{TMP\_SRC}[127:96] := \text{SRC}[127:96]$

If $VL \geq 256$

$\text{TMP\_SRC}[159:128] := \text{SRC}[191:160]$
$\text{TMP\_SRC}[191:160] := \text{SRC}[191:160]$
$\text{TMP\_SRC}[223:192] := \text{SRC}[255:224]$
$\text{TMP\_SRC}[255:224] := \text{SRC}[255:224]$

$\text{FI;}$

If $VL \geq 512$

$\text{TMP\_SRC}[287:256] := \text{SRC}[319:288]$
$\text{TMP\_SRC}[319:288] := \text{SRC}[319:288]$
$\text{TMP\_SRC}[351:320] := \text{SRC}[383:352]$
$\text{TMP\_SRC}[383:352] := \text{SRC}[383:352]$
$\text{TMP\_SRC}[415:384] := \text{SRC}[447:416]$
$\text{TMP\_SRC}[447:416] := \text{SRC}[447:416]$
$\text{TMP\_SRC}[479:448] := \text{SRC}[511:480]$
$\text{TMP\_SRC}[511:480] := \text{SRC}[511:480]$

$\text{FI;}$

For $j := 0$ to $KL-1$

$i := j \times 32$

If $\text{k1}[j]$ OR *no writemask*

Then $\text{DEST}[i+31:i] := \text{TMP\_SRC}[i+31:i]$

Else

If *merging-masking* ; merging-masking

Then *DEST*[i+31:i] remains unchanged*

Else ; zeroing-masking

$\text{DEST}[i+31:i] := 0$

$\text{FI}$

$\text{FI;}$

$\text{ENDFOR}$

$\text{DEST}[\text{MAXVL}-1:VL] := 0$

**VMOVSHDUP (VEX.256 Encoded Version)**

$\text{DEST}[31:0] := \text{SRC}[63:32]$
$\text{DEST}[63:32] := \text{SRC}[63:32]$
$\text{DEST}[95:64] := \text{SRC}[127:96]$
$\text{DEST}[127:96] := \text{SRC}[127:96]$
$\text{DEST}[159:128] := \text{SRC}[191:160]$
$\text{DEST}[191:160] := \text{SRC}[191:160]$
$\text{DEST}[223:192] := \text{SRC}[255:224]$
$\text{DEST}[255:224] := \text{SRC}[255:224]$
$\text{DEST}[\text{MAXVL}-1:256] := 0$

**VMOVSHDUP (VEX.128 Encoded Version)**

$\text{DEST}[31:0] := \text{SRC}[63:32]$
$\text{DEST}[63:32] := \text{SRC}[63:32]$
$\text{DEST}[95:64] := \text{SRC}[127:96]$
$\text{DEST}[127:96] := \text{SRC}[127:96]$
$\text{DEST}[\text{MAXVL}-1:128] := 0$
MOVSHDUP (128-bit Legacy SSE Version)

```
DEST[31:0] := SRC[63:32]
DEST[95:64] := SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VMOVSHDUP __m512 _mm512_movehdup_ps(__m512 a);
VMOVSHDUP __m512 _mm512_mask_movehdup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSHDUP __m512 _mm512_maskz_movehdup_ps(__mmask16 k, __m512 a);
VMOVSHDUP __m256 _mm256_movehdup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_mask_movehdup_ps(__mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_maskz_movehdup_ps(__mmask8 k, __m256 a);
VMOVSHDUP __m128 _mm128_movesd(__m128 s, __mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm128_mask_movehdup_ps(__mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm128_maskz_movehdup_ps(__mmask8 k, __m128 a);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
Additionally:

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
## MOVSLDUP—Replicate Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 12/r MOVSLDUP xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE3</td>
<td>Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F.WIG 12/r VMOVSLDUP xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move even index single precision floating-point values from xmm2/mem and duplicate each element into xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F.WIG 12/r VMOVSLDUP ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move even index single precision floating-point values from ymm2/mem and duplicate each element into ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F.W0 12/r VMOVSLDUP xmm1 [k1][z], xmm2/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move even index single precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 12/r VMOVSLDUP ymm1 [k1][z], ymm2/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move even index single precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 12/r VMOVSLDUP zmm1 [k1][z], zmm2/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move even index single precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Duplicates even-indexed single precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Figure 4-4. MOVSLDUP Operation

![MOVSLDUP Operation Diagram](image-url)
Operation

VMOVSLDUP (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP_SRC[31:0] := SRC[31:0]
TMP_SRC[63:32] := SRC[31:0]
TMP_SRC[95:64] := SRC[95:64]
TMP_SRC[127:96] := SRC[95:64]

IF VL >= 256
FI;

IF VL >= 512
    TMP_SRC[479:448] := SRC[479:448]
    TMP_SRC[511:480] := SRC[479:448]
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_SRC[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVSLDUP (VEX.256 Encoded Version)

DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:256] := 0

VMOVSLDUP (VEX.128 Encoded Version)

DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] := 0
MOVSLDUP (128-bit Legacy SSE Version)
DEST[31:0] := SRC[31:0]
DEST[63:32] := SRC[31:0]
DEST[95:64] := SRC[95:64]
DEST[127:96] := SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMOVSQDUP __m512 __m128_mm512_moveldup_ps(__m512 a);
VMOVSQDUP __m512 __m128_mm512_mask_moveldup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSQDUP __m512 __m128_mm512_maskz_moveldup_ps(__mmask16 k, __m512 a);
VMOVSQDUP __m256 __m256_mm256_moveldup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSQDUP __m256 __m256_mm256_mask_moveldup_ps(__mmask8 k, __m256 a);
VMOVSQDUP __m256 __m256_mm256_maskz_moveldup_ps(__mmask8 k, __m256 a);
VMOVSQDUP __m128 __m128_mm128_moveldup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSQDUP __m128 __m128_mm128_mask_moveldup_ps(__mmask8 k, __m128 a);
VMOVSQDUP __m128 __m128_mm128_maskz_moveldup_ps(__m128 a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
MOVSS—Move or Merge Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 OF 10 /r MOVSS xmm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Merge scalar single precision floating-point value from xmm2 to xmm1 register.</td>
</tr>
<tr>
<td>F3 OF 10 /r MOVSS xmm1, m32</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Load scalar single precision floating-point value from m32 to xmm1 register.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Merge scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Load scalar single precision floating-point value from m32 to xmm1 register.</td>
</tr>
<tr>
<td>F3 OF 11 /r MOVSS xmm2/m32, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>SSE</td>
<td>Move scalar single precision floating-point value from xmm1 register to xmm2/m32.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3</td>
<td>E</td>
<td>V/V</td>
<td>AVX</td>
<td>Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Move scalar single precision floating-point value from xmm1 register to m32.</td>
</tr>
<tr>
<td>EVE.XLLG.F3.0F.W0 10 /r VMOVSS xmm1 <a href="z">k1</a>, xmm2, xmm3</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVE.XLLG.F3.0F.W0 10 /r VMOVSS m32 <a href="z">k1</a>, m32</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single precision floating-point values from m32 to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVE.XLLG.F3.0F.W0 11 /r VMOVSS xmm1 <a href="z">k1</a>, xmm2, xmm3</td>
<td>E</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVE.XLLG.F3.0F.W0 11 /r VMOVSS m32 [k1], xmm1</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move scalar single precision floating-point values from xmm1 to m32 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>G</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Moves a scalar single precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination
operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction “VMOVSS m32, xmm1”, VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction “VMOVSS mv {k1}, xmm1”, EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMOVSS (EVEX.LLIG.F3.0F.W0 11 /r When the Source Operand is Memory and the Destination is an XMM Register)

\[
\text{IF } k1[0] \text{ or } \text{no writemask} \text{ THEN } \text{DEST}[31:0] := \text{SRC}[31:0] \\
\text{ELSE} \\
\text{IF } \text{merging-masking} \text{ THEN } \text{DEST}[31:0] \text{ remains unchanged} \\
\text{ELSE } \text{zeroing-masking} \text{ THEN } \text{DEST}[31:0] := 0 \\
\text{FI;} \\
\text{FI;} \\
\text{DEST}[\text{MAXVL}-1:32] := 0
\]

VMOVSS (EVEX.LLIG.F3.0F.W0 10 /r When the Source Operand is an XMM Register and the Destination is Memory)

\[
\text{IF } k1[0] \text{ or } \text{no writemask} \text{ THEN } \text{DEST}[31:0] := \text{SRC}[31:0] \\
\text{ELSE } \text{DEST}[31:0] \text{ remains unchanged} \text{ merging-masking} \\
\text{FI;}
\]

VMOVSS (EVEX.LLIG.F3.0F.W0 10/11 /r Where the Source and Destination are XMM Registers)

\[
\text{IF } k1[0] \text{ or } \text{no writemask} \text{ THEN } \text{DEST}[31:0] := \text{SRC2}[31:0] \\
\text{ELSE} \\
\text{IF } \text{merging-masking} \text{ THEN } \text{DEST}[31:0] \text{ remains unchanged} \\
\text{ELSE } \text{zeroing-masking} \text{ THEN } \text{DEST}[31:0] := 0 \\
\text{FI;} \\
\text{FI;} \\
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

MOVSS (Legacy SSE Version When the Source and Destination Operands are Both XMM Registers)

\[
\text{DEST}[31:0] := \text{SRC}[31:0] \\
\text{DEST}[\text{MAXVL}-1:32] \text{ (Unmodified)}
\]
**VMOVSS (VEX.128.F3.0F 11 /r Where the Destination is an XMM Register)**
DEST[31:0] := SRC2[31:0]
DEST[MAXVL-1:128] := 0

**VMOVSS (VEX.128.F3.0F 10 /r Where the Source and Destination are XMM Registers)**
DEST[31:0] := SRC2[31:0]
DEST[MAXVL-1:128] := 0

**VMOVSS (VEX.128.F3.0F 10 /r When the Source Operand is Memory and the Destination is an XMM Register)**
DEST[31:0] := SRC[31:0]
DEST[MAXVL-1:32] := 0

**MOVSS/VMOVSS (When the Source Operand is an XMM Register and the Destination is Memory)**
DEST[31:0] := SRC[31:0]

**MOVSS (Legacy SSE Version when the Source Operand is Memory and the Destination is an XMM Register)**
DEST[31:0] := SRC[31:0]
DEST[127:32] := 0
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss( __mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss( __mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions,” additionally:
#UD If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”
MOVSX/MOVSXD—Move With Sign-Extension

**Description**
Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**
DEST := SignExtend(SRC);

**Flags Affected**
None.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

---

**Table: Instruction Encoding**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F BE /r</td>
<td>MOVSX r16, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to word with sign-extension.</td>
</tr>
<tr>
<td>0F BE /r</td>
<td>MOVSX r32, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to doubleword with sign-extension.</td>
</tr>
<tr>
<td>REX.W + 0F BE /r</td>
<td>MOVSX r64, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move byte to quadword with sign-extension.</td>
</tr>
<tr>
<td>0F BF /r</td>
<td>MOVSX r32, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move word to doubleword, with sign-extension.</td>
</tr>
<tr>
<td>REX.W + 0F BF /r</td>
<td>MOVSX r64, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move word to quadword with sign-extension.</td>
</tr>
<tr>
<td>63 /r¹</td>
<td>MOVSX r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move word to word with sign-extension.</td>
</tr>
<tr>
<td>63 /r¹</td>
<td>MOVSX r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move doubleword to doubleword with sign-extension.</td>
</tr>
<tr>
<td>REX.W + 63 /r</td>
<td>MOVSX r64, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move doubleword to quadword with sign-extension.</td>
</tr>
</tbody>
</table>

NOTES:
1. The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

---

**Table: Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#UD** If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
MOVUPD—Move Unaligned Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Op / En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.</td>
</tr>
</tbody>
</table>

EVEX.512 encoded version:

Moves 512 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.
VEX.256 encoded version:
Moves 256 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:
Moves 128 bits of packed double precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.
When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

**VMOVUPD (EVEX Encoded Versions, Register-Copy Form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR \( j := 0 \) TO KL-1

\( i := j \times 64 \)

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+63:i] := \text{SRC}[i+63:i] \)

ELSE

IF *merging-masking*  
THEN \( \text{DEST}[i+63:i] \) remains unchanged*

ELSE \( \text{DEST}[i+63:i] := 0 \)  ; zeroing-masking

FI

FI;
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:V] := 0 \)

**VMOVUPD (EVEX Encoded Versions, Store-Form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR \( j := 0 \) TO KL-1

\( i := j \times 64 \)

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+63:i] := \text{SRC}[i+63:i] \)

ELSE \( \text{DEST}[i+63:i] \) remains unchanged*  ; merging-masking

FI;
ENDFOR;
VMOVUPD (EVEX Encoded Versions, Load-Form)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE DEST[i+63:i] := 0 ; zeroing-masking
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVUPD (VEX.256 Encoded Version, Load- and Register Copy)
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

VMOVUPD (VEX.256 Encoded Version, Store-Form)
DEST[255:0] := SRC[255:0]

VMOVUPD (VEX.128 Encoded Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

MOVUPD (128-bit Load- and Register-Copy- Form Legacy SSE Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVUPD (128-bit Store-Form Version)
DEST[127:0] := SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent
VMOVUPD __m512d _mm512_loadu_pd( void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd(__mask8 k, void * s);
VMOVUPD void _mm512_storeu_pd( void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd( void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_loadu_pd( double * p);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d a, __mmask8 k, double * p);
VMOVUPD __m256d _mm256_maskz_loadu_pd(__mmask8 k, double * p);
VMOVUPD void _mm256_storeu_pd( double * p, __m256d a);
VMOVUPD void _mm256_mask_storeu_pd( double * p, __mmask8 k, __m256d a);

SIMD Floating-Point Exceptions
None.
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Note treatment of #AC varies; additionally:
#UD If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
MOVUPS—Move Unaligned Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 10 /r MOVUPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Move unaligned packed single precision floating-point from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>NP 0F 11 /r MOVUPS xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>SSE</td>
<td>Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed single precision floating-point from xmm2/mem to xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 11 /r VMOVUPS xmm2/m128, xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed single precision floating-point from xmm1 to xmm2/mem.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 10 /r VMOVUPS ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed single precision floating-point from ymm2/mem to ymm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 11 /r VMOVUPS ymm2/m256, ymm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Move unaligned packed single precision floating-point from ymm1 to ymm2/mem.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 10 /r VMOVUPS xmm1 (k1){z}, xmm2/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 10 /r VMOVUPS ymm1 (k1){z}, ymm2/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 10 /r VMOVUPS zmm1 (k1){z}, zmm2/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 (k1){z}, xmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed single precision floating-point values from xmm1 to xmm2/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 (k1){z}, ymm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Move unaligned packed single precision floating-point values from ymm1 to ymm2/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 (k1){z}, zmm1</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Move unaligned packed single precision floating-point values from zmm1 to zmm2/m512 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.
VEX.256 and EVEX.256 encoded versions:
Moves 256 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:
Moves 128 bits of packed single precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged. When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

**Operation**

**VMOVUPS (EVEX Encoded Versions, Register-Copy Form)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 TO KL - 1\)

\(i := j * 32\)

IF \(k1[j] OR *no writemask*\)

THEN \(DEST[i+31:i] := SRC[i+31:i]\)

ELSE

IF *merging-mask*; merging-mask

THEN *DEST[i+31:i] remains unchanged*;

ELSE \(DEST[i+31:i] := 0\); zeroing-mask

FI

FI;
ENDFOR

\(DEST[MAXVL-1:VL] := 0\)

**VMOVUPS (EVEX Encoded Versions, Store-Form)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 TO KL - 1\)

\(i := j * 32\)

IF \(k1[j] OR *no writemask*\)

THEN \(DEST[i+31:i] := SRC[i+31:i]\)

ELSE *DEST[i+31:i] remains unchanged*; merging-mask

FI;
ENDFOR;
VMOVUPS (EVEX Encoded Versions, Load-Form)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := SRC[i+31:i]
  ELSE
      IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] := 0 ; zeroing-masking
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VMOVUPS (VEX.256 Encoded Version, Load- and Register Copy)
DEST[255:0] := SRC[255:0]
DEST[MAXVL-1:256] := 0

VMOVUPS (VEX.256 Encoded Version, Store-Form)
DEST[255:0] := SRC[255:0]

VMOVUPS (VEX.128 Encoded Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] := 0

MOVUPS (128-bit Load- and Register-Copy- Form Legacy SSE Version)
DEST[127:0] := SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)

(V)MOVUPS (128-bit Store-Form Version)
DEST[127:0] := SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent
VMOVUPS __m512 _mm512_loadu_ps( void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps( __mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps( void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps( void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps( __m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps( void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm128_mask_loadu_ps( __m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm128_maskz_loadu_ps( __mmask8 k, void * s);
VMOVUPS void _mm128_mask_storeu_ps( void * d, __mmask8 k, __m128 a);
MOVUPS _m256 _mm256_loadu_ps ( float * p);
MOVUPS _mm256 _storeu_ps( float *p, __m256 a);
MOVUPS _m128 _mm_loadu_ps ( float *p);
MOVUPS void _mm_storeu_ps( float *p, __m128 a);

SIMD Floating-Point Exceptions
None.
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Note treatment of #AC varies.
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
Additionally:
#UD  If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.
**MOVZX—Move With Zero-Extend**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F B6 /r</td>
<td>MOVZX r16, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to word with zero-extension.</td>
</tr>
<tr>
<td>0F B6 /r</td>
<td>MOVZX r32, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Move byte to doubleword, zero-extension.</td>
</tr>
<tr>
<td>REX.W + 0F B6 /r</td>
<td>MOVZX r64, r/m8</td>
<td>RM</td>
<td>N.E.</td>
<td>Valid</td>
<td>Move byte to quadword, zero-extension.</td>
</tr>
<tr>
<td>0F B7 /r</td>
<td>MOVZX r32, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move word to doubleword, zero-extension.</td>
</tr>
<tr>
<td>REX.W + 0F B7 /r</td>
<td>MOVZX r64, r/m16</td>
<td>RM</td>
<td>N.E.</td>
<td>N.E.</td>
<td>Move word to quadword, zero-extension.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, r/m8 cannot be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST := ZeroExtend(SRC);

**Flags Affected**

None.

**Protected Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

- #GP(0): If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0): If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code): If a page fault occurs.
- #AC(0): If alignment checking is enabled and an unaligned memory reference is made.
- #UD: If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #SS(0): If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0): If the memory address is in a non-canonical form.
- #PF(fault-code): If a page fault occurs.
- #AC(0): If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD: If the LOCK prefix is used.
MPSADBW—Compute Multiple Packed Sums of Absolute Difference

**Description**

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand. Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

---

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 42 /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm1 and xmm2/m128 and writes the results in xmm1. Starting offsets within xmm1 and xmm2/m128 are determined by imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 42 /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and xmm3/m128 and writes the results in xmm1. Starting offsets within xmm2 and xmm3/m128 are determined by imm8.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.WIG 42 /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in xmm2 and ymm3/m256 and writes the results in xmm1. Starting offsets within ymm2 and ymm3/m128 are determined by imm8.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

---

MPSADBW—Compute Multiple Packed Sums of Absolute Difference

4-138 Vol. 2B
The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

**Operation**

**VMPSADBW (VEX.256 Encoded Version)**

- BLK2_OFFSET := imm8[1:0]*32
- BLK1_OFFSET := imm8[2]*32
- SRC1_BYTE0 := SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
- SRC1_BYTE1 := SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
- SRC1_BYTE2 := SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
- SRC1_BYTE3 := SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
- SRC1_BYTE4 := SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
- SRC1_BYTE5 := SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]

**Figure 4-5. 256-bit VMPSADBW Operation**
SRC1_BYTE6 := SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 := SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 := SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 := SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 := SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]
SRC2_BYTE0 := SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 := SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 := SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 := SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 := ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] := TEMP0 + TEMP1 + TEMP2 + TEMP3

BLK2_OFFSET := imm8[4:3]*32 + 128
BLK1_OFFSET := imm8[5]*32 + 128
SRC1_BYTE0 := SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 := SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 := SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 := SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 := SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 := SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 := SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 := SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 := SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 := SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 := SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 := SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 := SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 := SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 := SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 := ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[143:128] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[159:144] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[175:160] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[191:176] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[207:192] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP2 := ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP3 := ABS(SRC1_BYTE8 - SRC2_BYTE2)
DEST[223:208] := TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 := ABS(SRC1_BYTE6 - SRC2_BYTE6)
TEMP1 := ABS(SRC1_BYTE7 - SRC2_BYTE7)
TEMP2 := ABS(SRC1_BYTE8 - SRC2_BYTE8)
TEMP3 := ABS(SRC1_BYTE9 - SRC2_BYTE9)
DEST[239:224] := TEMP0 + TEMP1 + TEMP2 + TEMP3

VMPSADBW (VEX.128 Encoded Version)
BLK2_OFFSET := imm8[1:0]*32
BLK1_OFFSET := imm8[2]*32
SRC1_BYTE0 := SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 := SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 := SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 := SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 := SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 := SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 := SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 := SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 := SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 := SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 := SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 := SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 := SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 := SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 := SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]
TEMPO := ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 := ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 := ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 := ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] := TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO := ABS(SRC1 BYTE4 - SRC2 BYTE0)
TEMP1 := ABS(SRC1 BYTE5 - SRC2 BYTE1)
TEMP2 := ABS(SRC1 BYTE6 - SRC2 BYTE2)
TEMP3 := ABS(SRC1 BYTE7 - SRC2 BYTE3)
DEST[79:64] := TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO := ABS(SRC1 BYTE5 - SRC2 BYTE0)
TEMP1 := ABS(SRC1 BYTE6 - SRC2 BYTE1)
TEMP2 := ABS(SRC1 BYTE7 - SRC2 BYTE2)
TEMP3 := ABS(SRC1 BYTE8 - SRC2 BYTE3)
DEST[95:80] := TEMPO + TEMP1 + TEMP2 + TEMP3

TEMPO := ABS(SRC1 BYTE6 - SRC2 BYTE0)
TEMP1 := ABS(SRC1 BYTE7 - SRC2 BYTE1)
TEMP2 := ABS(SRC1 BYTE8 - SRC2 BYTE2)
TEMP3 := ABS(SRC1 BYTE9 - SRC2 BYTE3)
DEST[111:96] := TEMPO + TEMP1 + TEMP2 + TEMP3

DEST[127:112] := 0

MPSADBW (128-bit Legacy SSE Version)
SRC OFFSET := imm8[1:0]*32
DEST OFFSET := imm8[2]*32
DEST BYTE0 := DEST[DEST OFFSET+7:DEST OFFSET]
DEST BYTE1 := DEST[DEST OFFSET+15:DEST OFFSET+8]
DEST BYTE2 := DEST[DEST OFFSET+23:DEST OFFSET+16]
DEST BYTE3 := DEST[DEST OFFSET+31:DEST OFFSET+24]
DEST BYTE4 := DEST[DEST OFFSET+39:DEST OFFSET+32]
DEST BYTE5 := DEST[DEST OFFSET+47:DEST OFFSET+40]
DEST BYTE6 := DEST[DEST OFFSET+55:DEST OFFSET+48]
DEST BYTE7 := DEST[DEST OFFSET+63:DEST OFFSET+56]
DEST BYTE8 := DEST[DEST OFFSET+71:DEST OFFSET+64]
DEST BYTE9 := DEST[DEST OFFSET+79:DEST OFFSET+72]
DEST BYTE10 := DEST[DEST OFFSET+87:DEST OFFSET+80]

SRC BYTE0 := SRC[SRC OFFSET+7:SRC OFFSET]
SRC BYTE1 := SRC[SRC OFFSET+15:SRC OFFSET+8]
SRC BYTE2 := SRC[SRC OFFSET+23:SRC OFFSET+16]
SRC BYTE3 := SRC[SRC OFFSET+31:SRC OFFSET+24]

TEMPO := ABS(DEST BYTE0 - SRC BYTE0)
TEMP1 := ABS(DEST BYTE1 - SRC BYTE1)
TEMP2 := ABS(DEST BYTE2 - SRC BYTE2)
TEMP3 := \text{ABS} (\text{DEST\_BYTE3} - \text{SRC\_BYTE3})
DEST[15:0] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE1} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE2} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE3} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE4} - \text{SRC\_BYTE3})
DEST[31:16] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE2} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE3} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE4} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE5} - \text{SRC\_BYTE3})
DEST[47:32] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE3} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE4} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE5} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE6} - \text{SRC\_BYTE3})
DEST[63:48] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE4} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE5} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE6} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE7} - \text{SRC\_BYTE3})
DEST[79:64] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE5} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE6} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE7} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE8} - \text{SRC\_BYTE3})
DEST[95:80] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE6} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE7} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE8} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE9} - \text{SRC\_BYTE3})
DEST[111:96] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}

TEMP0 := \text{ABS} (\text{DEST\_BYTE7} - \text{SRC\_BYTE0})
TEMP1 := \text{ABS} (\text{DEST\_BYTE8} - \text{SRC\_BYTE1})
TEMP2 := \text{ABS} (\text{DEST\_BYTE9} - \text{SRC\_BYTE2})
TEMP3 := \text{ABS} (\text{DEST\_BYTE10} - \text{SRC\_BYTE3})
DEST[127:112] := \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}
DEST[\text{MAXVL}-1:128] (\text{Unmodified})

\text{Intel C/C++ Compiler Intrinsic Equivalent}

(V)\text{MPSADBW} \_\text{m128i} \_\text{mm}_{\text{mpsadbw}_\text{epu8}} (\_\text{m128i} \text{s1}, \_\text{m128i} \text{s2}, \text{const int} \text{mask})$

VM\text{PSADBW} \_\text{m256i} \_\text{mm}_{\text{mpsadbw}_\text{epu8}} (\_\text{m256i} \text{s1}, \_\text{m256i} \text{s2}, \text{const int} \text{mask})$

\text{Flags Affected}

None.
Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions.”
MUL—Unsigned Multiply

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /4</td>
<td>MUL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (AX := AL * r/m8).</td>
</tr>
<tr>
<td>REX + F6 /4</td>
<td>MUL r/m8'</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned multiply (AX := AL * r/m8).</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (DX:AX := AX * r/m16).</td>
</tr>
<tr>
<td>F7 /4</td>
<td>MUL r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Unsigned multiply (EDX:EAX := EAX * r/m32).</td>
</tr>
<tr>
<td>REX.W + F7 /4</td>
<td>MUL r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Unsigned multiply (RDX:RAX := RAX * r/m64).</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Table 4-9. MUL Results

<table>
<thead>
<tr>
<th>Operand Size</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>AL</td>
<td>r/m8</td>
<td>AX</td>
</tr>
<tr>
<td>Word</td>
<td>AX</td>
<td>r/m16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>Doubleword</td>
<td>EAX</td>
<td>r/m32</td>
<td>EDX:EAX</td>
</tr>
<tr>
<td>Quadword</td>
<td>RAX</td>
<td>r/m64</td>
<td>RDX:RAX</td>
</tr>
</tbody>
</table>

Operation

IF (Byte operation)

THEN

AX := AL * SRC;
ELSE (* Word or doubleword operation *)

IF OperandSize = 16

THEN

DX:AX := AX * SRC;
ELSE IF OperandSize = 32

THEN EDX:EAX := EAX * SRC; FI;
ELSE (* OperandSize = 64 *)

RDX:RAX := RAX * SRC;
Flags Affected
The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions
#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#UD  If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.
#UD  If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
MULPD—Multiply Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 59 /r MULPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply packed double precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F:W1G 59 /r VMULPD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed double precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F:W1G 59 /r VMULPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed double precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 59 /r VMULPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 59 /r VMULPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 59 /r VMULPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst(er)</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Multiply packed double precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
**Operation**

VMULPD (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
            DEST[i+63:i] := SRC1[i+63:i] * SRC2[63:0]
        ELSE
            DEST[i+63:i] := SRC1[i+63:i] * SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
          FI
      FI
  ENDfor

DEST[MAXVL-1:VL] := 0

VMULPD (VEX.256 Encoded Version)

DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[MAXVL-1:256] := 0;

VMULPD (VEX.128 Encoded Version)

DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[MAXVL-1:128] := 0

MULPD (128-bit Legacy SSE Version)

DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VMULPD __m512d _mm512_mul_pd( __m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd( __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd( __m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d _mm512_maskz_mul_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd ( __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
**MULPS—Multiply Packed Single Precision Floating-Point Values**

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 59 /r MULPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Multiply packed single precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F:WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F:WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed single precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.0F:W0 59 /r VMULPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.0F:W0 59 /r VMULPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.0F:W0 59 /r VMULPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst (er)</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>O perand 1</th>
<th>O perand 2</th>
<th>O perand 3</th>
<th>O perand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Multiply the packed single precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double precision floating-point results in the destination operand.

**EVEX encoded versions:** The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

**VEX.128 encoded version:** The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.
Operation

**VMULPS (EVEX Encoded Version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

DEST[i+31:i] := SRC1[i+31:i] * SRC2[31:0]

ELSE

DEST[i+31:i] := SRC1[i+31:i] * SRC2[i+31:i]

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI

ENDFOR

DEST[MAXVL-1:VL] := 0

**VMULPS (VEX.256 Encoded Version)**

DEST[31:0] := SRC1[31:0] * SRC2[31:0]


DEST[95:64] := SRC1[95:64] * SRC2[95:64]


DEST[MAXVL-1:256] := 0;

**VMULPS (VEX.128 Encoded Version)**

DEST[31:0] := SRC1[31:0] * SRC2[31:0]


DEST[95:64] := SRC1[95:64] * SRC2[95:64]


DEST[MAXVL-1:128] := 0

**MULPS (128-bit Legacy SSE Version)**

DEST[31:0] := SRC1[31:0] * SRC2[31:0]


DEST[95:64] := SRC1[95:64] * SRC2[95:64]


DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 __mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 __mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 __mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 __mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 __mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 __mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 __mm256_mul_ps(__m256 a, __m256 b);
VMULPS __m128 __mm128_mul_ps(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
MULSD—Multiply Scalar Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 59 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the low double precision floating-point value in xmm2/m64 by low double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 59 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 59 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply the low double precision floating-point value in xmm3/m64 by low double precision floating-point value in xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Multiplies the low double precision floating-point value in the second source operand by the low double precision floating-point value in the first source operand, and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSD (EVEX Encoded Version)

IF (EVEX.b = 1) AND SRC2 *is a register*
THEN
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
THEN DEST[63:0] := SRC1[63:0] * SRC2[63:0]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[63:0] remains unchanged*
ELSE ; zeroing-masking
THEN DEST[63:0] := 0
FI
Fi;
ENDFOR
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**VMULSD (VEX.128 Encoded Version)**
DEST[63:0] := SRC1[63:0] * SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**MULSD (128-bit Legacy SSE Version)**
DEST[63:0] := DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd( __m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd( __m128d a, __m128d b)

**SIMD Floating-Point Exceptions**
Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
MULSS—Multiply Scalar Single Precision Floating-Point Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 59 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Multiply the low single precision floating-point value in xmm2/m32 by the low single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 59 /r VMULSS xmm1 [k1][z], xmm2, xmm3/m32 {er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply the low single precision floating-point value in xmm3/m32 by the low single precision floating-point value in xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Multiplies the low single precision floating-point value from the second source operand by the low single precision floating-point value in the first source operand, and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the write-mask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

**VMULSS (EVEX Encoded Version)**

IF (EVEX.b = 1) AND SRC2 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET ROUNDING MODE FOR THIS INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] := SRC1[31:0] * SRC2[31:0]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] := 0

4-156 Vol. 2B MULSS—Multiply Scalar Single Precision Floating-Point Values
FI

ENDIF
DEST[MAXVL-1:128] := 0

VMULSS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0] * SRC2[31:0]
DEST[MAXVL-1:128] := 0

MULSS (128-bit Legacy SSE Version)
DEST[31:0] := DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss( __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss( __m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions
Underflow, Overflow, Invalid, Precision, Denormal.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
MULX—Unsigned Multiply Without Affecting Flags

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.F2.0F38.W0 F6 /r MULX r32a, r32b, r/m32</td>
<td>RVM</td>
<td>V/V</td>
<td>BMI2</td>
<td>Unsigned multiply of r/m32 with EDX without affecting arithmetic flags.</td>
</tr>
<tr>
<td>VEX.LZ.F2.0F38.W1 F6 /r MULX r64a, r64b, r/m64</td>
<td>RVM</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Unsigned multiply of r/m64 with RDX without affecting arithmetic flags.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>RDX/EDX is implied 64/32 bits source</td>
</tr>
</tbody>
</table>

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications. If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
   SRC1 := EDX;
   DEST2 := (SRC1*SRC2)[31:0];
   DEST1 := (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
   SRC1 := RDX;
   DEST2 := (SRC1*SRC2)[63:0];
   DEST1 := (SRC1*SRC2)[127:64];
FI

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.
unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);
unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.
Other Exceptions
See Table 2-29, "Type 13 Class Exception Conditions."
MWAIT—Monitor Wait

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 C9</td>
<td>MWAIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

**MWAIT for Address Range Monitoring**

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.
MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized "waiting" states and will select among those states based on the hint argument. Table 4-10 describes the meaning of ECX and EAX registers for MWAIT extensions.

Table 4-10. MWAIT Extension Register (ECX)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF=0). May be set only if CPUID.05H:ECX[bit 1] = 1.</td>
</tr>
<tr>
<td>31: 1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 4-11. MWAIT Hints Register (EAX)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 : 0</td>
<td>Sub C-state within a C-state, indicated by bits [7:4]</td>
</tr>
<tr>
<td>7 : 4</td>
<td>Target C-state*</td>
</tr>
<tr>
<td></td>
<td>Value of 0 means C1; 1 means C2 and so on</td>
</tr>
<tr>
<td></td>
<td>Value of 01111B means C0</td>
</tr>
<tr>
<td></td>
<td>Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states</td>
</tr>
<tr>
<td>31: 8</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 15, “Power and Thermal Management,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Operation

(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension MWAIT EAX, ECX *)

```
{  
    WHILE ( "Monitor Hardware is in armed state") {  
      implementation_dependent_optimized_state(EAX, ECX); }  
  
  Set the state of Monitor Hardware as triggered;  
}  
```

Intel C/C++ Compiler Intrinsic Equivalent

MWAIT void _mm_mwait(unsigned extensions, unsigned hints)
**Example**

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF (!trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF (!trigger_store_happened) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

**Numeric Exceptions**

None.

**Protected Mode Exceptions**

- **#GP(0)**  
  If ECX[31:1] ≠ 0.
  If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.

- **#UD**  
  If CPUID.01H:ECX.MONITOR[bit 3] = 0.
  If current privilege level is not 0.

**Real Address Mode Exceptions**

- **#GP**  
  If ECX[31:1] ≠ 0.
  If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.

- **#UD**  
  If CPUID.01H:ECX.MONITOR[bit 3] = 0.

**Virtual 8086 Mode Exceptions**

- **#UD**  
  The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#GP(0)**  
  If RCX[63:1] ≠ 0.
  If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.

- **#UD**  
  If the current privilege level is not 0.
  If CPUID.01H:ECX.MONITOR[bit 3] = 0.
NEG—Two’s Complement Negation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /3</td>
<td>NEG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Two’s complement negate r/m8.</td>
</tr>
<tr>
<td>REX + F6 /3</td>
<td>NEG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Two’s complement negate r/m8.</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Two’s complement negate r/m16.</td>
</tr>
<tr>
<td>F7 /3</td>
<td>NEG r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Two’s complement negate r/m32.</td>
</tr>
<tr>
<td>REX.W + F7 /3</td>
<td>NEG r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Two’s complement negate r/m64.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Replaces the value of operand (the destination operand) with its two’s complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF DEST = 0
    THEN CF := 0;
ELSE CF := 1;
FI;
DEST := [– (DEST)]

Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used but the destination is not a memory operand.
Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
NOP—No Operation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 90</td>
<td>NOP</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>One byte no-operation instruction.</td>
</tr>
<tr>
<td>NP 0F 1F /0</td>
<td>NOP r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Multi-byte no-operation instruction.</td>
</tr>
<tr>
<td>NP 0F 1F /0</td>
<td>NOP r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Multi-byte no-operation instruction.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>M</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of “no operation” as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

<table>
<thead>
<tr>
<th>Length</th>
<th>Assembly</th>
<th>Byte Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>66 NOP</td>
<td>66 90H</td>
</tr>
<tr>
<td>3 bytes</td>
<td>NOP DWORD ptr [EAX]</td>
<td>0F 1F 00H</td>
</tr>
<tr>
<td>4 bytes</td>
<td>NOP DWORD ptr [EAX + 00H]</td>
<td>0F 1F 40 00H</td>
</tr>
<tr>
<td>5 bytes</td>
<td>NOP DWORD ptr [EAX + EAX*1 + 00H]</td>
<td>0F 1F 44 00 00H</td>
</tr>
<tr>
<td>6 bytes</td>
<td>66 NOP DWORD ptr [EAX + EAX*1 + 00H]</td>
<td>66 0F 1F 44 00 00H</td>
</tr>
<tr>
<td>7 bytes</td>
<td>NOP DWORD ptr [EAX + 00000000H]</td>
<td>0F 1F 80 00 00 00 00H</td>
</tr>
<tr>
<td>8 bytes</td>
<td>NOP DWORD ptr [EAX + EAX*1 + 00000000H]</td>
<td>0F 1F 84 00 00 00 00 00H</td>
</tr>
<tr>
<td>9 bytes</td>
<td>66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]</td>
<td>66 0F 1F 84 00 00 00 00 00H</td>
</tr>
</tbody>
</table>

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
NOT—One’s Complement Negation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6 /2</td>
<td>NOT r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Reverse each bit of r/m8.</td>
</tr>
<tr>
<td>REX + F6 /2</td>
<td>NOT r/m8^1</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Reverse each bit of r/m8.</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Reverse each bit of r/m16.</td>
</tr>
<tr>
<td>F7 /2</td>
<td>NOT r/m32</td>
<td>M</td>
<td>Valid</td>
<td></td>
<td>Reverse each bit of r/m32.</td>
</tr>
<tr>
<td>REX.W + F7 /2</td>
<td>NOT r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Reverse each bit of r/m64.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 cannot be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
DEST := NOT DEST;

Flags Affected
None.

Protected Mode Exceptions
- #GP(0) If the destination operand points to a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
OR—Logical Inclusive OR

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C ib</td>
<td>OR AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AL OR imm8.</td>
</tr>
<tr>
<td>0D iw</td>
<td>OR AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>AX OR imm16</td>
<td></td>
</tr>
<tr>
<td>0D id</td>
<td>OR EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>EAX OR imm32</td>
<td></td>
</tr>
<tr>
<td>REX.W + 0D id</td>
<td>OR RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX OR imm32 (sign-extended).</td>
</tr>
<tr>
<td>80 /1 ib</td>
<td>OR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 OR imm8.</td>
</tr>
<tr>
<td>REX + 80 /1 ib</td>
<td>OR r/m81, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 OR imm8.</td>
</tr>
<tr>
<td>81 /1 iw</td>
<td>OR r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 OR imm16.</td>
</tr>
<tr>
<td>81 /1 id</td>
<td>OR r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 OR imm32.</td>
</tr>
<tr>
<td>REX.W + 81 /1 id</td>
<td>OR r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 OR imm32 (sign-extended).</td>
</tr>
<tr>
<td>83 /1 ib</td>
<td>OR r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 OR imm16 (sign-extended).</td>
</tr>
<tr>
<td>83 /1 ib</td>
<td>OR r/m32, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 OR imm16 (sign-extended).</td>
</tr>
<tr>
<td>REX.W + 83 /1 ib</td>
<td>OR r/m64, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 OR imm16 (sign-extended).</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 OR r8.</td>
</tr>
<tr>
<td>REX + 0B /r</td>
<td>OR r/m81, r8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 OR r8.</td>
</tr>
<tr>
<td>09 /r</td>
<td>OR r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 OR r16.</td>
</tr>
<tr>
<td>09 /r</td>
<td>OR r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 OR r32.</td>
</tr>
<tr>
<td>REX.W + 09 /r</td>
<td>OR r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 OR r64.</td>
</tr>
<tr>
<td>0A /r</td>
<td>OR r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r8 OR r/m8.</td>
</tr>
<tr>
<td>REX + 0A /r</td>
<td>OR r81, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r8 OR r/m8.</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 OR r/m16.</td>
</tr>
<tr>
<td>0B /r</td>
<td>OR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r32 OR r/m32.</td>
</tr>
<tr>
<td>REX.W + 0B /r</td>
<td>OR r/m64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r64 OR r/m64.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (r, w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.
In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST := DEST OR SRC;

**Flags Affected**

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

**Protected Mode Exceptions**

- **#GP(0)** If the destination operand points to a non-writable segment.
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
**ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 56/r ORPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical OR of packed double precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 56 /r VORPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 56 /r VORPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 56 /r VORPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical OR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical OR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
### Operation

**VORPD (EVEX Encoded Versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] := SRC1[i+63:i] BITWISE OR SRC2[63:0]
        ELSE
          DEST[i+63:i] := SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
  ENDFOR

DEST[MAXVL-1:VL] := 0

**VORPD (VEX.256 Encoded Version)**

DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAXVL-1:256] := 0

**VORPD (VEX.128 Encoded Version)**

DEST[63:0] := SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAXVL-1:128] := 0

**ORPD (128-bit Legacy SSE Version)**

DEST[63:0] := DEST[63:0] BITWISE OR SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

VORPD __m512d _mm512_or_pd ( __m512d a, __m512d b);
VORPD __m512d _mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d _mm512_maskz_or_pd ( __mmask8 k, __m512d a, __m512d b);
VORPD __m256d _mm256_or_pd ( __m256d s, __mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_mask_or_pd ( __mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_maskz_or_pd ( __mmask8 k, __m256d a, __m256d b);
VORPD __m128d _mm_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_mask_or_pd ( __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_maskz_or_pd ( __mmask8 k, __m128d a, __m128d b);
ORPD __m128d _mm_or_pd ( __m128d s, __m128d a, __m128d b);
SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 56 /r ORPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical OR of packed single precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 56 /r VORPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed single precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 56 /r VORPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical OR of packed single precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 56 /r VORPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical OR of packed single precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a bitwise logical OR of the four, eight or sixteen packed single precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

VORPS (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[31:0]
        ELSE
          DEST[i+31:i] := SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] := 0
      FI;
    ELSE
      FI
  ENDFOR
DEST[MAXVL-1:VL] := 0

VORPS (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[MAXVL-1:256] := 0

VORPS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[MAXVL-1:128] := 0

ORPS (128-bit Legacy SSE Version)
DEST[31:0] := SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 __mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 __mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 __mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 __mm256_mask_or_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VORPS __m256 __mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 __mm128_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 __mm128_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
ORPS __m128 __mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
OUT—Output to Port

**Description**

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

---

**NOTES:**

1. See the IA-32 Architecture Compatibility section below.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

**Opcode**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E6 ib</td>
<td>OUT imm8, AL</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Output byte in AL to I/O port address imm8.</td>
</tr>
<tr>
<td>E7 ib</td>
<td>OUT imm8, AX</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Output word in AX to I/O port address imm8.</td>
</tr>
<tr>
<td>EE</td>
<td>OUT DX, AL</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output byte in AL to I/O port address in DX.</td>
</tr>
<tr>
<td>EF</td>
<td>OUT DX, AX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output word in AX to I/O port address in DX.</td>
</tr>
<tr>
<td>EF</td>
<td>OUT DX, EAX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output doubleword in EAX to I/O port address in DX.</td>
</tr>
</tbody>
</table>
Operation

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
        ELSE (* I/O operation is allowed *)
            DEST := SRC; (* Writes to selected I/O port *)
    FI;
ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST := SRC; (* Writes to selected I/O port *)
FI;

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

Same as protected mode exceptions.
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). The DS segment may be overridden with a segment override prefix. The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPS /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a
description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is support using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

**IA-32 Architecture Compatibility**

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

**Operation**

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
   THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
      IF (Any I/O Permission Bit for I/O port being accessed = 1)
         THEN (* I/O operation is not allowed *)
            #GP(0);
         ELSE (* I/O operation is allowed *)
            DEST := SRC; (* Writes to I/O port *)
      FI;
   ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
      DEST := SRC; (* Writes to I/O port *)
   FI;

Byte transfer:
   IF 64-bit mode
      Then
      IF 64-Bit Address Size
         THEN
            IF DF = 0
               THEN RSI := RSI RSI + 1;
               ELSE RSI := RSI or – 1;
            FI;
         ELSE (* 32-Bit Address Size *)
            IF DF = 0
               THEN ESI := ESI + 1;
               ELSE ESI := ESI – 1;
            FI;
      ELSE
         IF DF = 0
            THEN (E)SI := (E)SI + 1;
            ELSE (E)SI := (E)SI – 1;
         FI;
   FI;

Word transfer:
   IF 64-bit mode
      Then
      IF 64-Bit Address Size
THEN
  IF DF = 0
    THEN RSI := RSI + 2;
    ELSE RSI := RSI or – 2;
    FI;
  ELSE (* 32-Bit Address Size *)
    IF DF = 0
      THEN ESI := ESI + 2;
      ELSE ESI := ESI – 2;
      FI;
  FI;
ELSE
  IF DF = 0
    THEN (E)SI := (E)SI + 2;
    ELSE (E)SI := (E)SI – 2;
    FI;
Doubleword transfer:
  IF 64-bit mode
    THEN
      IF 64-Bit Address Size
        THEN
          IF DF = 0
            THEN RSI := RSI + 4;
            ELSE RSI := RSI or – 4;
            FI;
          ELSE (* 32-Bit Address Size *)
            IF DF = 0
              THEN ESI := ESI + 4;
              ELSE ESI := ESI – 4;
              FI;
        FI;
      ELSE
        IF DF = 0
          THEN (E)SI := (E)SI + 4;
          ELSE (E)SI := (E)SI – 4;
          FI;
    FI;
Flags Affected
None.
Protected Mode Exceptions
#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (I/OPL) and any of the
  corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
  If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS
  segment.
  If the segment register contains a NULL segment selector.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
## PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 1C /r†</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.</td>
</tr>
<tr>
<td>PABSB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 1C /r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>PABSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 1D /r†</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.</td>
</tr>
<tr>
<td>PABSD mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 1D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>PABSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 1E /r†</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.</td>
</tr>
<tr>
<td>PABSQ mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 1E /r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>PABSD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 1C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>VPABSB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 1D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>VPABSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 1E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.</td>
</tr>
<tr>
<td>VPABSD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 1C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1.</td>
</tr>
<tr>
<td>VPABSB ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 1D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.</td>
</tr>
<tr>
<td>VPABSW ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 1E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compute the absolute value of 32-bit integers in ymm2/m256 and store UNSIGNED result in ymm1.</td>
</tr>
<tr>
<td>VPABSD ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 1C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPABSB xmm1 {k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 1D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPABSW xmm1 {k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 1C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPABSB zmm1 {k1}{z}, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 1D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPABSW zmm1 {k1}{z}, ymm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 1E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compute the absolute value of 32-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPABSW zmm1 {k1}{z}, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.128 encoded versions: The source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.128 encoded versions: The source operand is an XMM register or a 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or a 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

**PABSB With 128-bit Operands:**

Unsigned DEST[7:0] := ABS(SRC[7:0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] := ABS(SRC[127:120])

**VPABSB With 128-bit Operands:**

Unsigned DEST[7:0] := ABS(SRC[7:0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] := ABS(SRC[127:120])

**VPABSB With 256-bit Operands:**

Unsigned DEST[7:0] := ABS(SRC[7:0])
Repeat operation for 2nd through 31st bytes

**VPABSB (EVEX Encoded Versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[i+7:i] := ABS(SRC[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] := 0
      FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

**PABSW With 128-bit Operands:**

Unsigned DEST[15:0] := ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words

**VPABSW With 128-bit Operands:**

Unsigned DEST[15:0] := ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words

**VPABSW With 256-bit Operands:**

Unsigned DEST[15:0] := ABS(SRC[15:0])
Repeat operation for 2nd through 15th 16-bit words
VPABSW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
  THEN
    Unsigned DEST[i+15:i] := ABS(SRC[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

PABSD With 128-bit Operands:
Unsigned DEST[31:0] := ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] := ABS(SRC[127:96])

VPABSD With 128-bit Operands:
Unsigned DEST[31:0] := ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] := ABS(SRC[127:96])

VPABSD With 256-bit Operands:
Unsigned DEST[31:0] := ABS(SRC[31:0])
Repeat operation for 2nd through 7th 32-bit double words
Unsigned DEST[255:224] := ABS(SRC[255:224])

VPABSD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN
        Unsigned DEST[i+31:i] := ABS(SRC[31:0])
      ELSE
        Unsigned DEST[i+31:i] := ABS(SRC[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
VPABSQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN
      UnsIGNED DEST[i+63:i] := ABS(SRC[63:0])
    ELSE
      UnsIGNED DEST[i+63:i] := ABS(SRC[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPABSB__m512i _mm512_abs_epi8 ( __m512i a)
VPABSW__m512i _mm512_abs_epi16 ( __m512i a)
VPABSBS__m512i__mm512_abs_epi8 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i__mm512_abs_epi16 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i__mm512_maskz_abs_epi8 ( __mmask64 m, __m512i a)
VPABSW__m512i__mm512_maskz_abs_epi16 ( __mmask32 m, __m512i a)
VPABSB__m256i__mm256_mask_abs_epi8 ( __m256i s, __mmask64 m, __m256i a)
VPABSW__m256i__mm256_mask_abs_epi16 ( __m256i s, __mmask32 m, __m256i a)
VPABSB__m256i__mm256_maskz_abs_epi8 ( __mmask32 m, __m256i a)
VPABSW__m256i__mm256_maskz_abs_epi16 ( __mmask16 m, __m256i a)
VPABSB__m128i__mm128_mask_abs_epi8 ( __m128i s, __mmask64 m, __m128i a)
VPABSW__m128i__mm128_mask_abs_epi16 ( __m128i s, __mmask32 m, __m128i a)
VPABSB__m128i__mm128_maskz_abs_epi8 ( __mmask32 m, __m128i a)
VPABSW__m128i__mm128_maskz_abs_epi16 ( __mmask16 m, __m128i a)
VPABSB__m512i__mm512_mask_abs_epi32 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i__mm512_mask_abs_epi32 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i__mm512_maskz_abs_epi32 ( __mmask64 m, __m512i a)
VPABSW__m512i__mm512_maskz_abs_epi32 ( __mmask32 m, __m512i a)
VPABSB__m256i__mm256_mask_abs_epi32 ( __m256i s, __mmask64 m, __m256i a)
VPABSW__m256i__mm256_mask_abs_epi32 ( __m256i s, __mmask32 m, __m256i a)
VPABSB__m256i__mm256_maskz_abs_epi32 ( __mmask32 m, __m256i a)
VPABSW__m256i__mm256_maskz_abs_epi32 ( __mmask16 m, __m256i a)
VPABSB__m128i__mm128_mask_abs_epi32 ( __m128i s, __mmask64 m, __m128i a)
VPABSW__m128i__mm128_mask_abs_epi32 ( __m128i s, __mmask32 m, __m128i a)
VPABSB__m128i__mm128_maskz_abs_epi32 ( __mmask32 m, __m128i a)
VPABSW__m128i__mm128_maskz_abs_epi32 ( __mmask16 m, __m128i a)
VPABSB__m128i__mm128_mask_abs_epi64 ( __m128i s, __mmask64 m, __m128i a)
VPABSW__m128i__mm128_mask_abs_epi64 ( __m128i s, __mmask32 m, __m128i a)
VPABSB__m128i__mm128_maskz_abs_epi64 ( __mmask32 m, __m128i a)
VPABSW__m128i__mm128_maskz_abs_epi64 ( __mmask16 m, __m128i a)
VPABSB__m512i__mm512_mask_abs_epi64 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i__mm512_mask_abs_epi64 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i__mm512_maskz_abs_epi64 ( __mmask32 m, __m512i a)
VPABSW__m512i__mm512_maskz_abs_epi64 ( __mmask16 m, __m512i a)
VPABSB__m256i__mm256_mask_abs_epi64 ( __m256i s, __mmask64 m, __m256i a)
VPABSW__m256i__mm256_mask_abs_epi64 ( __m256i s, __mmask32 m, __m256i a)
VPABSB__m256i__mm256_maskz_abs_epi64 ( __mmask32 m, __m256i a)
VPABSW__m256i__mm256_maskz_abs_epi64 ( __mmask16 m, __m256i a)
VPABSB__m128i__mm128_mask_abs_epi64 ( __m128i s, __mmask64 m, __m128i a)
VPABSW__m128i__mm128_mask_abs_epi64 ( __m128i s, __mmask32 m, __m128i a)
VPABSB__m128i__mm128_maskz_abs_epi64 ( __mmask32 m, __m128i a)
VPABSW__m128i__mm128_maskz_abs_epi64 ( __mmask16 m, __m128i a)
VPABSB__m512i__mm512_mask_abs_epi64 ( __m512i s, __mmask64 m, __m512i a)
VPABSW__m512i__mm512_mask_abs_epi64 ( __m512i s, __mmask32 m, __m512i a)
VPABSB__m512i__mm512_maskz_abs_epi64 ( __mmask32 m, __m512i a)
VPABSW__m512i__mm512_maskz_abs_epi64 ( __mmask16 m, __m512i a)

PABSB/PABSW/PABSD/PABSQ—Packed Absolute Value
VPABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPABSD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
### PACKSSWB/PACKSSDW—Pack With Signed Saturation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 63 /r&lt;sup&gt;1&lt;/sup&gt; PACKSSWB mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Converts 4 packed signed word integers from mm1 and from mm2/m64 into 8 packed signed byte integers in mm1 using signed saturation.</td>
</tr>
<tr>
<td>66 0F 63 /r PACKSSWB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Converts 8 packed signed word integers from xmm1 and from xmm2/m128 into 16 packed signed byte integers in xmm1 using signed saturation.</td>
</tr>
<tr>
<td>NP 0F 6B /r&lt;sup&gt;1&lt;/sup&gt; PACKSSDW mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Converts 2 packed signed doubleword integers from mm1 and from mm2/m64 into 4 packed signed word integers in mm1 using signed saturation.</td>
</tr>
<tr>
<td>66 0F 6B /r PACKSSDW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Converts 4 packed signed doubleword integers from xmm1 and from xmm2/m128 into 8 packed signed word integers in xmm1 using signed saturation.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Converts 8 packed signed word integers from xmm2 and from xmm3/m128 into 16 packed signed byte integers in xmm1 using signed saturation.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 6B /r VPACKSSDW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Converts 4 packed signed doubleword integers from xmm2 and from xmm3/m128 into 8 packed signed word integers in xmm1 using signed saturation.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Converts 16 packed signed word integers from ymm2 and from ymm3/m256 into 32 packed signed byte integers in ymm1 using signed saturation.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 6B /r VPACKSSDW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Converts 8 packed signed doubleword integers from ymm2 and from ymm3/m256 into 16 packed signed word integers in ymm1 using signed saturation.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 63 /r VPACKSSWB xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts packed signed word integers from xmm2 and from xmm3/m128 into packed signed byte integers in xmm1 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG 63 /r VPACKSSWB ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts packed signed word integers from ymm2 and from ymm3/m256 into packed signed byte integers in ymm1 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG 63 /r VPACKSSWB zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts packed signed word integers from zmm2 and from zmm3/m512 into packed signed byte integers in zmm1 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 6B /r VPACKSSDW xmm1[k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts packed signed doubleword integers from xmm2 and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.</td>
</tr>
</tbody>
</table>
**Description**

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using saturation to handle overflow conditions beyond the range of signed byte integers. If the signed word value is beyond the range of a signed byte value (i.e., greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVAEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.
bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the write-mask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM destination register destination are unmodified.

Operation

PACKSSwB Instruction (128-bit Legacy SSE Version)

\[
\begin{align*}
\text{DEST}[7:0] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[15:0]); \\
\text{DEST}[15:8] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[31:16]); \\
\text{DEST}[23:16] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[47:32]); \\
\text{DEST}[31:24] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[63:48]); \\
\text{DEST}[39:32] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[79:64]); \\
\text{DEST}[47:40] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[95:80]); \\
\text{DEST}[55:48] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[111:96]); \\
\text{DEST}[63:56] & := \text{SaturateSignedWordToSignedByte} (\text{DEST}[127:112]); \\
\text{DEST}[71:64] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[15:0]); \\
\text{DEST}[79:72] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[31:16]); \\
\text{DEST}[87:80] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[47:32]); \\
\text{DEST}[95:88] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[63:48]); \\
\text{DEST}[103:96] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[79:64]); \\
\text{DEST}[111:104] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[95:80]); \\
\text{DEST}[119:112] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[111:96]); \\
\text{DEST}[127:120] & := \text{SaturateSignedWordToSignedByte} (\text{SRC}[127:112]); \\
\text{DEST}[\text{MAXVL-1:128}] & \text{(Unmodified)}
\end{align*}
\]

PACKSSdW Instruction (128-bit Legacy SSE Version)

\[
\begin{align*}
\text{DEST}[15:0] & := \text{SaturateSignedDwordToSignedWord} (\text{DEST}[31:0]); \\
\text{DEST}[31:16] & := \text{SaturateSignedDwordToSignedWord} (\text{DEST}[63:32]); \\
\text{DEST}[47:32] & := \text{SaturateSignedDwordToSignedWord} (\text{DEST}[95:64]); \\
\text{DEST}[63:48] & := \text{SaturateSignedDwordToSignedWord} (\text{DEST}[127:96]); \\
\text{DEST}[79:64] & := \text{SaturateSignedDwordToSignedWord} (\text{SRC}[31:0]); \\
\text{DEST}[95:80] & := \text{SaturateSignedDwordToSignedWord} (\text{SRC}[63:32]); \\
\text{DEST}[111:96] & := \text{SaturateSignedDwordToSignedWord} (\text{SRC}[95:64]); \\
\text{DEST}[127:112] & := \text{SaturateSignedDwordToSignedWord} (\text{SRC}[127:96]); \\
\text{DEST}[\text{MAXVL-1:128}] & \text{(Unmodified)}
\end{align*}
\]
VPACKSSWB Instruction (VEX.128 Encoded Version)
DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[MAXVL-1:128] := 0;

VPACKSSDW Instruction (VEX.128 Encoded Version)
DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[MAXVL-1:128] := 0;

VPACKSSWB Instruction (VEX.256 Encoded Version)
DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] := SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[207:200] := SaturateSignedWordToSignedByte (SRC2[159:144]);
DEST[MAXVL-1:256] := 0;

**VPACKSSDW Instruction (VEX.256 Encoded Version)**

DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[79:64] := SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] := SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] := SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[143:128] := SaturateSignedDwordToSignedWord (SRC1[159:128]);
DEST[159:144] := SaturateSignedDwordToSignedWord (SRC1[191:160]);
DEST[MAXVL-1:256] := 0;

**VPACKSSWB (EVEX Encoded Versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[47:40] := SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[71:64] := SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[103:96] := SaturateSignedWordToSignedByte (SRC2[79:64]);
IF VL >= 256
  TMP_DEST[143:136] := SaturateSignedWordToSignedByte (SRC1[159:144]);
VPACKSSWB/PACKSSDW—Pack With Signed Saturation

INSTRUCTION SET REFERENCE, M-U

Vol. 2B 4-193

FI;
IF VL >= 512


TMP_DEST[503:496] := SaturateSignedWordToSignedByte (SRC2[495:480]);
TMP_DEST[511:504] := SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
  THEN
    DEST[i+j] := TMP_DEST[i+j]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+7:i] remains unchanged*
  ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+7:i] := 0
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

VPACKSSDW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO (KL/2 - 1)
i := j * 32

IF (EVEX.b == 1) AND (SRC2 *is memory*)
  THEN
    TMP_SRC2[i+31:i] := SRC2[31:0]
  ELSE
    TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;

TMP_DEST[15:0] := SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] := SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] := SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[79:64] := SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[111:96] := SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
IF VL >= 256
  FI;
IF VL >= 512
  TMP_DEST[303:288] := SaturateSignedDwordToSignedWord (SRC1[351:320]);
  TMP_DEST[335:320] := SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
  TMP_DEST[415:400] := SaturateSignedDwordToSignedWord (SRC1[447:416]);
  TMP_DEST[431:416] := SaturateSignedDwordToSignedWord (SRC1[479:448]);
TMP\_DEST[447:432] := SaturateSignedDwordToSignedWord (SRC1[511:480]);
TMP\_DEST[495:480] := SaturateSignedDwordToSignedWord (TMP\_SRC2[479:448]);
TMP\_DEST[511:496] := SaturateSignedDwordToSignedWord (TMP\_SRC2[511:480]);

FI;

FOR j := 0 TO KL-1
i := j \* 16
IF k1[j] OR *no writemask*
THEN DEST[i+15:i] := TMP\_DEST[i+15:i]
ELSE
IF *merging-masking*
THEN *DEST[i+15:i] remains unchanged*
ELSE *zeroing-masking*
DEST[i+15:i] := 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPACKSSDW\_m512i _mm512\_packs\_epi32(__m512i m1, __m512i m2);
VPACKSSDW\_m512i _mm512\_mask\_packs\_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW\_m256i _mm256\_mask\_packs\_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW\_m128i _mm128\_mask\_packs\_epi32(__mmask8 k, __m128i m1, __m128i m2);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPACKSSDW, see Table 2-50, “Type E4NF Class Exception Conditions.”
EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
PACKUSDW—Pack With Unsigned Saturation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Convert 4 packed signed doubleword integers from xmm1 and 4 packed signed doubleword integers from xmm2/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.</td>
</tr>
<tr>
<td>VEX.128.66.0F38 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Convert 4 packed signed doubleword integers from xmm2 and 4 packed signed doubleword integers from xmm3/m128 into 8 packed unsigned word integers in xmm1 using unsigned saturation.</td>
</tr>
<tr>
<td>VEX.256.66.0F38 2B /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Convert 8 packed signed doubleword integers from ymm2 and 8 packed signed doubleword integers from ymm3/m256 into 16 packed unsigned word integers in ymm1 using unsigned saturation.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 2B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 2B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 2B /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Convert packed signed doubleword integers from zmm2 and packed signed doubleword integers from zmm3/m512/m32bcst into packed unsigned word integers in zmm1 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.
128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding destination register destination are unmodified.

**Operation**

**PACKUSDW (Legacy SSE Instruction)**

\[
\begin{align*}
\text{TMP}[15:0] & := (\text{DEST}[31:0] < 0) ? 0 : \text{DEST}[15:0]; \\
\text{DEST}[15:0] & := (\text{DEST}[31:0] > FFFFH) ? FFFFH : \text{TMP}[15:0]; \\
\text{TMP}[31:16] & := (\text{DEST}[63:32] < 0) ? 0 : \text{DEST}[47:32]; \\
\text{TMP}[47:32] & := (\text{DEST}[95:64] < 0) ? 0 : \text{DEST}[79:64]; \\
\text{DEST}[47:32] & := (\text{DEST}[95:64] > FFFFH) ? FFFFH : \text{TMP}[47:32]; \\
\text{TMP}[63:48] & := (\text{DEST}[127:96] < 0) ? 0 : \text{DEST}[111:96]; \\
\text{TMP}[79:64] & := (\text{SRC}[31:0] < 0) ? 0 : \text{SRC}[15:0]; \\
\text{DEST}[79:64] & := (\text{SRC}[31:0] > FFFFH) ? FFFFH : \text{TMP}[79:64]; \\
\text{TMP}[95:80] & := (\text{SRC}[63:32] < 0) ? 0 : \text{SRC}[47:32]; \\
\text{DEST}[95:80] & := (\text{SRC}[63:32] > FFFFH) ? FFFFH : \text{TMP}[95:80]; \\
\text{TMP}[111:96] & := (\text{SRC}[95:64] < 0) ? 0 : \text{SRC}[79:64]; \\
\text{DEST}[111:96] & := (\text{SRC}[95:64] > FFFFH) ? FFFFH : \text{TMP}[111:96]; \\
\text{TMP}[127:112] & := (\text{SRC}[127:96] < 0) ? 0 : \text{SRC}[111:96]; \\
\text{DEST}[\text{MAXVL}-1:128] & := 0;
\end{align*}
\]

**PACKUSDW (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{TMP}[15:0] & := (\text{SRC1}[31:0] < 0) ? 0 : \text{SRC1}[15:0]; \\
\text{DEST}[15:0] & := (\text{SRC1}[31:0] > FFFFH) ? FFFFH : \text{TMP}[15:0]; \\
\text{TMP}[31:16] & := (\text{SRC1}[63:32] < 0) ? 0 : \text{SRC1}[47:32]; \\
\text{DEST}[31:16] & := (\text{SRC1}[63:32] > FFFFH) ? FFFFH : \text{TMP}[31:16]; \\
\text{TMP}[47:32] & := (\text{SRC1}[95:64] < 0) ? 0 : \text{SRC1}[79:64]; \\
\text{DEST}[47:32] & := (\text{SRC1}[95:64] > FFFFH) ? FFFFH : \text{TMP}[47:32]; \\
\text{TMP}[63:48] & := (\text{SRC1}[127:96] < 0) ? 0 : \text{SRC1}[111:96]; \\
\text{TMP}[79:64] & := (\text{SRC2}[31:0] < 0) ? 0 : \text{SRC2}[15:0]; \\
\text{DEST}[79:64] & := (\text{SRC2}[31:0] > FFFFH) ? FFFFH : \text{TMP}[79:64]; \\
\text{TMP}[95:80] & := (\text{SRC2}[63:32] < 0) ? 0 : \text{SRC2}[47:32]; \\
\text{DEST}[95:80] & := (\text{SRC2}[63:32] > FFFFH) ? FFFFH : \text{TMP}[95:80]; \\
\text{TMP}[111:96] & := (\text{SRC2}[95:64] < 0) ? 0 : \text{SRC2}[79:64]; \\
\text{DEST}[111:96] & := (\text{SRC2}[95:64] > FFFFH) ? FFFFH : \text{TMP}[111:96]; \\
\text{TMP}[127:112] & := (\text{SRC2}[127:96] < 0) ? 0 : \text{SRC2}[111:96]; \\
\text{DEST}[\text{MAXVL}-1:128] & := 0;
\end{align*}
\]

**VPACKUSDW (VEX.256 Encoded Version)**

\[
\begin{align*}
\text{TMP}[15:0] & := (\text{SRC1}[31:0] < 0) ? 0 : \text{SRC1}[15:0]; \\
\text{DEST}[15:0] & := (\text{SRC1}[31:0] > FFFFH) ? FFFFH : \text{TMP}[15:0]; \\
\text{TMP}[31:16] & := (\text{SRC1}[63:32] < 0) ? 0 : \text{SRC1}[47:32]; \\
\text{DEST}[31:16] & := (\text{SRC1}[63:32] > FFFFH) ? FFFFH : \text{TMP}[31:16]; \\
\text{TMP}[47:32] & := (\text{SRC1}[95:64] < 0) ? 0 : \text{SRC1}[79:64]; \\
\text{DEST}[47:32] & := (\text{SRC1}[95:64] > FFFFH) ? FFFFH : \text{TMP}[47:32]; \\
\text{TMP}[63:48] & := (\text{SRC1}[127:96] < 0) ? 0 : \text{SRC1}[111:96]; \\
\text{TMP}[79:64] & := (\text{SRC2}[31:0] < 0) ? 0 : \text{SRC2}[15:0]; \\
\text{DEST}[79:64] & := (\text{SRC2}[31:0] > FFFFH) ? FFFFH : \text{TMP}[79:64]; \\
\text{TMP}[95:80] & := (\text{SRC2}[63:32] < 0) ? 0 : \text{SRC2}[47:32]; \\
\text{DEST}[95:80] & := (\text{SRC2}[63:32] > FFFFH) ? FFFFH : \text{TMP}[95:80]; \\
\text{TMP}[111:96] & := (\text{SRC2}[95:64] < 0) ? 0 : \text{SRC2}[79:64]; \\
\text{DEST}[111:96] & := (\text{SRC2}[95:64] > FFFFH) ? FFFFH : \text{TMP}[111:96]; \\
\text{TMP}[127:112] & := (\text{SRC2}[127:96] < 0) ? 0 : \text{SRC2}[111:96]; \\
\text{DEST}[\text{MAXVL}-1:128] & := 0;
\end{align*}
\]
VPACKUSDw (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO ((KL/2) - 1)
    i := j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] := SRC2[i+31:i]
        FI;
    ENDFOR;

    TMP[15:0] := (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
    DEST[15:0] := (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
    TMP[47:32] := (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
    TMP[63:48] := (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
    TMP[79:64] := (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
    DEST[79:64] := (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
    TMP[111:96] := (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
    IF VL >= 256
FI;

IF VL >= 512
TMP[287:272] := (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
TMP[303:288] := (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
TMP[431:416] := (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
TMP[447:432] := (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
TMP[491:476] := (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
TMP[511:492] := (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];

FI;
FOR j := 0 TO KL-1
i := j * 16
IF k1[j] OR *no writemask*

THEN
  DEST[i+15:i] := TMP_DEST[i+15:i]
ELSE
  IF *merging-masking*
      THEN *DEST[i+15:i] remains unchanged*
  ELSE *zeroing-masking*
      DEST[i+15:i] := 0
  FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32( __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i _mm256_mask_packus_epi32( __m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32( __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i _mm_mask_packus_epi32( __m128i s, __mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32( __m128i m1, __m128i m2);
VPACKUSDW__m256i _mm256_packus_epi32( __m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
PACKUSWB—Pack With Unsigned Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 67 /r</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Converts 4 signed word integers from mm and 4 signed word integers from mm/m64 into 8 unsigned byte integers in mm using unsigned saturation.</td>
</tr>
<tr>
<td>PACKUSWB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 67 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Converts 8 signed word integers from xmm1 and 8 signed word integers from xmm2/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.</td>
</tr>
<tr>
<td>PACKUSWB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G 67 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Converts 8 signed word integers from xmm2 and 8 signed word integers from xmm3/m128 into 16 unsigned byte integers in xmm1 using unsigned saturation.</td>
</tr>
<tr>
<td>VPACKUSWB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G 67 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Converts 16 signed word integers from ymm2 and 16 signed word integers from ymm3/m256 into 32 unsigned byte integers in ymm1 using unsigned saturation.</td>
</tr>
<tr>
<td>VPACKUSWB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1G 67 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts signed word integers from xmm2 and signed word integers from xmm3/m128 into unsigned byte integers in xmm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKUSWB xmm1{k1}[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1G 67 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts signed word integers from ymm2 and signed word integers from ymm3/m256 into unsigned byte integers in ymm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKUSWB ymm1{k1}[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1G 67 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts signed word integers from zmm2 and signed word integers from zmm3/m512 into unsigned byte integers in zmm1 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPACKUSWB zmm1{k1}[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts 4, 8, 16, or 32 signed word integers from the destination operand (first operand) and 4, 8, 16, or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.
VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation**

PACKUSwB (with 64-bit Operands)

\[
\begin{align*}
\text{DEST}[7:0] &:= \text{SaturateSignedWordToUnsignedByte} \text{DEST}[15:0]; \\
\text{DEST}[15:8] &:= \text{SaturateSignedWordToUnsignedByte} \text{DEST}[31:16]; \\
\text{DEST}[23:16] &:= \text{SaturateSignedWordToUnsignedByte} \text{DEST}[47:32]; \\
\text{DEST}[31:24] &:= \text{SaturateSignedWordToUnsignedByte} \text{DEST}[63:48]; \\
\text{DEST}[39:32] &:= \text{SaturateSignedWordToUnsignedByte} \text{SRC}[15:0]; \\
\text{DEST}[47:40] &:= \text{SaturateSignedWordToUnsignedByte} \text{SRC}[31:16]; \\
\text{DEST}[55:48] &:= \text{SaturateSignedWordToUnsignedByte} \text{SRC}[47:32]; \\
\text{DEST}[63:56] &:= \text{SaturateSignedWordToUnsignedByte} \text{SRC}[63:48]; \\
\end{align*}
\]

PACKUSwB (Legacy SSE Instruction)

\[
\begin{align*}
\text{DEST}[7:0] &:= \text{SaturateSignedWordToUnsignedByte} (\text{DEST}[15:0]); \\
\text{DEST}[15:8] &:= \text{SaturateSignedWordToUnsignedByte} (\text{DEST}[31:16]); \\
\text{DEST}[23:16] &:= \text{SaturateSignedWordToUnsignedByte} (\text{DEST}[47:32]); \\
\text{DEST}[31:24] &:= \text{SaturateSignedWordToUnsignedByte} (\text{DEST}[63:48]); \\
\text{DEST}[39:32] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[15:0]); \\
\text{DEST}[47:40] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[31:16]); \\
\text{DEST}[55:48] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[47:32]); \\
\text{DEST}[63:56] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[63:48]); \\
\end{align*}
\]

PACKUSwB (VEX.128 Encoded Version)

\[
\begin{align*}
\text{DEST}[7:0] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[15:0]); \\
\text{DEST}[15:8] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[31:16]); \\
\text{DEST}[23:16] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[47:32]); \\
\text{DEST}[31:24] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[63:48]); \\
\text{DEST}[39:32] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[79:64]); \\
\text{DEST}[47:40] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[95:80]); \\
\text{DEST}[55:48] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[111:96]); \\
\text{DEST}[63:56] &:= \text{SaturateSignedWordToUnsignedByte} (\text{SRC}[127:112]); \\
\end{align*}
\]
DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] := 0;

VPACKUSWB (VEX.256 Encoded Version)
DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] := SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] := SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[287:272]);
DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[303:288]);
DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[319:304]);
DEST[167:160] := SaturateSignedWordToUnsignedByte (SRC1[335:320]);
DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[351:336]);
DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[367:352]);
DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[383:368]);
DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC1[409:394]);
DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC1[425:410]);
DEST[215:208] := SaturateSignedWordToUnsignedByte (SRC1[441:426]);
DEST[223:216] := SaturateSignedWordToUnsignedByte (SRC1[457:442]);
DEST[239:232] := SaturateSignedWordToUnsignedByte (SRC1[489:474]);
DEST[247:240] := SaturateSignedWordToUnsignedByte (SRC1[505:490]);
DEST[255:248] := SaturateSignedWordToUnsignedByte (SRC1[521:506]);

VPACKUSWB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] := SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] := SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[39:32] := SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] := SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] := SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] := SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] := SaturateSignedWordToUnsignedByte (SRC1[143:128]);
TMP_DEST[79:72] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
TMP_DEST[87:80] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
TMP_DEST[95:88] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
TMP_DEST[103:96] := SaturateSignedWordToUnsignedByte (SRC1[207:192]);
TMP_DEST[111:104] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
TMP_DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
TMP_DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC1[255:240]);
TMP_DEST[139:132] := SaturateSignedWordToUnsignedByte (SRC1[287:272]);
TMP_DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[303:288]);
TMP_DEST[147:140] := SaturateSignedWordToUnsignedByte (SRC1[319:304]);
TMP_DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[335:320]);
TMP_DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[367:352]);
TMP_DEST[163:156] := SaturateSignedWordToUnsignedByte (SRC1[383:368]);
TMP_DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[441:426]);
TMP_DEST[179:172] := SaturateSignedWordToUnsignedByte (SRC1[457:442]);
TMP_DEST[187:180] := SaturateSignedWordToUnsignedByte (SRC1[489:474]);
TMP_DEST[191:184] := SaturateSignedWordToUnsignedByte (SRC1[505:490]);
TMP_DEST[119:112] := SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] := SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
  TMP_DEST[143:136] := SaturateSignedWordToUnsignedByte (SRC1[159:144]);
  TMP_DEST[151:144] := SaturateSignedWordToUnsignedByte (SRC1[175:160]);
  TMP_DEST[159:152] := SaturateSignedWordToUnsignedByte (SRC1[191:176]);
  TMP_DEST[175:168] := SaturateSignedWordToUnsignedByte (SRC1[223:208]);
  TMP_DEST[183:176] := SaturateSignedWordToUnsignedByte (SRC1[239:224]);
  TMP_DEST[199:192] := SaturateSignedWordToUnsignedByte (SRC2[143:128]);
  TMP_DEST[207:200] := SaturateSignedWordToUnsignedByte (SRC2[159:144]);
  TMP_DEST[231:224] := SaturateSignedWordToUnsignedByte (SRC2[207:192]);
FI;
IF VL >= 512
  TMP_DEST[263:256] := SaturateSignedWordToUnsignedByte (SRC1[271:256]);
  TMP_DEST[279:272] := SaturateSignedWordToUnsignedByte (SRC1[303:288]);
  TMP_DEST[287:280] := SaturateSignedWordToUnsignedByte (SRC1[319:304]);
  TMP_DEST[295:288] := SaturateSignedWordToUnsignedByte (SRC1[335:320]);
  TMP_DEST[303:296] := SaturateSignedWordToUnsignedByte (SRC1[351:336]);
  TMP_DEST[311:304] := SaturateSignedWordToUnsignedByte (SRC1[367:352]);
  TMP_DEST[319:312] := SaturateSignedWordToUnsignedByte (SRC1[383:368]);
  TMP_DEST[327:320] := SaturateSignedWordToUnsignedByte (SRC2[271:256]);
  TMP_DEST[335:328] := SaturateSignedWordToUnsignedByte (SRC2[287:272]);
  TMP_DEST[343:336] := SaturateSignedWordToUnsignedByte (SRC2[303:288]);
  TMP_DEST[351:344] := SaturateSignedWordToUnsignedByte (SRC2[319:304]);
  TMP_DEST[359:352] := SaturateSignedWordToUnsignedByte (SRC2[335:320]);
  TMP_DEST[375:368] := SaturateSignedWordToUnsignedByte (SRC2[367:352]);
  TMP_DEST[399:392] := SaturateSignedWordToUnsignedByte (SRC1[415:400]);
  TMP_DEST[407:400] := SaturateSignedWordToUnsignedByte (SRC1[431:416]);
  TMP_DEST[415:408] := SaturateSignedWordToUnsignedByte (SRC1[447:432]);
  TMP_DEST[423:416] := SaturateSignedWordToUnsignedByte (SRC1[463:448]);
  TMP_DEST[439:432] := SaturateSignedWordToUnsignedByte (SRC1[495:480]);
  TMP_DEST[447:440] := SaturateSignedWordToUnsignedByte (SRC1[511:496]);
  TMP_DEST[455:448] := SaturateSignedWordToUnsignedByte (SRC2[399:384]);
  TMP_DEST[463:456] := SaturateSignedWordToUnsignedByte (SRC2[415:400]);
  TMP_DEST[479:472] := SaturateSignedWordToUnsignedByte (SRC2[447:432]);
  TMP_DEST[487:480] := SaturateSignedWordToUnsignedByte (SRC2[463:448]);
  TMP_DEST[495:488] := SaturateSignedWordToUnsignedByte (SRC2[479:464]);
TMP_DEST[503:496] := SaturateSignedWordToUnsignedByte (SRC2[495:480]);
TMP_DEST[511:504] := SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j := 0 TO KL-1
i := j * 8
IF k1[j] OR *no writemask*
    THEN
        DEST[i+7:i] := TMP_DEST[i+7:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] := 0
        FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSWB __m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);
VPACKUSWB __m256i _mm256_mask_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m128i _mm128_packus_epi16(__m128i m1, __m128i m2);
PACKUSWB __m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
PACKUSWB __m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);
(V)PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
### PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F FC/r1</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Add packed byte integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>PADDB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F FD/r1</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Add packed word integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>PADDW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F FE/r1</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Add packed doubleword integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>PADDD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F D4/r1</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Add packed quadword integers from mm/m64 and mm.</td>
</tr>
<tr>
<td>PADDQ mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FC/r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Add packed byte integers from xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>PADD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FD/r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Add packed word integers from xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>PADDW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FE/r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Add packed doubleword integers from xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>PADDD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D4/r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Add packed quadword integers from xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>PADDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG FC/r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1.</td>
</tr>
<tr>
<td>VPADDB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG FD/r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Add packed word integers from xmm2, xmm3/m128 and store in xmm1.</td>
</tr>
<tr>
<td>VPADDW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG FE/r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Add packed doubleword integers from xmm2, xmm3/m128 and store in xmm1.</td>
</tr>
<tr>
<td>VPADDD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D4/r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Add packed quadword integers from xmm2, xmm3/m128 and store in xmm1.</td>
</tr>
<tr>
<td>VPADDQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG FC/r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1.</td>
</tr>
<tr>
<td>VPADDB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG FD/r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Add packed word integers from ymm2, ymm3/m256 and store in ymm1.</td>
</tr>
<tr>
<td>VPADDW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG FE/r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Add packed doubleword integers from ymm2, ymm3/m256 and store in ymm1.</td>
</tr>
<tr>
<td>VPADDQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG FC/r</td>
<td>C V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td>Add packed byte integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPADDQ xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG FD/r</td>
<td>C V/V</td>
<td>AVX512VL AVX512Bw</td>
<td></td>
<td>Add packed word integers from xmm2, and xmm3/m128 and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPADDQ xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 FE/r</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td></td>
<td>Add packed doubleword integers from xmm2, and xmm3/m128/m32bcst and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPADDQ xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 D4/r</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td></td>
<td>Add packed quadword integers from xmm2, and xmm3/m128/m64bcst and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPADDQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode/Instruction</td>
<td>Op / En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------</td>
<td>------------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG FC /r VPADDB ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed byte integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG FD /r VPADDW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed word integers from ymm2, and ymm3/m256 and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 FE /r VPADD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed doubleword integers from ymm2, ymm3/m256/m32bcst and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 D4 /r VPADDQ ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Add packed quadword integers from ymm2, ymm3/m256/m64bcst and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG FC /r VPADDB zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Add packed byte integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG FD /r VPADDW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Add packed word integers from zmm2, and zmm3/m512 and store in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 FE /r VPADD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>D V/V</td>
<td>AVX512F</td>
<td>Add packed doubleword integers from zmm2, zmm3/m512/m32bcst and store in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 D4 /r VPADDQ zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>D V/V</td>
<td>AVX512F</td>
<td>Add packed quadword integers from zmm2, zmm3/m512/m64bcst and store in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to
be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two’s complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDW/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

**PADDB (with 64-bit Operands)**

\[
\text{DEST}[7:0] := \text{DEST}[7:0] + \text{SRC}[7:0]; \\
\text{DEST}[63:56] := \text{DEST}[63:56] + \text{SRC}[63:56];
\]

(* Repeat add operation for 2nd through 7th byte *)

**PADDW (with 64-bit Operands)**

\[
\text{DEST}[15:0] := \text{DEST}[15:0] + \text{SRC}[15:0]; \\
\text{DEST}[63:48] := \text{DEST}[63:48] + \text{SRC}[63:48];
\]

(* Repeat add operation for 2nd and 3rd word *)

**PADDD (with 64-bit Operands)**

\[
\text{DEST}[31:0] := \text{DEST}[31:0] + \text{SRC}[31:0]; \\
\]

**PADDQ (with 64-Bit Operands)**

\[
\text{DEST}[63:0] := \text{DEST}[63:0] + \text{SRC}[63:0];
\]
**PADDB (Legacy SSE Instruction)**

DEST[7:0] := DEST[7:0] + SRC[7:0];

(* Repeat add operation for 2nd through 15th byte *)


DEST[MAXVL-1:128] (Unmodified)

**PADDW (Legacy SSE Instruction)**

DEST[15:0] := DEST[15:0] + SRC[15:0];

(* Repeat add operation for 2nd through 7th word *)


DEST[MAXVL-1:128] (Unmodified)

**PADDD (Legacy SSE Instruction)**

DEST[31:0] := DEST[31:0] + SRC[31:0];

(* Repeat add operation for 2nd and 3rd doubleword *)


DEST[MAXVL-1:128] (Unmodified)

**PADDQ (Legacy SSE Instruction)**

DEST[63:0] := DEST[63:0] + SRC[63:0];

DEST[127:64] := DEST[127:64] + SRC[127:64];

DEST[MAXVL-1:128] := 0;

**VPADDB (VEX.128 Encoded Instruction)**

DEST[7:0] := SRC1[7:0] + SRC2[7:0];

(* Repeat add operation for 2nd through 15th byte *)


DEST[MAXVL-1:128] := 0;

**VPADDW (VEX.128 Encoded Instruction)**

DEST[15:0] := SRC1[15:0] + SRC2[15:0];

(* Repeat add operation for 2nd through 7th word *)


DEST[MAXVL-1:128] := 0;

**VPADDD (VEX.128 Encoded Instruction)**

DEST[31:0] := SRC1[31:0] + SRC2[31:0];

(* Repeat add operation for 2nd and 3rd doubleword *)


DEST[MAXVL-1:128] := 0;

**VPADDQ (VEX.128 Encoded Instruction)**

DEST[63:0] := SRC1[63:0] + SRC2[63:0];

DEST[127:64] := SRC1[127:64] + SRC2[127:64];

DEST[MAXVL-1:128] := 0;

**VPADDB (VEX.256 Encoded Instruction)**

DEST[7:0] := SRC1[7:0] + SRC2[7:0];

(* Repeat add operation for 2nd through 31th byte *)

VPADDw (VEX.256 Encoded Instruction)
DEST[15:0] := SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)

VPADD (VEX.256 Encoded Instruction)
DEST[31:0] := SRC1[31:0] + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)

VPADDD (VEX.256 Encoded Instruction)
DEST[63:0] := SRC1[63:0] + SRC2[63:0];
DEST[127:64] := SRC1[127:64] + SRC2[127:64];

VPADDB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SRC1[i+7:i] + SRC2[i+7:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] = 0
      FI
  ELSE
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

VPADDD (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC1[i+15:i] + SRC2[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] = 0
      FI
  ELSE
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
VPADDD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := SRC1[i+31:i] + SRC2[31:0]
      ELSE DEST[i+31:i] := SRC1[i+31:i] + SRC2[i+31:i]
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

VPADDQ (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := SRC1[i+63:i] + SRC2[63:0]
      ELSE DEST[i+63:i] := SRC1[i+63:i] + SRC2[i+63:i]
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents
VPADDB__m512i _mm512_add_epi8 ( __m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 ( __m512i a, __m512i b)
VPADDD__m512i _mm512_mask_add_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDD__m512i _mm512_mask_add_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDQ__m512i _mm512_maskz_add_epi8 ( __m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDQ__m512i _mm512_maskz_add_epi16 ( __m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 ( __m256i s, __mmask128 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_mask_add_epi16 ( __m256i s, __mmask64 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi8 ( __m256i s, __mmask128 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 ( __m256i s, __mmask64 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 ( __m128i s, __mmask256 m, __m128i a, __m128i b)
VPADDB__m128i _mm_mask_add_epi16 ( __m128i s, __mmask128 m, __m128i a, __m128i b)
VPADDB__m128i _mm_maskz_add_epi8 ( __m128i s, __mmask256 m, __m128i a, __m128i b)
VPADDB__m128i _mm_maskz_add_epi16 ( __m128i s, __mmask128 m, __m128i a, __m128i b)
VPADDB__m128i_mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i_mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD__m512i_mm512_add_epi32(__m512i a, __m512i b);
VPADDD__m512i_mm512_maskz_add_epi32(__mmask16 k, __m512i a, __m512i b);
VPADDD__m256i_mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD__m256i_mm256_maskz_add_epi32(__mmask8 k, __m256i a, __m256i b);
VPADDD__m128i_mm_maskz_add_epi32(__mmask16 k, __m128i a, __m128i b);
VPADDD__m128i_mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ__m512i_mm512_add_epi64(__m512i a, __m512i b);
VPADDQ__m512i_mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ__m256i_mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ__m256i_mm256_maskz_add_epi64(__mmask8 k, __m256i a, __m256i b);
VPADDQ__m128i_mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ__m128i_mm_maskz_add_epi64(__mmask8 k, __m128i a, __m128i b);
PADDB__m128i_mm_add_epi8 (__m128i a, __m128i b);
PADDW__m128i_mm_add_epi16 (__m128i a, __m128i b);
PADDD__m128i_mm_add_epi32 (__m128i a, __m128i b);
PADDQ__m128i_mm_add_epi64 (__m128i a, __m128i b);
VPADDQ__m256i_mm256_add_epi8 (__m256i a, __m256i b);
VPADDQ__m256i_mm256_add_epi16 (__m256i a, __m256i b);
VPADDQ__m256i_mm256_add_epi32 (__m256i a, __m256i b);
VPADDQ__m256i_mm256_add_epi64 (__m256i a, __m256i b);
PADDQ__m64___mm_add_pi8(__m64 m1, __m64 m2);
PADDQ__m64___mm_add_pi16(__m64 m1, __m64 m2);
PADDQ__m64___mm_add_pi32(__m64 m1, __m64 m2);
PADDQ__m64___mm_add_pi64(__m64 m1, __m64 m2);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPADDD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EC /r&lt;sup&gt;1&lt;/sup&gt; PADDSB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Add packed signed byte integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F EC /r PADDSB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed signed byte integers from xmm2/m128 and xmm1 saturate the results.</td>
</tr>
<tr>
<td>NP 0F ED /r&lt;sup&gt;1&lt;/sup&gt; PADDSW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Add packed signed word integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F ED /r PADDSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed signed word integers from xmm2/m128 and xmm1 and saturate the results.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG EC /r PADDSB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed signed byte integers from xmm3/m128 and xmm2 saturate the results.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG ED /r VPADDSW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed signed word integers from xmm3/m128 and xmm2 and saturate the results.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG EC /r VPADDSW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG ED /r VPADDSW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG EC /r VPADDSB xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed signed byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG EC /r VPADDSB ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed signed byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG EC /r VPADDSB zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed signed byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG ED /r VPADDSW xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed signed word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG ED /r VPADDSW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed signed word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG ED /r VPADDSW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed signed word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination operand is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDSB (With 64-bit Operands)

\[
\text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0])
\]

(* Repeat add operation for 2nd through 7th bytes *)

\[
\text{DEST}[63:56] := \text{SaturateToSignedByte}(\text{DEST}[63:56] + \text{SRC}[63:56])
\]

PADDSB (With 128-bit Operands)

\[
\text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0])
\]

(* Repeat add operation for 2nd through 14th bytes *)

\[
\text{DEST}[127:120] := \text{SaturateToSignedByte}(\text{DEST}[111:120] + \text{SRC}[127:120])
\]

VPADDSB (VEX.128 Encoded Version)

\[
\text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{SRC1}[7:0] + \text{SRC2}[7:0])
\]

(* Repeat subtract operation for 2nd through 14th bytes *)

\[
\text{DEST}[127:120] := \text{SaturateToSignedByte}(\text{SRC1}[111:120] + \text{SRC2}[127:120])
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

VPADDSB (VEX.256 Encoded Version)

\[
\text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{SRC1}[7:0] + \text{SRC2}[7:0])
\]

(* Repeat add operation for 2nd through 31st bytes *)

\[
\]
VPADDSB (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
    ELSE
      IF *merging-masking*
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+7:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

PADDSW (with 64-bit operands)

(* Repeat add operation for 2nd and 7th words *)

PADDSW (with 128-bit operands)

(* Repeat add operation for 2nd through 7th words *)

VPADDSW (VEX.128 Encoded Version)

DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] := 0

VPADDSW (VEX.256 Encoded Version)

DEST[15:0] := SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)

VPADDSW (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
    ELSE
      IF *merging-masking*
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+15:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalents

PADDSB __m64 _mm_adds_pi8(__m64 m1, __m64 m2)
(V)PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)
VPADDSB _m256i _mm256_adds_epi8 (__m256i a, __m256i b)
PADDSW __m64 _mm_adds_pi16(__m64 m1, __m64 m2)
(V)PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)
VPADDSW _m256i _mm256_adds_epi16 (__m256i a, __m256i b)
VPADDSB __m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW __m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB __m512i _mm512_mask_adds_epi8 (__m512i a, __m512i b)
VPADDSW __m512i _mm512_mask_adds_epi16 (__m512i a, __m512i b)
VPADDSB __m128i _mm_mask_adds_epi8 (__m128i a, __m128i b)
VPADDSW __m128i _mm_mask_adds_epi16 (__m128i a, __m128i b)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
## PADDUSB/PADDUSW—Add Packed Unsigned Integers With Unsigned Saturation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F DC /r(^1) PADDUSB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Add packed unsigned byte integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F DC /r PADDUSB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed unsigned byte integers from xmm2/m128 and xmm1 saturate the results.</td>
</tr>
<tr>
<td>NP 0F DD /r(^1) PADDUSW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Add packed unsigned word integers from mm/m64 and mm and saturate the results.</td>
</tr>
<tr>
<td>66 0F DD /r PADDUSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Add packed unsigned word integers from xmm2/m128 to xmm1 and saturate the results.</td>
</tr>
<tr>
<td>VEX.128.660F.WIG DC /r VPADDUSB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed unsigned byte integers from xmm3/m128 to xmm2 and saturate the results.</td>
</tr>
<tr>
<td>VEX.128.660F.WIG DD /r VPADDUSW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Add packed unsigned word integers from xmm3/m128 to xmm2 and saturate the results.</td>
</tr>
<tr>
<td>VEX.256.660F.WIG DC /r VPADDUSB ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add packed unsigned byte integers from ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>VEX.256.660F.WIG DD /r VPADDUSW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add packed unsigned word integers from ymm3/m256 and store the saturated results in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.660F.WIG DC /r VPADDUSB xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned byte integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.660F.WIG DC /r VPADDUSB ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned byte integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.660F.WIG DC /r VPADDUSB zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed unsigned byte integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.660F.WIG DD /r VPADDUSB xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned word integers from xmm2, and xmm3/m128 and store the saturated results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.660F.WIG DD /r VPADDUSW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Add packed unsigned word integers from ymm2, and ymm3/m256 and store the saturated results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.660F.WIG DD /r VPADDUSW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Add packed unsigned word integers from zmm2, and zmm3/m512 and store the saturated results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**
PADDUSB/PADDUSW—Add Packed Unsigned Integers With Unsigned Saturation

InstructionOperandEncoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg(r, w)</td>
<td>ModRMreg(r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg(w)</td>
<td>VEX.vvvv(r)</td>
<td>ModRMreg(r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg(w)</td>
<td>EVEX.vvvv(r)</td>
<td>ModRMreg(r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is a ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDUSB (With 64-bit Operands)

\[
\text{DEST}[7:0] := \text{SaturateToUnsignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0]);
\]

\[
\text{DEST}[63:56] := \text{SaturateToUnsignedByte}(\text{DEST}[63:56] + \text{SRC}[63:56])
\]

PADDUSB (With 128-bit Operands)

\[
\text{DEST}[7:0] := \text{SaturateToUnsignedByte}(\text{DEST}[7:0] + \text{SRC}[7:0]);
\]

\[
\text{DEST}[127:120] := \text{SaturateToUnSignedByte}(\text{DEST}[127:120] + \text{SRC}[127:120]);
\]

VPADDUSB (VEX.128 Encoded Version)

\[
\text{DEST}[7:0] := \text{SaturateToUnSignedByte}(\text{SRC1}[7:0] + \text{SRC2}[7:0]);
\]

\[
\text{DEST}[127:120] := \text{SaturateToUnsignedByte}(\text{SRC1}[111:120] + \text{SRC2}[127:120]);
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
VPADDUSB (VEX.256 Encoded Version)

\[
\text{DEST}[7:0] := \text{SaturateToUnsignedByte}(\text{SRC1}[7:0] + \text{SRC2}[7:0]);
\]

(* Repeat add operation for 2nd through 31st bytes *)

\[
\text{DEST}[255:248] := \text{SaturateToUnsignedByte}(\text{SRC1}[255:248] + \text{SRC2}[255:248]);
\]

PADDUSW (With 64-bit Operands)

\[
\text{DEST}[15:0] := \text{SaturateToUnsignedWord}(\text{DEST}[15:0] + \text{SRC}[15:0]);
\]

(* Repeat add operation for 2nd and 3rd words *)

\[
\text{DEST}[63:48] := \text{SaturateToUnsignedWord}(\text{DEST}[63:48] + \text{SRC}[63:48]);
\]

PADDUSW (With 128-bit Operands)

\[
\text{DEST}[15:0] := \text{SaturateToUnsignedWord}(\text{DEST}[15:0] + \text{SRC}[15:0]);
\]

(* Repeat add operation for 2nd through 7th words *)

\[
\text{DEST}[127:112] := \text{SaturateToUnsignedWord}(\text{DEST}[127:112] + \text{SRC}[127:112]);
\]

VPADDUSW (VEX.128 Encoded Version)

\[
\text{DEST}[15:0] := \text{SaturateToUnsignedWord}(\text{SRC1}[15:0] + \text{SRC2}[15:0]);
\]

(* Repeat subtract operation for 2nd through 7th words *)

\[
\text{DEST}[127:112] := \text{SaturateToUnsignedWord}(\text{SRC1}[127:112] + \text{SRC2}[127:112]);
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

VPADDUSW (VEX.256 Encoded Version)

\[
\text{DEST}[15:0] := \text{SaturateToUnsignedWord}(\text{SRC1}[15:0] + \text{SRC2}[15:0]);
\]

(* Repeat add operation for 2nd through 15th words *)

\[
\text{DEST}[255:240] := \text{SaturateToUnsignedWord}(\text{SRC1}[255:240] + \text{SRC2}[255:240])
\]

VPADDUSB (EVEX Encoded Versions)

\((\text{KL}, \text{VL}) = (16, 128), (32, 256), (64, 512)\)

\[
\text{FOR } j := 0 \text{ TO } \text{KL}-1
\]

\[
i := j \times 8
\]

IF \(k1[j] \text{ OR } \text{*no writemask*}\)

\[
\text{THEN } \text{DEST}[i+7:i] := \text{SaturateToUnsignedByte}(\text{SRC1}[i+7:i] + \text{SRC2}[i+7:i])
\]

ELSE

\[
\text{IF } \text{*merging-masking*} ; \text{merging-masking}
\]

\[
\text{THEN } \text{*DEST}[i+7:i] \text{ remains unchanged*}
\]

\[
\text{ELSE } \text{*zeroing-masking*} ; \text{zeroing-masking}
\]

\[
\text{DEST}[i+7:i] := 0
\]

\[
\text{FI}
\]

\[
\text{FI;}
\]

ENDFOR;

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]
VPADDUSw (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] = 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB __m64 ___mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW __m64 ___mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB __m128i ___mm_adds_epu8 (__m128i a, __m128i b)
(V)PADDUSW __m128i ___mm_adds_epu16 (__m128i a, __m128i b)
VPADDUSB __m256i ___mm256_adds_epu8 (__m256i a, __m256i b)
VPADDUSW __m256i ___mm256_adds_epu16 (__m256i a, __m256i b)
VPADDUSB __m512i ___mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW __m512i ___mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB __m512i ___mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i ___mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB __m512i ___mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW __m512i ___mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB __m256i ___mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i ___mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB __m256i ___mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW __m256i ___mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB __m128i ___mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i ___mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB __m128i ___mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW __m128i ___mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."
PALIGNR—Packed Align Right

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3A 0F /r ib¹</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1.</td>
</tr>
<tr>
<td>66 0F 3A 0F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WI</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Concatenate xmm2 and xmm3/m128, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.</td>
</tr>
<tr>
<td>G VEX.256.66.0F3A.WIG</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.WIG</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.WIG</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.WIG</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.</td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX, XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e., 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.
In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

EVEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block.

The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

---

**Figure 4-7. 256-bit VPALIGN Instruction Operation**

**Operation**

**PALIGNR (With 64-bit Operands)**

\[
\text{temp1}[127:0] = \text{CONCATENATE}(\text{DEST, SRC}) >> (\text{imm8*8}) \\
\text{DEST}[63:0] = \text{temp1}[63:0]
\]

**PALIGNR (With 128-bit Operands)**

\[
\text{temp1}[255:0] := ((\text{DEST}[127:0] \ll 128 \text{ OR SRC}[127:0]) >> (\text{imm8*8})) \\
\text{DEST}[127:0] := \text{temp1}[127:0] \\
\text{DEST}[\text{MAXVL}-1:128] \quad \text{(Unmodified)}
\]
VPALIGNR (VEX.128 Encoded Version)
\[
temp1[255:0] := ((SRC1[127:0] \ll 128) \text{ OR } SRC2[127:0]) \gg (\text{imm8} \times 8);
\]
\[
\text{DEST}[127:0] := temp1[127:0]
\]
\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

VPALIGNR (VEX.256 Encoded Version)
\[
temp1[255:0] := ((SRC1[127:0] \ll 128) \text{ OR } SRC2[127:0]) \gg (\text{imm8}[7:0] \times 8);
\]
\[
\text{DEST}[127:0] := temp1[127:0]
\]
\[
temp1[255:0] := ((SRC1[255:128] \ll 128) \text{ OR } SRC2[255:128]) \gg (\text{imm8}[7:0] \times 8);
\]
\[
\text{DEST}[\text{MAXVL}-1:128] := temp1[127:0]
\]

VPALIGNR (EVEX Encoded Versions)
\[
(KL, VL) = (16, 128), (32, 256), (64, 512)
\]
\[
\text{FOR } l := 0 \text{ TO } VL-1 \text{ with increments of 128 }
\]
\[
\quad \text{temp1}[255:0] := ((SRC1[l+127:l] \ll 128) \text{ OR } SRC2[l+127:l]) \gg (\text{imm8}[7:0] \times 8);
\]
\[
\quad \text{TMP\_DEST}[l+127:l] := temp1[127:0]
\]
\[
\text{ENDFOR;}
\]
\[
\text{FOR } j := 0 \text{ TO } KL-1
\]
\[
\quad i := j \times 8
\]
\[
\quad \text{IF } k1[j] \text{ OR } *\text{no writemask}*
\]
\[
\quad \quad \text{THEN } \text{DEST}[i+7:i] := \text{TMP\_DEST}[i+7:i]
\]
\[
\quad \quad \text{ELSE}
\]
\[
\quad \quad \quad \quad \quad \text{IF } *\text{merging\_masking}*$
\]
\[
\quad \quad \quad \quad \quad \text{THEN } *\text{DEST}[i+7:i] \text{ remains unchanged}*
\]
\[
\quad \quad \quad \quad \quad \text{ELSE } *\text{zeroing\_masking}*$
\]
\[
\quad \quad \quad \quad \quad \text{DEST}[i+7:i] = 0
\]
\[
\quad \quad \FI
\]
\[
\quad \FI
\]
\[
\text{ENDFOR;}
\]
\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

Intel C/C++ Compiler Intrinsic Equivalents
PALIGNR __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)
(V)PALIGNR __m128i __mm_alignr_epi8 (__m128i a, __m128i b, int n)
VPALIGNR __m256i __mm256_alignr_epi8 (__m256i a, __m256i b, const int n)
VPALIGNR __m512i __mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i __mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i __mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i __mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i __mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i __mm128_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i __mm128_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
## PAND—Logical AND

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F DB /r&lt;sup&gt;1&lt;/sup&gt; PAND mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Bitwise AND mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F DB /r PAND xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Bitwise AND of xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Bitwise AND of xmm3/m128 and xmm.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG DB /r VPAND ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 DB /r VPANDD xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 DB /r VPANDD ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 DB /r VPANDD zmm1 [k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 DB /r VPANDQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 DB /r VPANDQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 DB /r VPANDQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Notes:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with write-mask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PAND (64-bit Operand)
\[ \text{DEST} := \text{DEST AND SRC} \]

PAND (128-bit Legacy SSE Version)
\[ \text{DEST} := \text{DEST AND SRC} \]
\[ \text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \]

VPAND (VEX.128 Encoded Version)
\[ \text{DEST} := \text{SRC1 AND SRC2} \]
\[ \text{DEST}[\text{MAXVL}-1:128] := 0 \]

VPAND (VEX.256 Encoded Instruction)
\[ \text{DEST}[255:0] := (\text{SRC1}[255:0] \text{ AND SRC2}[255:0]) \]
\[ \text{DEST}[\text{MAXVL}-1:256] := 0 \]

VPANDD (EVEX Encoded Versions)
\[(KL, VL) = (4, 128), (8, 256), (16, 512) \]
\[ \text{FOR } j := 0 \text{ TO KL-1} \]
\[ \quad i := j \times 32 \]
\[ \quad \text{IF } k1[j] \text{ OR } *\text{no writemask}^* \]
\[ \quad \text{THEN} \]
\[ \quad \quad \text{IF (EVEX.b = 1) AND (SRC2 *is memory*)} \]
\[ \quad \quad \quad \text{THEN } \text{DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE AND SRC2}[31:0] \]
\[ \quad \quad \quad \text{ELSE } \text{DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE AND SRC2}[i+31:i] \]
\[ \quad \quad \text{FI;} \]
\[ \quad \text{ELSE} \]
\[ \quad \quad \text{IF *merging-masking*} \quad ; \text{merging-masking} \]
\[ \quad \quad \quad \text{THEN } *\text{DEST}[i+31:i] \text{ remains unchanged*} \]
\[ \quad \quad \quad \text{ELSE} \quad ; \text{zeroing-masking} \]
\[ \quad \quad \quad \text{DEST}[i+31:i] := 0 \]
\[ \quad \text{FI} \]
\[ \text{ENDFOR} \]
\[ \text{DEST}[\text{MAXVL}-1:VL] := 0 \]
VPANDQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        Fl;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
  ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPANDD __m512i _mm512_and_epi32( __m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDO __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDO __m512i _mm512_maskz_and_epi64( __mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64( __mmask8 k, __m256i a, __m256i b);
VPAND __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND __m128i _mm_and_si128 ( __m128i a, __m128i b)
VPAND __m256i _mm256_and_si256 ( __m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
## PANDN—Logical AND NOT

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP DF /r&lt;sup&gt;1&lt;/sup&gt;</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Bitwise AND NOT of mm/m64 and mm.</td>
</tr>
<tr>
<td>PANDN mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Bitwise AND NOT of xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>66 0F DF /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Bitwise AND NOT of xmm3/m128 and xmm2.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG DF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.Wig DF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.</td>
</tr>
<tr>
<td>VEX.512.66.0F.W0 DF /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPANDND xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPANDND ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPANDND zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPANDNO xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPANDNO ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPANDNO zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMrm/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMrm/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**Operation**

**PANDN (64-bit Operand)**

$\text{DEST} := \text{NOT(DEST)} \text{ AND SRC}$

**PANDN (128-bit Legacy SSE Version)**

$\text{DEST} := \text{NOT(DEST)} \text{ AND SRC}$

$\text{DEST}[\text{MAXVL}-1:128]$ (Unmodified)

**VPANDN (VEX.128 Encoded Version)**

$\text{DEST} := \text{NOT}(\text{SRC1}) \text{ AND SRC2}$

$\text{DEST}[\text{MAXVL}-1:128] := 0$

**VPANDN (VEX.256 Encoded Instruction)**

$\text{DEST}[255:0] := ((\text{NOT SRC1}[255:0]) \text{ AND SRC2}[255:0])$

$\text{DEST}[\text{MAXVL}-1:256] := 0$

**VPANDND (EVEX Encoded Versions)**

$(\text{KL, VL}) = (4, 128), (8, 256), (16, 512)$

FOR $j := 0 \text{ TO KL}-1$

$i := j * 32$

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN $\text{DEST}[i+31:i] := ((\text{NOT SRC1}[i+31:i]) \text{ AND SRC2}[31:0])$

ELSE $\text{DEST}[i+31:i] := ((\text{NOT SRC1}[i+31:i]) \text{ AND SRC2}[i+31:i])$

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *$\text{DEST}[i+31:i]$ remains unchanged*  
ELSE ; zeroing-masking

$\text{DEST}[i+31:i] := 0$

FI

ENDIF

ENDFOR

$\text{DEST}[\text{MAXVL}-1:VL] := 0$
VPANDNQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+63:i] := ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])
ELSE DEST[i+63:i] := (NOT SRC1[i+63:i]) AND SRC2[i+63:i]
FI;
ELSE
IF *merging-masking*; merging-masking
THEN \*DEST[i+63:i] remains unchanged*
ELSE \*zeroing-masking
DEST[i+63:i] := 0
FI
ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPANDND __m512i _mm512_andnot_epi32( __m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32( __mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_andnot_epi32(__m256i s, __m256i a, __m256i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32( __mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm128_andnot_epi32(__m128i s, __m128i a, __m128i b);
VPANDND __m128i _mm128_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm128_maskz_andnot_epi32( __mmask8 k, __m128i a, __m128i b);
VPANDN __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN __m128i _mm_andnot_si128 (__m128i a, __m128i b)
VPANDN __m256i _mm256_andnot_si256 ( __m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
INSTRUCTION SET REFERENCE, M-U

PAUSE—Spin Loop Hint

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 90</td>
<td>PAUSE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Gives hint to processor that improves performance of spin-wait loops.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Improves the performance of spin-wait loops. When executing a "spin-wait loop," processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Execute_Next/Instruction(Delay);

### Numeric Exceptions

None.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.
### PAVGB/PAVGW—Average Packed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E0 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.</td>
</tr>
<tr>
<td>PAVGB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E0, /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Average packed unsigned byte integers from xmm2/m128 and xmm1 with rounding.</td>
</tr>
<tr>
<td>PAVGB xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F E3 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Average packed unsigned word integers from mm2/m64 and mm1 with rounding.</td>
</tr>
<tr>
<td>PAVGW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E3 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Average packed unsigned word integers from xmm2/m128 and xmm1 with rounding.</td>
</tr>
<tr>
<td>PAVGW xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E0 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Average packed unsigned byte integers from xmm3/m128 and xmm2 with rounding.</td>
</tr>
<tr>
<td>VPAVGB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E3 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Average packed unsigned word integers from xmm3/m128 and xmm2 with rounding.</td>
</tr>
<tr>
<td>VPAVGW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E0 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1.</td>
</tr>
<tr>
<td>VPAVGB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E3 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Average packed unsigned word integers from ymm2, and ymm3/m256 with rounding and store to ymm1.</td>
</tr>
<tr>
<td>VPAVGW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG E0 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned byte integers from xmm2, and xmm3/m128 with rounding and store to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPAVGB xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG E0 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned byte integers from ymm2, and ymm3/m256 with rounding and store to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPAVGB ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG E0 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Average packed unsigned byte integers from zmm2, and zmm3/m512 with rounding and store to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPAVGB zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG E3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned word integers from xmm2, xmm3/m128 with rounding to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPAVGB xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG E3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Average packed unsigned word integers from ymm2, ymm3/m256 with rounding to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPAVGB ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG E3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Average packed unsigned word integers from zmm2, zmm3/m512 with rounding to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPAVGB zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:
Description
Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

Operation
PAVGB (with 64-bit Operands)
\[
\text{DEST}[7:0] := (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) >> 1; (* \text{Temp sum before shifting is 9 bits} *)
\]
\[
\text{DEST}[63:56] := (\text{SRC}[63:56] + \text{DEST}[63:56] + 1) >> 1;
\]

PAVGW (with 64-bit Operands)
\[
\text{DEST}[15:0] := (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) >> 1; (* \text{Temp sum before shifting is 17 bits} *)
\]
\[
\]

PAVGB (with 128-bit Operands)
\[
\text{DEST}[7:0] := (\text{SRC}[7:0] + \text{DEST}[7:0] + 1) >> 1; (* \text{Temp sum before shifting is 9 bits} *)
\]
\[
\text{DEST}[127:120] := (\text{SRC}[127:120] + \text{DEST}[127:120] + 1) >> 1;
\]

PAVGW (with 128-bit Operands)
\[
\text{DEST}[15:0] := (\text{SRC}[15:0] + \text{DEST}[15:0] + 1) >> 1; (* \text{Temp sum before shifting is 17 bits} *)
\]
\[
\]
VPAVGB (VEX.128 Encoded Version)
DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] := (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[MAXVL-1:128] := 0

VPAVGW (VEX.128 Encoded Version)
DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[MAXVL-1:128] := 0

VPAVGB (VEX.256 Encoded Instruction)
DEST[7:0] := (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 31)

VPAVGW (VEX.256 Encoded Instruction)
DEST[15:0] := (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 15)

VPAVGB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

VPAVGW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
      ; (* Temp sum before shifting is 17 bits *)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalents**

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_avg_epu8(__m256i a, __m256i b);
VPAVGW __m256i _mm256_avg_epu16(__m256i a, __m256i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__mmask8 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask16 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask8 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_avg_epu8(__m128i a, __m128i b);
VPAVGW __m128i _mm_avg_epu16(__m128i a, __m128i b);
VPAVGB __m128i _mm_mask_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
### PBLENDVB—Variable Blend Packed Bytes

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 10 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select byte values from xmm1 and xmm2/m128 from mask specified in the high bit of each byte in XMM0 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 4C /r /is4</td>
<td>RVMR</td>
<td>V/V</td>
<td>AVX</td>
<td>Select byte values from xmm2 and xmm3/m128 using mask bits in the specified mask register, xmm4, and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 4C /r /is4</td>
<td>RVMR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select byte values from ymm2 and ymm3/m256 from mask specified in the high bit of each byte in ymm4 and store the values into ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>&lt;XMM0&gt;</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMR</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8[7:4]</td>
</tr>
</tbody>
</table>

### Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source operand is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute VPBLENDVB encoded with a VEX prefix will cause a #UD exception.

### Operation

**PBLENDVB (128-bit Legacy SSE Version)**

\[
\text{MASK} := \text{XMM0} \\
\begin{align*}
\text{IF} \ (\text{MASK}[7]) = 1 & \ \text{THEN} \ \text{DEST}[7:0] := \text{SRC}[7:0]; \\
\text{ELSE} \ \text{DEST}[7:0] & := \text{DEST}[7:0]; \\
\text{IF} \ (\text{MASK}[15]) = 1 & \ \text{THEN} \ \text{DEST}[15:8] := \text{SRC}[15:8]; \\
\text{ELSE} \ \text{DEST}[15:8] & := \text{DEST}[15:8]; \\
\text{IF} \ (\text{MASK}[23]) = 1 & \ \text{THEN} \ \text{DEST}[23:16] := \text{SRC}[23:16]
\end{align*}
\]
ELSE DEST[31:24] := DEST[31:24];
IF (MASK[47] = 1) THEN DEST[47:40] := SRC[47:40]
ELSE DEST[47:40] := DEST[47:40];
IF (MASK[63] = 1) THEN DEST[63:56] := SRC[63:56]
ELSE DEST[63:56] := DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] := SRC[71:64]
ELSE DEST[71:64] := DEST[71:64];
ELSE DEST[79:72] := DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] := SRC[87:80]
ELSE DEST[87:80] := DEST[87:80];
ELSE DEST[95:88] := DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] := SRC[103:96]
ELSE DEST[103:96] := DEST[103:96];
ELSE DEST[111:104] := DEST[111:104];
ELSE DEST[119:112] := DEST[119:112];
ELSE DEST[127:120] := DEST[127:120];
DEST[MAXVL-1:128] (Unmodified)

VPBLENDVB (VEX.128 Encoded Version)
MASK := SRC3
IF (MASK[7] = 1) THEN DEST[7:0] := SRC2[7:0];
ELSE DEST[7:0] := SRC1[7:0];
ELSE DEST[15:8] := SRC1[15:8];
ELSE DEST[31:24] := SRC1[31:24];
IF (MASK[47] = 1) THEN DEST[47:40] := SRC2[47:40]
ELSE DEST[47:40] := SRC1[47:40];
IF (MASK[63] = 1) THEN DEST[63:56] := SRC2[63:56]
ELSE DEST[63:56] := SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] := SRC2[71:64]
ELSE DEST[71:64] := SRC1[71:64];
ELSE DEST[79:72] := SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] := SRC2[87:80]
ELSE DEST[87:80] := SRC1[87:80];
ELSE DEST[95:88] := SRC[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] := SRC2[103:96]
ELSE DEST[103:96] := SRC[103:96];
ELSE DEST[111:104] := SRC[111:104];
ELSE DEST[119:112] := SRC[119:112];
ELSE DEST[127:120] := SRC[127:120];
DEST[MAXVL-1:128] := 0

VPBLENDVB (VEX.256 Encoded Version)
MASK := SRC3
IF (MASK[7] == 1) THEN DEST[7:0] := SRC2[7:0];
ELSE DEST[7:0] := SRC[7:0];
ELSE DEST[15:8] := SRC[15:8];
IF (MASK[31] == 1) THEN DEST[31:24] := SRC2[31:24];
ELSE DEST[31:24] := SRC[31:24];
IF (MASK[47] == 1) THEN DEST[47:40] := SRC2[47:40];
ELSE DEST[47:40] := SRC[47:40];
IF (MASK[63] == 1) THEN DEST[63:56] := SRC2[63:56];
ELSE DEST[63:56] := SRC[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] := SRC2[71:64];
ELSE DEST[71:64] := SRC[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] := SRC2[79:72];
ELSE DEST[79:72] := SRC[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] := SRC2[87:80];
ELSE DEST[87:80] := SRC[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] := SRC2[95:88];
ELSE DEST[95:88] := SRC[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] := SRC2[103:96];
ELSE DEST[103:96] := SRC[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] := SRC2[111:104];
ELSE DEST[111:104] := SRC[111:104];
ELSE DEST[119:112] := SRC[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] := SRC2[127:120];
ELSE DEST[127:120] := SRC[127:120];
IF (MASK[143] == 1) THEN DEST[143:136] := SRC2[143:136];
ELSE DEST[143:136] := SRC[1143:136];
ELSE DEST[151:144] := SRC[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] := SRC2[159:152];
ELSE DEST[159:152] := SRC[159:152];
ELSE DEST[175:168] := SRC1[175:168];
ELSE DEST[183:176] := SRC1[183:176];
ELSE DEST[191:184] := SRC1[191:184];
ELSE DEST[199:192] := SRC1[199:192];
ELSE DEST[207:200] := SRC1[207:200];
ELSE DEST[231:224] := SRC1[231:224];
ELSE DEST[239:232] := SRC1[239:232];
ELSE DEST[255:248] := SRC1[255:248];

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PBLENDVB __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
VPBLENDVB __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD If VEX.W = 1.
PBLENDW—Blend Packed Words

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0E /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Select words from xmm1 and xmm2/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VPBLENDW xmm1, xmm2/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Select words from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VPBLENDW ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select words from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

PBLENDW (128-bit Legacy SSE Version)

IF (imm8[0] = 1) THEN DEST[15:0] := SRC[15:0]  
ELSE DEST[15:0] := DEST[15:0]  
ELSE DEST[31:16] := DEST[31:16]  
ELSE DEST[79:64] := DEST[79:64]  
ELSE DEST[95:80] := DEST[95:80]  
ELSE DEST[111:96] := DEST[111:96]  

**VPBLENDW (VEX.128 Encoded Version)**

IF (imm8[0] = 1) THEN DEST[15:0] := SRC2[15:0]
ELSE DEST[15:0] := SRC1[15:0]
ELSE DEST[31:16] := SRC1[31:16]
ELSE DEST[79:64] := SRC1[79:64]
ELSE DEST[95:80] := SRC1[95:80]
ELSE DEST[111:96] := SRC1[111:96]
DEST[MAXVL-1:128] := 0

**VPBLENDW (VEX.256 Encoded Version)**

IF (imm8[0] == 1) THEN DEST[15:0] := SRC2[15:0]
ELSE DEST[15:0] := SRC1[15:0]
ELSE DEST[31:16] := SRC1[31:16]
ELSE DEST[79:64] := SRC1[79:64]
ELSE DEST[95:80] := SRC1[95:80]
ELSE DEST[111:96] := SRC1[111:96]
IF (imm8[0] == 1) THEN DEST[143:128] := SRC2[143:128]
ELSE DEST[143:128] := SRC1[143:128]
ELSE DEST[159:144] := SRC1[159:144]
Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);
VPBLENDW __m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions,” additionally:
#UD If VEX.L = 1 and AVX2 = 0.
PCLMULQDQ—Carry-Less Multiplication Quadword

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 44 /r ib PCLMULQDQ xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>PCLMULQDQ</td>
<td>Carry-less multiplication of one quadword of xmm1 by one quadword of xmm2/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm1 and xmm2/m128 should be used.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>PCLMULQDQ AVX</td>
<td>Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.WIG 44 /r ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>PCLMULQDQ AVX</td>
<td>Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8</td>
<td>C</td>
<td>V/V</td>
<td>PCLMULQDQ AVX512VL</td>
<td>Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.WIG 44 /r ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8</td>
<td>C</td>
<td>V/V</td>
<td>PCLMULQDQ AVX512VL</td>
<td>Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.WIG 44 /r ib VPCLMULQDQ zmm1, zmm2, zmm3/m512, imm8</td>
<td>C</td>
<td>V/V</td>
<td>PCLMULQDQ AVX512F</td>
<td>Carry-less multiplication of one quadword of zmm2 by one quadword of zmm3/m512, stores the 128-bit result in zmm1. The immediate is used to determine which quadwords of zmm2 and zmm3/m512 should be used.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-13, other bits of the immediate byte are ignored.

The EVEX encoded form of this instruction does not support memory fault suppression.
The first source operand and the destination operand are the same and must be a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for imm8.

Table 4-13. PCLMULQDQ Quadword Selection of Immediate Byte

<table>
<thead>
<tr>
<th>Imm[4]</th>
<th>Imm[0]</th>
<th>PCLMULQDQ Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>CL_MUL( SRC2[63:0], SRC1[63:0] )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CL_MUL( SRC2[63:0], SRC1[127:64] )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CL_MUL( SRC2[127:64], SRC1[63:0] )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CL_MUL( SRC2[127:64], SRC1[127:64] )</td>
</tr>
</tbody>
</table>

NOTES:
1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for imm8.

Table 4-14. Pseudo-Op and PCLMULQDQ Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>Imm8 Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCLMULLQDQ</td>
<td>0000_0000B</td>
</tr>
<tr>
<td>xmm1, xmm2</td>
<td></td>
</tr>
<tr>
<td>PCLMULHQLDQ</td>
<td>0000_0001B</td>
</tr>
<tr>
<td>xmm1, xmm2</td>
<td></td>
</tr>
<tr>
<td>PCLMULLHQDQ</td>
<td>0001_0000B</td>
</tr>
<tr>
<td>xmm1, xmm2</td>
<td></td>
</tr>
<tr>
<td>PCLMULHQQDQ</td>
<td>0001_0001B</td>
</tr>
<tr>
<td>xmm1, xmm2</td>
<td></td>
</tr>
</tbody>
</table>

Operation

define PCLMUL128(X,Y):  // helper function
    FOR i := 0 to 63:
        TMP [i] := X[0] and Y[i]
    FOR j := 1 to i:
        TMP [i] := TMP [i] xor (X[j] and Y[i-j])
    DEST[i] := TMP[i]
    FOR i := 64 to 126:
        TMP [i] := 0
    FOR j := i - 63 to 63:
        TMP [i] := TMP [i] xor (X[j] and Y[i-j])
    DEST[i] := TMP[i]
DEST[127] := 0;
RETURN DEST // 128b vector
PCLMULQDQ (SSE Version)

IF imm8[0] = 0:
   TEMP1 := SRC1.qword[0]
ELSE:
   TEMP1 := SRC1.qword[1]
IF imm8[4] = 0:
   TEMP2 := SRC2.qword[0]
ELSE:
   TEMP2 := SRC2.qword[1]
DEST[127:0] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:128] := 0

VPCLMULQDQ (128b and 256b VEX Encoded Versions)

(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
   IF imm8[0] = 0:
      TEMP1 := SRC1.xmm[i].qword[0]
   ELSE:
      TEMP1 := SRC1.xmm[i].qword[1]
   IF imm8[4] = 0:
      TEMP2 := SRC2.xmm[i].qword[0]
   ELSE:
      TEMP2 := SRC2.xmm[i].qword[1]
   DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0

VPCLMULQDQ (EVEX Encoded Version)

(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
   IF imm8[0] = 0:
      TEMP1 := SRC1.xmm[i].qword[0]
   ELSE:
      TEMP1 := SRC1.xmm[i].qword[1]
   IF imm8[4] = 0:
      TEMP2 := SRC2.xmm[i].qword[0]
   ELSE:
      TEMP2 := SRC2.xmm[i].qword[1]
   DEST.xmm[i] := PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ __m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)
PCLMULQDQ __m256i _mm256_clmulepi64_epi128(__m256i, __m256i, const int);
PCLMULQDQ __m512i _mm512_clmulepi64_epi128(__m512i, __m512i, const int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD
   If VEX.L = 1.
EVEX-encoded: See Table 2-50, “Type E4NF Class Exception Conditions.”
# PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 74 /r&lt;sup&gt;1&lt;/sup&gt; PCMPEQB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed bytes in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 74 /r PCMPEQB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed bytes in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>NP 0F 75 /r&lt;sup&gt;1&lt;/sup&gt; PCMPEQW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed words in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 75 /r PCMPEQW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed words in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>NP 0F 76 /r&lt;sup&gt;1&lt;/sup&gt; PCMPEQD mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed doublewords in mm/m64 and mm for equality.</td>
</tr>
<tr>
<td>66 0F 76 /r PCMPEQD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed doublewords in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 74 /r VPCMPEQB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed bytes in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 75 /r VPCMPEQW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed words in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 76 /r VPCMPEQD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed doublewords in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed bytes in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 75 /r VPCMPEQW ymm1, ymm2, ymm3 /m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed words in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 76 /r VPCMPEQD ymm1, ymm2, ymm3 /m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed doublewords in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
</tbody>
</table>
**Description**

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPFQBW: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

**Operation**

**PCMPEQB (With 64-bit Operands)**

IF DEST[7:0] = SRC[7:0]
THEN DEST[7:0] := FFH;
ELSE DEST[7:0] := 0; FI;

(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)

IF DEST[63:56] = SRC[63:56]
THEN DEST[63:56] := FFH;
ELSE DEST[63:56] := 0; FI;

**COMPARE_BYTES_EQUAL (SRC1, SRC2)**

IF SRC[7:0] = SRC2[7:0]
THEN DEST[7:0] := FFH;
ELSE DEST[7:0] := 0; FI;

(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)

IF SRC[127:120] = SRC2[127:120]
THEN DEST[127:120] := FFH;
ELSE DEST[127:120] := 0; FI;

**COMPARE_WORDS_EQUAL (SRC1, SRC2)**

IF SRC[15:0] = SRC2[15:0]
THEN DEST[15:0] := FFFFH;
ELSE DEST[15:0] := 0; FI;

(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)

IF SRC[127:112] = SRC2[127:112]
THEN DEST[127:112] := FFFFH;
ELSE DEST[127:112] := 0; FI;

**COMPARE_DWORDS_EQUAL (SRC1, SRC2)**

IF SRC[31:0] = SRC2[31:0]
THEN DEST[31:0] := FFFFFFFFH;
ELSE DEST[31:0] := 0; FI;

(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)

IF SRC[127:96] = SRC2[127:96]
THEN DEST[127:96] := FFFFFFFFH;
ELSE DEST[127:96] := 0; FI;

**PCMPEQB (With 128-bit Operands)**

DEST[127:0] := COMPARE_BYTES_EQUAL(DEST[127:0], SRC[127:0])

DEST[MAXVL-1:128] (Unmodified)
VPCMPEQB (VEX.128 Encoded Version)
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0

VPCMPEQB (VEX.256 Encoded Version)
DEST[127:0] := COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:256] := 0

VPCMPEQB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
  i := j * 8
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP := SRC1[i+7:i] == SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE DEST[j] := 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

PCMPEQW (With 64-bit Operands)
IF DEST[15:0] = SRC[15:0]
  THEN DEST[15:0] := FFFFH;
  ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
  THEN DEST[63:48] := FFFFH;
  ELSE DEST[63:48] := 0; FI;

PCMPEQW (With 128-bit Operands)
DEST[127:0] := COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPEQW (VEX.128 Encoded Version)
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] := 0

VPCMPEQW (VEX.256 Encoded Version)
DEST[127:0] := COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:256] := 0
VPCMPEQW (EVEX Encoded Versions)  
(KL, VL) = (8, 128), (16, 256), (32, 512)  
FOR j := 0 TO KL-1  
i := j * 16  
IF k2[j] OR *no writemask*  
THEN  
   /* signed comparison */  
   CMP := SRC1[i+15:i] == SRC2[i+15:i];  
   IF CMP = TRUE  
      THEN DEST[j] := 1;  
      ELSE DEST[j] := 0; FI;  
   ELSE DEST[j] := 0 ; zeroing-masking onlyFI;  
ENDFOR  
DEST[MAX_KL-1:KL] := 0  

PCMPEQD (With 64-bit Operands)  
IF DEST[31:0] = SRC[31:0]  
THEN DEST[31:0] := FFFFFFFFH;  
ELSE DEST[31:0] := 0; FI;  
THEN DEST[63:32] := FFFFFFFFH;  
ELSE DEST[63:32] := 0; FI;  

PCMPEQD (With 128-bit Operands)  
DEST[127:0] := COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])  
DEST[MAXVL-1:128] := 0  

VPCMPEQD (VEX.128 Encoded Version)  
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])  
DEST[MAXVL-1:128] := 0  

VPMCPEQD (VEX.256 Encoded Version)  
DEST[127:0] := COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])  
DEST[MAXVL-1:256] := 0  

VPCMPEQD (EVEX Encoded Versions)  
(KL, VL) = (4, 128), (8, 256), (16, 512)  
FOR j := 0 TO KL-1  
i := j * 32  
IF k2[j] OR *no writemask*  
THEN  
   /* signed comparison */  
   IF (EVEX.b = 1) AND (SRC2 *is memory*)  
      THEN CMP := SRC1[i+31:i] = SRC2[31:0];  
      ELSE CMP := SRC1[i+31:i] = SRC2[i+31:i];  
      FI;  
   IF CMP = TRUE  
      THEN DEST[j] := 1;  
      ELSE DEST[j] := 0; FI;  
   ELSE DEST[j] := 0 ; zeroing-masking onlyFI;  
ENDFOR
DEST[MAX_KL-1:KL] := 0

**Intel C/C++ Compiler Intrinsic Equivalents**

- \( \text{VPCMPEQB}\_\text{mmask64 \_mm512\_cmpeq\_epi8\_mask(__m512i a, __m512i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask64 \_mm512\_mask\_cmpeq\_epi8\_mask(__mmask64 k, __m512i a, __m512i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask32 \_mm256\_cmpeq\_epi8\_mask(__m256i a, __m256i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask32 \_mm256\_mask\_cmpeq\_epi8\_mask(__mmask32 k, __m256i a, __m256i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask16 \_mm\_cmpeq\_epi8\_mask(__m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask16 \_mm\_mask\_cmpeq\_epi8\_mask(__mmask16 k, __m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask32 \_mm512\_mask\_cmpeq\_epi16\_mask(__m512i a, __m512i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask32 \_mm512\_mask\_cmpeq\_epi32\_mask(__m512i a, __m512i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask16 \_mm\_cmpeq\_epi16\_mask(__m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask16 \_mm\_mask\_cmpeq\_epi16\_mask(__mmask16 k, __m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask8 \_mm\_cmpeq\_epi16\_mask(__m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask8 \_mm\_mask\_cmpeq\_epi16\_mask(__mmask8 k, __m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask16 \_mm512\_cmpeq\_epi32\_mask(__m512i a, __m512i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask8 \_mm256\_cmpeq\_epi32\_mask(__m256i a, __m256i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask8 \_mm256\_mask\_cmpeq\_epi32\_mask(__mmask8 k, __m256i a, __m256i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask8 \_mm\_cmpeq\_epi32\_mask(__m128i a, __m128i b);} \)
- \( \text{VPCMPEQB}\_\text{mmask8 \_mm\_mask\_cmpeq\_epi32\_mask(__mmask8 k, __m128i a, __m128i b);} \)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

- Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
- EVEX-encoded VPCMPEQD, see Table 2-49, “Type E4 Class Exception Conditions.”
- EVEX-encoded VPCMPEQBW, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
**PCMPEQQ—Compare Packed Qword Data for Equal**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 29 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed qwords in xmm2/m128 and xmm1 for equality.</td>
</tr>
<tr>
<td>PCMPEQQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 29 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed quadwords in xmm3/m128 and xmm2 for equality.</td>
</tr>
<tr>
<td>VPCMPEQQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 29 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed quadwords in ymm3/m256 and ymm2 for equality.</td>
</tr>
<tr>
<td>VPCMPEQQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEK encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.
Operation

PCMPEQQ (With 128-bit Operands)

IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0; FI;

IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] := 0; FI;

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] := FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] := 0; FI;

IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] := FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] := 0; FI;

VPCMPEQQ (VEX.128 Encoded Version)

DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1, SRC2)

DEST[MAXVL-1:128] := 0

VPCMPEQQ (VEX.256 Encoded Version)

DEST[127:0] := COMPARE_QWORDS_EQUAL(SRC1[127:0], SRC2[127:0])


DEST[MAXVL-1:256] := 0

VPCMPEQQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] = SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] = SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE DEST[j] := 0 ; zeroing-masking only
    FI;
ENDFOR

DEST[MAX_KL-1:KL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask( __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm256_cmpeq_epi64_mask( __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm_cmpeq_epi64_mask( __m128i a, __m128i b);
VPCMPEQQ __mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ __m128i _mm_cmpeq_epi64( __m128i a, __m128i b);
VPCMPEQQ __m256i _mm256_cmpeq_epi64( __m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPEQQ, see Table 2-49, “Type E4 Class Exception Conditions.”
**PCMPESTRI—Packed Compare Explicit Length Strings, Return Index**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 61 /r imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.</td>
</tr>
<tr>
<td>PCMPESTRI xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F3A 61 /ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.</td>
</tr>
<tr>
<td>VPCMPESTRI xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The instruction compares and processes data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to the count register (ECX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
- SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
- OFlag - IntRes2[0]
- AFlag – Reset
- PFlag – Reset

**Effective Operand Size**

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Length 1</th>
<th>Length 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>ECX</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>ECX</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>ECX</td>
</tr>
<tr>
<td>64 bit + REX.W</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>RAX</td>
<td>RDX</td>
<td>ECX</td>
</tr>
</tbody>
</table>
Intel C/C++ Compiler Intrinsic Equivalent For Returning Index
int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results
int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions,” additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD If VEX.L = 1.
If VEX.vvvv ≠ 1111B.
**PCMPESTRM—Packed Compare Explicit Length Strings, Return Mask**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 60 /r imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A 60 /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in XMM0.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag – Reset if IntRes2 is equal to zero, set otherwise
- ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
- SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
- OFlag -IntRes2[0]
- AFlag – Reset
- PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.
Effective Operand Size

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Length 1</th>
<th>Length 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>XMM0</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>XMM0</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>EAX</td>
<td>EDX</td>
<td>XMM0</td>
</tr>
<tr>
<td>64 bit + REX.W</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>RAX</td>
<td>RDX</td>
<td>XMM0</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD       If VEX.L = 1.
If VEX.vvvv ≠ 1111B.
## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 64 /r&lt;sup&gt;1&lt;/sup&gt; PCMPGTB mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed signed byte integers in mm and mm/m64 for greater than.</td>
</tr>
<tr>
<td>66 0F 64 /r PCMPGTB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 for greater than.</td>
</tr>
<tr>
<td>NP 0F 65 /r&lt;sup&gt;1&lt;/sup&gt; PCMPGTW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed signed word integers in mm and mm/m64 for greater than.</td>
</tr>
<tr>
<td>66 0F 65 /r PCMPGTW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed word integers in xmm1 and xmm2/m128 for greater than.</td>
</tr>
<tr>
<td>NP 0F 66 /r&lt;sup&gt;1&lt;/sup&gt; PCMPGTD mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Compare packed signed doubleword integers in mm and mm/m64 for greater than.</td>
</tr>
<tr>
<td>66 0F 66 /r PCMPGTD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed signed doubleword integers in xmm1 and xmm2/m128 for greater than.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 64 /r VPCMPGTB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 65 /r VPCMPGTW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed word integers in xmm2 and xmm3/m128 for greater than.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 66 /r VPCMPGTD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed doubleword integers in xmm2 and xmm3/m128 for greater than.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 64 /r VPCMPGTB ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 65 /r VPCMPGTW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed word integers in ymm2 and ymm3/m256 for greater than.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 66 /r VPCMPGTD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed doubleword integers in ymm2 and ymm3/m256 for greater than.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 66 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 66 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 66 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask: k2.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
</tbody>
</table>
INSTRUCTION SET REFERENCE, M-U

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

NOTES:


Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTDT: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

**Operation**

**PCMPGTB (with 64-bit Operands)**

- IF DEST[7:0] > SRC[7:0]  
  THEN DEST[7:0] := FFH;  
  ELSE DEST[7:0] := 0; FI;

- (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)

- IF DEST[63:56] > SRC[63:56]  
  THEN DEST[63:56] := FFH;  
  ELSE DEST[63:56] := 0; FI;

**COMPARE_BYTES_GREATER (SRC1, SRC2)**

- IF SRC1[7:0] > SRC2[7:0]  
  THEN DEST[7:0] := FFH;  
  ELSE DEST[7:0] := 0; FI;

- (* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)

- IF SRC1[127:120] > SRC2[127:120]  
  THEN DEST[127:120] := FFH;  
  ELSE DEST[127:120] := 0; FI;

**COMPARE_WORDS_GREATER (SRC1, SRC2)**

- IF SRC1[15:0] > SRC2[15:0]  
  THEN DEST[15:0] := FFFFH;  
  ELSE DEST[15:0] := 0; FI;

- (* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)

  THEN DEST[127:112] := FFFFH;  
  ELSE DEST[127:112] := 0; FI;

**COMPARE_DWORDS_GREATER (SRC1, SRC2)**

- IF SRC1[31:0] > SRC2[31:0]  
  THEN DEST[31:0] := FFFFFFFFH;  
  ELSE DEST[31:0] := 0; FI;

- (* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)

**PCMPGTB (with 128-bit Operands)**

- DEST[127:0] := COMPARE_BYTES_GREATER(DEST[127:0], SRC[127:0])

- DEST[MAXVL-1:128] (Unmodified)
VPCMPGTB (VEX.128 Encoded Version)
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPCMPGTB (VEX.256 Encoded Version)
DEST[127:0] := COMPARE_BYTES_GREATER(SRC1[127:0], SRC2[127:0])
DEST[MAXVL-1:256] := 0

VPCMPGTB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k2[j] OR *no writemask*
    THEN
      /* signed comparison */
      CMP := SRC1[i+7:i] > SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE DEST[j] := 0; FI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

PCMPGTW (With 64-bit Operands)
IF DEST[15:0] > SRC[15:0]
  THEN DEST[15:0] := FFFFH;
  ELSE DEST[15:0] := 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] > SRC[63:48]
  THEN DEST[63:48] := FFFFH;
  ELSE DEST[63:48] := 0; FI;

PCMPGTW (With 128-bit Operands)
DEST[127:0] := COMPARE_WORDS_GREATER(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPGTW (VEX.128 Encoded Version)
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPCMPGTW (VEX.256 Encoded Version)
DEST[127:0] := COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[MAXVL-1:256] := 0
VPCMPGTW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP := SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE DEST[j] := 0 ; zeroing-masking only FI;
        FI;
    ENDFOR
DEST[MAX_KL-1:KL] := 0

PCMPGT (with 64-bit Operands)
IF DEST[31:0] > SRC[31:0]
    THEN DEST[31:0] := FFFFFFFFH;
    ELSE DEST[31:0] := 0; FI;
    THEN DEST[63:32] := FFFFFFFFH;
    ELSE DEST[63:32] := 0; FI;

PCMPGT (with 128-bit Operands)
DEST[127:0] := COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPGTD (VEX.128 Encoded Version)
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0

VPCMPGTD (VEX.256 Encoded Version)
DEST[127:0] := COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:256] := 0

VPCMPGTD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP := SRC1[i+31:i] > SRC2[i+31:i];
                FI;
                IF CMP = TRUE
                    THEN DEST[j] := 1;
                    ELSE DEST[j] := 0; FI;
            ELSE DEST[j] := 0 ; zeroing-masking only FI;
        FI;
    ENDFOR
DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPGTD, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
PCMPGTQ—Compare Packed Data for Greater Than

### Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 37 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Compare packed qwords in xmm2/m128 and xmm1 for greater than.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1G 37 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed qwords in xmm2 and xmm3/m128 for greater than.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1G 37 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed qwords in ymm2 and ymm3/m256 for greater than.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 37 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 37 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 37 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Operation

COMPARE_QWORDS_GREATER (SRC1, SRC2)

IF SRC1[63:0] > SRC2[63:0]
THEN DEST[63:0] := FFFFFFFFFFFFFFFF;
ELSE DEST[63:0] := 0; FI;
IF SRC1[127:64] > SRC2[127:64]
THEN DEST[127:64] := FFFFFFFFFFFFFFFF;
ELSE DEST[127:64] := 0; FI;

VPCMPGTQ (VEX.128 Encoded Version)
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] := 0

VPCMPGTQ (VEX.256 Encoded Version)
DEST[127:0] := COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:256] := 0

VPCMPGTQ (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k2[j] OR *no writemask*
THEN
  /* signed comparison */
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN CMP := SRC1[63:i+63:i] > SRC2[63:0];
    ELSE CMP := SRC1[63:i+63:i] > SRC2[63:i+63:i];
  FI;
  IF CMP = TRUE
    THEN DEST[j] := 1;
    ELSE DEST[j] := 0; FI;
ELSE DEST[j] := 0 ; zeroing-masking only
FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b);
VPCMPGTQ __m256i _mm256_cmpgt_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPCMPGTQ, see Table 2-49, “Type E4 Class Exception Conditions.”
PCMPISTRI—Packed Compare Implicit Length Strings, Return Index

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 OF 3A 63 /r imm8</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.</td>
</tr>
<tr>
<td>PCMPISTRI xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F:3A.WIG 63 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.</td>
</tr>
<tr>
<td>VPCMPISTRI xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RM</td>
</tr>
</tbody>
</table>

Description

The instruction compares data from two strings based on the encoded value in the imm8 control byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPDESTRI / PCMPDESTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag - Reset if IntRes2 is equal to zero, set otherwise
- ZFlag - Set if any byte/word of xmm2/mem128 is null, reset otherwise
- SFlag - Set if any byte/word of xmm1 is null, reset otherwise
- OFlag - IntRes2[0]
- AFlag - Reset
- PFlag - Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>ECX</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>ECX</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>ECX</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```
int __m128i_mm_cmpistri(__m128i a, __m128i b, const int mode);
```
Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```c
int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__m128i a, __m128i b, const int mode);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, “Type 4 Class Exception Conditions,” additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

- **#UD**
  - If VEX.L = 1.
  - If VEX.vvvv ≠ 1111B.
PCMPISTRM—Packed Compare Implicit Length Strings, Return Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 62 /r imm8 PCMPISTRM xmm1, xmm2/m128, imm8</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_2</td>
<td>Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in XMM0.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM xmm1, xmm2/m128, imm8</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in XMM0.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”) generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

- CFlag - Reset if IntRes2 is equal to zero, set otherwise
- ZFlag - Set if any byte/word of xmm2/mem128 is null, reset otherwise
- SFlag - Set if any byte/word of xmm1 is null, reset otherwise
- OFlag - IntRes2[0]
- AFlag - Reset
- PFlag - Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

<table>
<thead>
<tr>
<th>Operating mode/size</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>XMM0</td>
</tr>
<tr>
<td>32 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>XMM0</td>
</tr>
<tr>
<td>64 bit</td>
<td>xmm</td>
<td>xmm/m128</td>
<td>XMM0</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode);
Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, “Type 4 Class Exception Conditions,” additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and:

#UD  If VEX.L = 1.
      If VEX.vvvv ≠ 1111B.
PCONFIG—Platform Configuration

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 C5 PCONFIG</td>
<td>A</td>
<td>V/V</td>
<td>PCONFIG</td>
<td>This instruction is used to execute functions for configuring platform features.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The PCONFIG instruction allows software to configure certain platform features. It supports these features with multiple leaf functions, selecting a leaf function using the value in EAX.

Depending on the leaf function, the registers RBX, RCX, and RDX may be used to provide input information or for the instruction to report output information. Addresses and operands are 32 bits outside 64-bit mode and are 64 bits in 64-bit mode. The value of CS.D does not affect operand size or address size.

Executions of PCONFIG may fail for platform-specific reasons. An execution reports failure by setting the ZF flag and loading EAX with a non-zero failure reason; a successful execution clears ZF and EAX.

Each PCONFIG leaf function applies to a specific hardware block called a PCONFIG target. The leaf function is supported only if the processor supports that target. Each target is associated with a numerical target identifier, and CPUID leaf 1BH (PCONFIG information) enumerates the identifiers of the supported targets. An attempt to execute an undefined leaf function, or a leaf function that applies to an unsupported target identifier, results in a general-protection exception (#GP).

**Leaf Function MKTME_KEY_PROGRAM**

As of this writing, the only defined PCONFIG leaf function is used for key programming for total memory encryption-multi-key (TME-MK). This leaf function is called MKTME_KEY_PROGRAM and it pertains to the TME-MK target, which has target identifier 1. The leaf function is selected by loading EAX with value 0. The MKTME_KEY_PROGRAM leaf function uses the EBX (or RBX) register for additional input information.

Software uses the MKTME_KEY_PROGRAM leaf function to manage the encryption key associated with a particular key identifier (KeyID). The leaf function uses a data structure called the **TME-MK key programming structure** (MKTME_KEY_PROGRAM_STRUCT). Software provides the address of the structure (as an offset in the DS segment) in EBX (or RBX). The format of the structure is given in Table 4-15.

**Table 4-15. MKTME_KEY_PROGRAM_STRUCT Format**

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (bytes)</th>
<th>Size (bytes)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYID</td>
<td>0</td>
<td>2</td>
<td>Key Identifier.</td>
</tr>
</tbody>
</table>
| KEYID_CTRL  | 2              | 4            | KeyID control:  
|             |                |              | • Bits 7:0: key-programming command (COMMAND)  
|             |                |              | • Bits 23:8: encryption algorithm (ENC_ALG)  
|             |                |              | • Bits 31:24: Reserved, must be zero (RSVD) |
| Ignored     | 6              | 58           | Not used. |
| KEY_FIELD_1 | 64             | 64           | Software supplied data key or entropy for data key. |
| KEY_FIELD_2 | 128            | 64           | Software supplied tweak key or entropy for tweak key. |

1. Further details on TME-MK can be found here:  
A description of each of the fields in MKTME_KEY_PROGRAM_STRUCT is provided below:

- **KEYID:** The key identifier (KeyID) being programmed to the MKTME engine. PCONFIG causes a general-protection exception (#GP) if the KeyID is zero. KeyID zero always uses the current behavior configured for TME (total memory encryption), either to encrypt with platform TME key or to bypass TME encryption. PCONFIG also causes a #GP if the KeyID exceeds the maximum enumerated in IA32_TME_CAPABILITY.MK_TME_MAX-_KEYS[bits 50:36] or configured by the setting of IA32_TME_ACTIVATE.MK_TME_KEYID_BITS[bits 35:32].

- **KEYID_CTRL:** The KEYID_CTRL field comprises two sub-fields used by software to control the encryption performed for the selected KeyID:
  - Key-programming command (COMMAND; bits 7:0). This 8-bit field should contain one of the following values:
    - KEYID_SET_KEY_DIRECT (value 0). With this command, software programs directly the encryption key to be used for the selected KeyID.
    - KEYID_SET_KEY_RANDOM (value 1). With this command, software has the CPU generate and assign an encryption key to be used for the selected KeyID using a hardware random-number generator.
      If this command is used and there is insufficient entropy for the random-number generator, PCONFIG will fail and report the failure by loading EAX with value 2 (ENTROPY_ERROR).
      Because the keys programed by PCONFIG are discarded on reset and software cannot read the programmed keys, the keys programmed with this command are ephemeral.
    - KEYID_CLEAR_KEY (value 2). With this command, software indicates that the selected KeyID should use the current behavior configured for TME (see above).
    - KEYID_NO_ENCRYPT (value 3). With this command, software indicates that no encryption should be used for the selected KeyID.
      If any other value is used, PCONFIG causes a #GP.
  - Encryption algorithm (ENC_ALG, bits 23:8). Bits 63:48 of the IA32_TME_ACTIVATE MSR (MSR index 982H) indicate which encryption algorithms are supported by the platform. The 16-bit ENC_ALG field should specify one of the algorithms indicated in IA32_TME_ACTIVATE. PCONFIG causes a #GP if ENC_ALG does not set exactly one bit or if it sets a bit whose corresponding bit is not set in IA32_TME_ACTIVATE[63:48].

- **KEY_FIELD_1:** Use of this field depends upon selected key-programming command:
  - If the direct key-programming command is used (KEYID_SET_KEY_DIRECT), this field carries the software supplied data key to be used for the KeyID.
  - If the random key-programming command is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random data key.
    - This field is ignored when one of the other key-programming commands is used.
    It is software’s responsibility to ensure that the key supplied for the direct key-programming option or the entropy supplied for the random key-programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys.

- **KEY_FIELD_2:** Use of this field depends upon selected key-programming command:
  - If the direct key-programming command is used (KEYID_SET_KEYDIRECT), this field carries the software supplied tweak key to be used for the KeyID.
  - If the random key-programming command is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random tweak key.
    - This field is ignored when one of the other key-programming commands is used.
    It is software’s responsibility to ensure that the key supplied for the direct key-programming option or the entropy supplied for the random key-programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys.

All KeyIDs default to TME behavior (encrypt with TME key or bypass encryption) on activation of TME-MK. Software can at any point decide to change the key for a KeyID using the MKTME_KEY_PROGRAM leaf function of the PCONFIG instruction. Changing the key for a KeyID does not change the state of the TLB caches or memory pipeline. Software is responsible for taking appropriate actions to ensure correct behavior.
The key table used by TME-MK is shared by all logical processors in a platform. For this reason, execution of the MKTME_KEY_PROGRAM leaf function must gain exclusive access to the key table before updating it. The leaf function does this by acquiring lock (implemented in the platform) and retaining that lock until the execution completes. An execution of the leaf function may fail to acquire the lock if it is already in use. In this situation, the leaf function will load EAX with failure reason 5 (DEVICE_BUSY) indicating that software must retry. When this happens, the key table is not updated, and software should retry execution of PCONFIG.

**NOTES**

Earlier versions of this manual specified that bytes 63:6 of MKTME_KEY_PROGRAM_STRUCT were reserved and that PCONFIG would cause a #GP if they were not all zero. This is not the case. As indicated in Table 4-15, PCONFIG ignores those bytes.

They also specified that PCONFIG would cause a #GP if the upper 48 bytes of each of the 64-byte key fields were not all 0. This is not the case. From each of these fields, PCONFIG uses the number of bytes required by the selected encryption algorithm (e.g., 32 bytes for AES-XTS 256) and ignores the upper bytes.

They also specified that PCONFIG would complete and report a failure reason in EAX if the structure specified an incorrect KeyID, and unsupported key-programming command, or an incorrect selection of an encryption algorithm. This is not the case. As indicated above (and in the Operation section), those conditions cause #GP.

**Operation**

(* #UD if PCONFIG is not enumerated or CPL > 0 *)

IF CPUID.7:EDX[18] = 0 OR CPL > 0  
THEN #UD; Ff;

(* #GP(0) for an unsupported leaf function *)

IF EAX != 0  
THEN #GP(0); Ff;

CASE (EAX) (* operation based on selected leaf function *)

0 (MKTME_KEY_PROGRAM):

(* Confirm that TME-MK is properly enabled by the IA32_TME_ACTIVATE MSR *)

(* The MSR must be locked, encryption enabled, and a non-zero number of KeyID bits specified *)

IF IA32_TME_ACTIVATE[0] = 0 OR IA32_TME_ACTIVATE[1] = 0 OR IA32_TME_ACTIVATE[35:32] = 0  
THEN #GP(0); Ff;

IF DS:RBX is not 256-byte aligned  
THEN #GP(0); Ff;

Load TMP_KEY_PROGRAM_STRUCT from 192 bytes at linear address DS:RBX;

IF TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL sets any reserved bits  
THEN #GP(0); Ff;

(* Check for a valid command *)

IF TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND > 3  
THEN #GP(0); Ff;

(* Check that the KEYID being operated upon is a valid KEYID *)

IF TMP_KEY_PROGRAM_STRUCT.KEYID = 0 OR  
   TMP_KEY_PROGRAM_STRUCT.KEYID > 2^IA32_TME_ACTIVATE.MK_TME_KEYID_BITS - 1 OR  
   TMP_KEY_PROGRAM_STRUCT.KEYID > IA32_TME_CAPABILITY.MK_TME_MAX_KEYS  
THEN #GP(0); Ff;
(* Check that only one encryption algorithm is requested for the KeyID and it is one of the activated algorithms *)
IF TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG does not set exactly one bit OR
(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG & IA32_TME_ACTIVATE[63:48]) = 0
THEN #GP(0); FI:

Attempt to acquire lock to gain exclusive access to platform key table;
IF attempt is unsuccessful
THEN (* PCONFIG failure *)
  RFLAGS.ZF := 1;
  RAX := DEVICE_BUSY; (* failure reason 5 *)
  GOTO EXIT;
FI;

CASE (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND) OF
  0 (KEYID_SET_KEY_DIRECT):
    Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
    Encrypt with the selected key
    Use the encryption algorithm selected by TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG
    (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
    DATA_KEY is TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1
    TWEAK_KEY is TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2
    BREAK;
  1 (KEYID_SET_KEY_RANDOM):
    Load TMP_RND_DATA_KEY with a random key using hardware RNG; (* key size depends on selected encryption algorithm *)
    IF there was insufficient entropy
    THEN (* PCONFIG failure *)
      RFLAGS.ZF := 1;
      RAX := ENTROPY_ERROR; (* failure reason 2 *)
      Release lock on platform key table;
      GOTO EXIT;
    FI;
    Load TMP_RND_TWEAK_KEY with a random key using hardware RNG; (* key size depends on selected encryption algorithm *)
    IF there was insufficient entropy
    THEN (* PCONFIG failure *)
      RFLAGS.ZF := 1;
      RAX := ENTROPY_ERROR; (* failure reason 2 *)
      Release lock on platform key table;
      GOTO EXIT;
    FI;
    (* Combine software-supplied entropy to the data key and tweak key *)
    (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
    TMP_RND_DATA_KEY := TMP_RND_KEY XOR TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1;
    TMP_RND_TWEAK_KEY := TMP_RND_TWEAK_KEY XOR TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2;
    Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
    Encrypt with the selected key
    Use the encryption algorithm selected by TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG
    (* The number of bytes used by the next two lines depends on selected encryption algorithm *)
    DATA_KEY is TMP_RND_DATA_KEY
    TWEAK_KEY is TMP_RND_TWEAK_KEY
    BREAK;
  2 (KEYID_CLEAR_KEY):
Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
   Encrypt (or not) using the current configuration for TME
   The specified encryption algorithm and key values are not used.
   BREAK;

3 (KEYID_NO_ENCRYPT):
Update TME-MK table for TMP_KEY_PROGRAM_STRUCT.KEYID as follows:
   Do not encrypt
   The specified encryption algorithm and key values are not used.
   BREAK;
   ESAC;
   Release lock on platform key table;
ESAC;

RAX := 0;
RFLAGS.ZF := 0;

EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

Protected Mode Exceptions
#GP(0) If input value in EAX encodes an unsupported leaf function.
   If a memory operand effective address is outside the relevant segment limit.

MKTME_KEY_PROGRAM leaf function:
   If IA32_TME_ACTIVATE MSR is not locked.
   If hardware encryption and TME-MK capability are not enabled in IA32_TME_ACTIVATE MSR.
   If the memory operand is not 256B aligned.
   If any of the reserved bits in the KEYID_CTRL field of the MKTME_KEY_PROGRAM_STRUCT are
      set or that field indicates an unsupported KeyID, key-programming command, or encryption
      algorithm.

#PF(fault-code) If a page fault occurs in accessing memory operands.
#UD If any of the LOCK/REP/Operand Size/VEX prefixes are used.
   If current privilege level is not 0.
   If CPUID.7.0:EDX[bit 18] = 0

Real-Address Mode Exceptions
#GP If input value in EAX encodes an unsupported leaf function.

MKTME_KEY_PROGRAM leaf function:
   If IA32_TME_ACTIVATE MSR is not locked.
   If hardware encryption and TME-MK capability are not enabled in IA32_TME_ACTIVATE MSR.
   If a memory operand is not 256B aligned.
   If any of the reserved bits in the KEYID_CTRL field of the MKTME_KEY_PROGRAM_STRUCT are
      set or that field indicates an unsupported KeyID, key-programming command, or encryption
      algorithm.

#UD If any of the LOCK/REP/Operand Size/VEX prefixes are used.
   If current privilege level is not 0.
   If CPUID.7.0:EDX.PCONFIG[bit 18] = 0
Virtual-8086 Mode Exceptions

#UD  PCONFIG instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If input value in EAX encodes an unsupported leaf function.
        If a memory operand is non-canonical form.
        **MKTME_KEY_PROGRAM leaf function:**
        If IA32_TME_ACTIVATE MSR is not locked.
        If hardware encryption and TME-MK capability are not enabled in IA32_TME_ACTIVATE MSR.
        If a memory operand is not 256B aligned.
        If any of the reserved bits in the KEYID_CTRL field of the MKTME_KEY_PROGRAM_STRUCT are set or that field indicates an unsupported KeyID, key-programming command, or encryption algorithm.

#PF(fault-code)  If a page fault occurs in accessing memory operands.

#UD  If any of the LOCK/REP/Operand Size/VEX prefixes are used.
        If the current privilege level is not 0.
        If CPUID.7.0:EDX.PCONFIG[bit 18] = 0.
PDEP—Parallel Bits Deposit

**Description**

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

**Figure 4-8. PDEP Example**

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

```
TEMP := SRC1;
MASK := SRC2;
DEST := 0;
m := 0, k := 0;
DO WHILE m < OperandSize
    IF MASK[ m] = 1 THEN
        DEST[ m] := TEMP[ k];
        k := k + 1;
    FI
    m := m + 1;
OD
```
**Flags Affected**
None.

**Intel C/C++ Compiler Intrinsic Equivalent**
PDEP unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);
PDEP unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-29, “Type 13 Class Exception Conditions.”
PEXT—Parallel Bits Extract

**Description**

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

**Figure 4-9. PEXT Example**

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.F3.0F38.W0 F5 /r PEXT r32a, r32b, r/m32</td>
<td>RVM</td>
<td>V/V</td>
<td>BMI2</td>
<td>Parallel extract of bits from r32b using mask in r/m32, result is written to r32a.</td>
</tr>
<tr>
<td>VEX.LZ.F3.0F38.W1 F5 /r PEXT r64a, r64b, r/m64</td>
<td>RVM</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Parallel extract of bits from r64b using mask in r/m64, result is written to r64a.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.
Operation

TEMP := SRC1;
MASK := SRC2;
DEST := 0;
m := 0, k := 0;
DO WHILE m < OperandSize

    IF MASK[ m] = 1 THEN
        DEST[ k] := TEMP[ m];
        k := k + 1;
    FI
m := m + 1;

OD

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PEXT unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);
PEXT unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-29, “Type 13 Class Exception Conditions.”
PEXTRB/PEXTRD/PEXTRQ—Extract Byte/Dword/Qword

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEXTRB reg/m8, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r32 or r64 are zeroed.</td>
</tr>
<tr>
<td>PEXTRD r/m32, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r/m32.</td>
</tr>
<tr>
<td>PEXTRQ r/m64, xmm2, imm8</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE4_1</td>
<td>Extract a qword integer value from xmm2 at the source qword offset specified by imm8 into r/m64.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8</td>
<td>A</td>
<td>V/I</td>
<td>AVX</td>
<td>Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1G 14 /r ib VPEXTRB reg/m8, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1G 16 /r ib VPEXTRD r32/m32, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1G 16 /r ib VPEXTRQ r64/m64, xmm2, imm8</td>
<td>B</td>
<td>V/N.E.</td>
<td>AVX512DQ</td>
<td>Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L”L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.
Operation

CASE of

PEXTRB: SEL := COUNT[3:0];
  TEMP := (Src >> SEL*8) AND FFH;
  IF (DEST = Mem8)
    THEN
      Mem8 := TEMP[7:0];
    ELSE IF (64-Bit Mode and 64-bit register selected)
      THEN
        R64[7:0] := TEMP[7:0];
        r64[63:8] := ZERO_FILL;
      ELSE
        R32[7:0] := TEMP[7:0];
        r32[31:8] := ZERO_FILL;
    FI;

PEXTRD:SEL := COUNT[1:0];
  TEMP := (Src >> SEL*32) AND FFFF_FFFFH;
  DEST := TEMP;

PEXTRQ: SEL := COUNT[0];
  TEMP := (Src >> SEL*64);
  DEST := TEMP;

EASC:

VPEXTRTD/VPEXTRQ
IF (64-Bit Mode and 64-bit dest operand)
  THEN
    Src_Offset := imm8[0]
    r64/m64 := (Src >> Src_Offset * 64)
  ELSE
    Src_Offset := imm8[1:0]
    r32/m32 := ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
  FI

VPEXTR ( dest=m8)
SRC_Offset := imm8[3:0]
Mem8 := (Src >> Src_Offset*8)

VPEXTR ( dest=reg)
IF (64-Bit Mode )
  THEN
    SRC_Offset := imm8[3:0]
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] := ZERO_FILL;
  ELSE
    SRC_Offset := imm8[3:0];
    DEST[7:0] := ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] := ZERO_FILL;
  FI

Intel C/C++ Compiler Intrinsic Equivalent
PEXTRB int _mm_extract_epi8 (__m128i src, const int ndx);
PEXTRD int _mm_extract_epi32 (__m128i src, const int ndx);
PEXTRQ __int64 _mm_extract_epi64 (__m128i src, const int ndx);
Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-22, "Type 5 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-57, "Type E9NF Class Exception Conditions."
Additionally:
#UD  If VEX.L = 1 or EVEX.L’L > 0.
     If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
PEXTRW—Extract Word

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C5 /r ib³</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Extract the word specified by imm8 from mm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.</td>
</tr>
<tr>
<td>PEXTRW reg, mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F C5 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Extract the word specified by imm8 from xmm and move it to reg, bits 15-0. The upper bits of r32 or r64 is zeroed.</td>
</tr>
<tr>
<td>PEXTRW reg, xmm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 3A 15 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Extract the word specified by imm8 from xmm and copy it to lowest 16 bits of reg or m16. Zero-extend the result in the destination, r32 or r64.</td>
</tr>
<tr>
<td>PEXTRW reg/m16, xmm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W0 C5 /r ib</td>
<td>A</td>
<td>V²/V</td>
<td>AVX</td>
<td>Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VPEXTRW reg, xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 15 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VPEXTRW reg/m16, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG C5 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VPEXTRW reg, xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.WIG 15 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.</td>
</tr>
<tr>
<td>VPEXTRW reg/m16, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.
Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

**Operation**

IF (DEST = Mem16)
  THEN
    SEL := COUNT[2:0];
    TEMP := (Src >> SEL*16) AND FFFFH;
    Mem16 := TEMP[15:0];
  ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PSEXTRW instruction with 64-bit source operand)
            { SEL := COUNT[1:0];
              TEMP := (SRC >> (SEL * 16)) AND FFFFH;
              r64[15:0] := TEMP[15:0];
              r64[63:16] := ZERO_FILL; }
        FOR (PSEXTRW instruction with 128-bit source operand)
            { SEL := COUNT[2:0];
              TEMP := (SRC >> (SEL * 16)) AND FFFFH;
              r64[15:0] := TEMP[15:0];
              r64[63:16] := ZERO_FILL; }
    ELSE
        FOR (PSEXTRW instruction with 64-bit source operand)
            { SEL := COUNT[1:0];
              TEMP := (SRC >> (SEL * 16)) AND FFFFH;
              r32[15:0] := TEMP[15:0];
              r32[31:16] := ZERO_FILL; }
        FOR (PSEXTRW instruction with 128-bit source operand)
            { SEL := COUNT[2:0];
              TEMP := (SRC >> (SEL * 16)) AND FFFFH;
              r32[15:0] := TEMP[15:0];
              r32[31:16] := ZERO_FILL; }
  ELSE
        FOR (PSEXTRW instruction with 64-bit source operand)
            { SEL := COUNT[1:0];
              TEMP := (SRC >> (SEL * 16)) AND FFFFH;
              r32[15:0] := TEMP[15:0];
              r32[31:16] := ZERO_FILL; }
        FOR (PSEXTRW instruction with 128-bit source operand)
            { SEL := COUNT[2:0];
              TEMP := (SRC >> (SEL * 16)) AND FFFFH;
              r32[15:0] := TEMP[15:0];
              r32[31:16] := ZERO_FILL; }
  FI;
FI;

VPEXTRW (dest=m16)
SRC_Offset := imm8[2:0]
Mem16 := (Src >> Src_Offset*16)

VPEXTRW (dest=reg)
IF (64-Bit Mode )
  THEN
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] := ZERO_FILL;
  ELSE
    SRC_Offset := imm8[2:0]
    DEST[15:0] := ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] := ZERO_FILL;
  FI
Intel C/C++ Compiler Intrinsic Equivalent
PEXTRW int _mm_extract_pi16 (__m64 a, int n)
PEXTRW int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
Additionally:
#UD If VEX.L = 1 or EVEX.L’L’ > 0.
   If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
**PHADDW/PHADDD—Packed Horizontal Add**

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 01 /r (^1)</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 16-bit integers horizontally, pack to mm1.</td>
</tr>
<tr>
<td>PHADDW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 01 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 16-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>PHADDW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 38 02 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 32-bit integers horizontally, pack to mm1.</td>
</tr>
<tr>
<td>PHADDD mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 02 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Add 32-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>PHADDD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 01 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add 16-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>VPHADDW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 02 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Add 32-bit integers horizontally, pack to xmm1.</td>
</tr>
<tr>
<td>VPHADDD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 01 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add 16-bit signed integers horizontally, pack to ymm1.</td>
</tr>
<tr>
<td>VPHADDW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 02 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Add 32-bit signed integers horizontally, pack to ymm1.</td>
</tr>
<tr>
<td>VPHADDD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Op/En Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.
VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

**Figure 4-10. 256-bit VPHADD Instruction Operation**

**Operation**

**PHADDw (With 64-bit Operands)**
- \( mm1[15:0] = mm1[31:16] + mm1[15:0]; \)
- \( mm1[31:16] = mm1[63:48] + mm1[47:32]; \)
- \( mm1[47:32] = mm2/m64[31:16] + mm2/m64[15:0]; \)
- \( mm1[63:48] = mm2/m64[63:48] + mm2/m64[47:32]; \)

**PHADDw (With 128-bit Operands)**
- \( xmm1[15:0] = xmm1[31:16] + xmm1[15:0]; \)
- \( xmm1[47:32] = xmm1[95:80] + xmm1[79:64]; \)
- \( xmm1[63:48] = xmm1[127:112] + xmm1[111:96]; \)
- \( xmm1[79:64] = xmm2/m128[31:16] + xmm2/m128[15:0]; \)
- \( xmm1[95:80] = xmm2/m128[63:48] + xmm2/m128[47:32]; \)
- \( xmm1[111:96] = xmm2/m128[95:80] + xmm2/m128[79:64]; \)
- \( xmm1[127:112] = xmm2/m128[127:112] + xmm2/m128[111:96]; \)

**VPHADDw (VEX.128 Encoded Version)**
- \( \text{DEST}[15:0] := \text{SRC1}[31:16] + \text{SRC1}[15:0] \)
- \( \text{DEST}[47:32] := \text{SRC1}[95:80] + \text{SRC1}[79:64] \)
- \( \text{DEST}[79:64] := \text{SRC2}[31:16] + \text{SRC2}[15:0] \)
- \( \text{DEST}[111:96] := \text{SRC2}[95:80] + \text{SRC2}[79:64] \)
- \( \text{DEST}[\text{MAXVL-1:128}] := 0 \)
VPHADDW (VEX.256 Encoded Version)

PHADD (With 64-bit Operands)
mm1[31-0] = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];

PHADD (With 128-bit Operands)
xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-64] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];

VPHADD (VEX.128 Encoded Version)
DEST[31-0] := SRC1[63-32] + SRC1[31-0]
DEST[63-32] := SRC1[127-96] + SRC1[95-64]
DEST[95-64] := SRC2[63-32] + SRC2[31-0]
DEST[127-96] := SRC2[127-96] + SRC2[95-64]
DEST[MAXVL-1:128] := 0

VPHADD (VEX.256 Encoded Version)
DEST[31-0] := SRC1[63-32] + SRC1[31-0]
DEST[63-32] := SRC1[127-96] + SRC1[95-64]
DEST[95-64] := SRC2[63-32] + SRC2[31-0]
DEST[127-96] := SRC2[127-96] + SRC2[95-64]
DEST[159-128] := SRC1[191-160] + SRC1[159-128]

Intel C/C++ Compiler Intrinsic Equivalents
PHADDW_m64 _mm_hadd_pi16 (__m64 a, __m64 b)
PHADDW_m64 _mm_hadd_pi32 (__m64 a, __m64 b)
(V)PHADDW_m128i _mm_hadd_epi16 (__m128i a, __m128i b)
(V)PHADDW_m128i _mm_hadd_epi32 (__m128i a, __m128i b)
VPHADDW_m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
VPHADDW_m256i _mm256_hadd_epi32 (__m256i a, __m256i b)
SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions,” additionally:
#UD If VEX.L = 1.
**PHADDSSW—Packed Horizontal Add and Saturate**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 03 /r&lt;sup&gt;1&lt;/sup&gt; PHADDSSW mm1, mm2/m64</td>
<td>RM V/V SSSE3</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to mm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 03 /r PHADDSSW xmm1, xmm2/m128</td>
<td>RM V/V SSSE3</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 03 /r PHADDSSW xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V AVX</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 03 /r PHADDSSW ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V AVX2</td>
<td>Add 16-bit signed integers horizontally, pack saturated integers to ymm1.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM.reg (r, w)</td>
<td>ModRM.r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM.reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM.r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

(V)PHADDSSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

**Operation**

PHADDSSW (With 64-bit Operands)

```plaintext
mm1[15-0] = SaturateToSignedWord(mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```
PHADDW (With 128-bit Operands)

\[
\begin{align*}
\text{xmm1}[15:0] &= \text{SaturateToSignedWord}(\text{xmm1}[31:16] + \text{xmm1}[15:0]); \\
\text{xmm1}[31:16] &= \text{SaturateToSignedWord}(\text{xmm1}[63:48] + \text{xmm1}[47:32]); \\
\text{xmm1}[47:32] &= \text{SaturateToSignedWord}(\text{xmm1}[95:80] + \text{xmm1}[79:64]); \\
\text{xmm1}[63:48] &= \text{SaturateToSignedWord}(\text{xmm1}[127:112] + \text{xmm1}[111:96]); \\
\text{xmm1}[79:64] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[31:16] + \text{xmm2/m128}[15:0]); \\
\text{xmm1}[95:80] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[63:48] + \text{xmm2/m128}[47:32]); \\
\text{xmm1}[111:96] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[95:80] + \text{xmm2/m128}[79:64]); \\
\text{xmm1}[127:112] &= \text{SaturateToSignedWord}(\text{xmm2/m128}[127-112] + \text{xmm2/m128}[111-96]);
\end{align*}
\]

VPHADDW (VEX.128 Encoded Version)

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SaturateToSignedWord}(\text{SRC1}[31:16] + \text{SRC1}[15:0]) \\
\text{DEST}[31:16] &= \text{SaturateToSignedWord}(\text{SRC1}[63:48] + \text{SRC1}[47:32]) \\
\text{DEST}[47:32] &= \text{SaturateToSignedWord}(\text{SRC1}[95:80] + \text{SRC1}[79:64]) \\
\text{DEST}[63:48] &= \text{SaturateToSignedWord}(\text{SRC1}[127:112] + \text{SRC1}[111:96]) \\
\text{DEST}[79:64] &= \text{SaturateToSignedWord}(\text{SRC2}[31:16] + \text{SRC2}[15:0]) \\
\text{DEST}[95:80] &= \text{SaturateToSignedWord}(\text{SRC2}[63:48] + \text{SRC2}[47:32]) \\
\text{DEST}[111:96] &= \text{SaturateToSignedWord}(\text{SRC2}[95:80] + \text{SRC2}[79:64]) \\
\text{DEST}[127:112] &= \text{SaturateToSignedWord}(\text{SRC2}[127:112] + \text{SRC2}[111:96]) \\
\text{DEST}[\text{MAXVL}-1:128] &= 0
\end{align*}
\]

VPHADDW (VEX.256 Encoded Version)

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SaturateToSignedWord}(\text{SRC1}[31:16] + \text{SRC1}[15:0]) \\
\text{DEST}[31:16] &= \text{SaturateToSignedWord}(\text{SRC1}[63:48] + \text{SRC1}[47:32]) \\
\text{DEST}[47:32] &= \text{SaturateToSignedWord}(\text{SRC1}[95:80] + \text{SRC1}[79:64]) \\
\text{DEST}[63:48] &= \text{SaturateToSignedWord}(\text{SRC1}[127:112] + \text{SRC1}[111:96]) \\
\text{DEST}[79:64] &= \text{SaturateToSignedWord}(\text{SRC2}[31:16] + \text{SRC2}[15:0]) \\
\text{DEST}[95:80] &= \text{SaturateToSignedWord}(\text{SRC2}[63:48] + \text{SRC2}[47:32]) \\
\text{DEST}[111:96] &= \text{SaturateToSignedWord}(\text{SRC2}[95:80] + \text{SRC2}[79:64]) \\
\text{DEST}[127:112] &= \text{SaturateToSignedWord}(\text{SRC2}[127:112] + \text{SRC2}[111:96]) \\
\text{DEST}[143:128] &= \text{SaturateToSignedWord}(\text{SRC1}[159:144] + \text{SRC1}[143:128]) \\
\text{DEST}[159:144] &= \text{SaturateToSignedWord}(\text{SRC1}[191:176] + \text{SRC1}[175:160]) \\
\text{DEST}[175:160] &= \text{SaturateToSignedWord}(\text{SRC1}[223:208] + \text{SRC1}[207:192]) \\
\text{DEST}[191:176] &= \text{SaturateToSignedWord}(\text{SRC1}[255:240] + \text{SRC1}[239:224]) \\
\text{DEST}[207:192] &= \text{SaturateToSignedWord}(\text{SRC2}[127:112] + \text{SRC2}[143:128]) \\
\text{DEST}[223:208] &= \text{SaturateToSignedWord}(\text{SRC2}[159:144] + \text{SRC2}[175:160]) \\
\text{DEST}[239:224] &= \text{SaturateToSignedWord}(\text{SRC2}[191-160] + \text{SRC2}[159-128]) \\
\text{DEST}[255:240] &= \text{SaturateToSignedWord}(\text{SRC2}[255:240] + \text{SRC2}[239:224])
\end{align*}
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{PHADDW} \_\_\text{m64} \_\_\text{mm}\_\text{hadds}\_\text{pi16} (\_\_\text{m64} \text{a}, \_\_\text{m64} \text{b}) \\
(V)\text{PHADDW} \_\_\text{m128i} \_\_\text{mm}\_\text{hadds}\_\text{epi16} (\_\_\text{m128i} \text{a}, \_\_\text{m128i} \text{b}) \\
\text{VPHADDW} \_\_\text{m256i} \_\_\text{mm}\_\text{256}\_\text{hadds}\_\text{epi16} (\_\_\text{m256i} \text{a}, \_\_\text{m256i} \text{b})
\end{align*}
\]

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

\#UD If VEX.L = 1.
PHMINPOSUW—Packed Horizontal Word Minimum

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding XMM destination register remain unchanged. VEX.128 encoded version: Bits (MAXVL-1:128) of the destination XMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

**Operation**

**PHMINPOSUW (128-bit Legacy SSE Version)**

INDEX := 0;
MIN := SRC[15:0]
IF (SRC[31:16] < MIN)
   THEN INDEX := 1; MIN := SRC[31:16]; FI;
IF (SRC[47:32] < MIN)
   THEN INDEX := 2; MIN := SRC[47:32]; FI;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
   THEN INDEX := 7; MIN := SRC[127:112]; FI;
DEST[15:0] := MIN;
DEST[18:16] := INDEX;
DEST[127:19] := 0000000000000000000000000000H;

**VPHMINPOSUW (VEX.128 Encoded Version)**

INDEX := 0
MIN := SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX := 1; MIN := SRC[31:16]
* Repeat operation for words 3 through 6
DEST[15:0] := MIN
DEST[18:16] := INDEX
DEST[127:19] := 0000000000000000000000000000H
DEST[MAXVL-1:128] := 0
INSTRUCTION SET REFERENCE, M-U

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW __m128i _mm_minpos_epu16( __m128i packed_words);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD If VEX.L = 1.
   If VEX.vvvv ≠ 1111B.
### PHSUBW/PHSUBD—Packed Horizontal Subtract

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 38 05 /r¹ PHSUBW mm1, mm2/m64</td>
<td>RM V/V</td>
<td>SSSE3</td>
<td>Subtract 16-bit signed integers horizontally, pack to mm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 05 /r PHSUBW xmm1, xmm2/m128</td>
<td>RM V/V</td>
<td>SSSE3</td>
<td>Subtract 16-bit signed integers horizontally, pack to xmm1.</td>
<td></td>
</tr>
<tr>
<td>NP 38 06 /r PHSUBD mm1, mm2/m64</td>
<td>RM V/V</td>
<td>SSSE3</td>
<td>Subtract 32-bit signed integers horizontally, pack to mm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 06 /r PHSUBD xmm1, xmm2/m128</td>
<td>RM V/V</td>
<td>SSSE3</td>
<td>Subtract 32-bit signed integers horizontally, pack to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 05 /r VPHSUBW xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Subtract 16-bit signed integers horizontally, pack to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 06 /r VPHSUBD xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Subtract 32-bit signed integers horizontally, pack to xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 05 /r VPHSUBW ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V</td>
<td>AVX2</td>
<td>Subtract 16-bit signed integers horizontally, pack to ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 06 /r VPHSUBD ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V</td>
<td>AVX2</td>
<td>Subtract 32-bit signed integers horizontally, pack to ymm1.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**


### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.
VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

**Operation**

**PHSUBW (With 64-bit Operands)**

\[
\begin{align*}
\text{mm1}[15:0] &= \text{mm1}[15:0] - \text{mm1}[31:16]; \\
\text{mm1}[31:16] &= \text{mm1}[47:32] - \text{mm1}[63:48]; \\
\text{mm1}[47:32] &= \text{mm2/m64}[15:0] - \text{mm2/m64}[31:16]; \\
\text{mm1}[63:48] &= \text{mm2/m64}[47:32] - \text{mm2/m64}[63:48];
\end{align*}
\]

**PHSUBW (With 128-bit Operands)**

\[
\begin{align*}
\text{xmm1}[15:0] &= \text{xmm1}[15:0] - \text{xmm1}[31:16]; \\
\text{xmm1}[31:16] &= \text{xmm1}[47:32] - \text{xmm1}[63:48]; \\
\text{xmm1}[47:32] &= \text{xmm1}[79:64] - \text{xmm1}[95:80]; \\
\text{xmm1}[63:48] &= \text{xmm1}[111:96] - \text{xmm1}[127:112]; \\
\text{xmm1}[79:64] &= \text{xmm2/m128}[15:0] - \text{xmm2/m128}[31:16]; \\
\text{xmm1}[95:80] &= \text{xmm2/m128}[47:32] - \text{xmm2/m128}[63:48]; \\
\text{xmm1}[111:96] &= \text{xmm2/m128}[79:64] - \text{xmm2/m128}[95:80]; \\
\text{xmm1}[127:112] &= \text{xmm2/m128}[111:96] - \text{xmm2/m128}[127:112];
\end{align*}
\]

**VPHSUBw (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SRC1}[15:0] - \text{SRC1}[31:16] \\
\text{DEST}[31:16] &= \text{SRC1}[47:32] - \text{SRC1}[63:48] \\
\text{DEST}[47:32] &= \text{SRC1}[79:64] - \text{SRC1}[95:80] \\
\text{DEST}[63:48] &= \text{SRC1}[111:96] - \text{SRC1}[127:112] \\
\text{DEST}[79:64] &= \text{SRC2}[15:0] - \text{SRC2}[31:16] \\
\text{DEST}[111:96] &= \text{SRC2}[79:64] - \text{SRC2}[95:80] \\
\text{DEST}[\text{MAXVL-1:128}] &= 0
\end{align*}
\]

**VPHSUBw (VEX.256 Encoded Version)**

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SRC1}[15:0] - \text{SRC1}[31:16] \\
\text{DEST}[31:16] &= \text{SRC1}[47:32] - \text{SRC1}[63:48] \\
\text{DEST}[47:32] &= \text{SRC1}[79:64] - \text{SRC1}[95:80] \\
\text{DEST}[63:48] &= \text{SRC1}[111:96] - \text{SRC1}[127:112] \\
\text{DEST}[79:64] &= \text{SRC2}[15:0] - \text{SRC2}[31:16] \\
\text{DEST}[111:96] &= \text{SRC2}[79:64] - \text{SRC2}[95:80] \\
\text{DEST}[143:128] &= \text{SRC1}[143:128] - \text{SRC1}[159:144] \\
\text{DEST}[159:144] &= \text{SRC1}[175:160] - \text{SRC1}[191:176] \\
\text{DEST}[175:160] &= \text{SRC1}[207:192] - \text{SRC1}[223:208] \\
\text{DEST}[191:176] &= \text{SRC1}[239:224] - \text{SRC1}[255:240] \\
\text{DEST}[207:192] &= \text{SRC2}[143:128] - \text{SRC2}[159:144] \\
\text{DEST}[223:208] &= \text{SRC2}[175:160] - \text{SRC2}[191:176] \\
\text{DEST}[239:224] &= \text{SRC2}[207:192] - \text{SRC2}[223:208] \\
\text{DEST}[255:240] &= \text{SRC2}[239:224] - \text{SRC2}[255:240]
\end{align*}
\]

**PHSUBD (With 64-bit Operands)**

\[
\begin{align*}
\text{mm1}[31:0] &= \text{mm1}[31:0] - \text{mm1}[63:32]; \\
\text{mm1}[63:32] &= \text{mm2/m64}[31:0] - \text{mm2/m64}[63:32];
\end{align*}
\]
**PHSUBD (With 128-bit Operands)**

- `xmm1[31-0] = xmm1[31-0] - xmm1[63-32];`
- `xmm1[63-32] = xmm1[95-64] - xmm1[127-96];`
- `xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];`
- `xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];`

**VPHSUBD (VEX.128 Encoded Version)**

- `DEST[31-0] := SRC1[31-0] - SRC1[63-32]`
- `DEST[95-64] := SRC2[31-0] - SRC2[63-32]`
- `DEST[MAXVL-1:128] := 0`

**VPHSUBD (VEX.256 Encoded Version)**


**Intel C/C++ Compiler Intrinsic Equivalents**

- `PHSUBW __m64 _mm_hsub_pi16 (__m64 a, __m64 b)`
- `PHSUBD __m64 _mm_hsub_pi32 (__m64 a, __m64 b)`
- `(V)PHSUBW __m128i _mm_hsub_epi16 (__m128i a, __m128i b)`
- `(V)PHSUBD __m128i _mm_hsub_epi32 (__m128i a, __m128i b)`
- `VPHSUBW __m256i _mm256_hsub_epi16 (__m256i a, __m256i b)`
- `VPHSUBD __m256i _mm256_hsub_epi32 (__m256i a, __m256i b)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, “Type 4 Class Exception Conditions,” additionally:

#UD If VEX.L = 1.
PHSUBSW—Packed Horizontal Subtract and Saturate

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 07 \r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Subtract 16-bit signed integer horizontally, pack saturated integers to mm1.</td>
</tr>
<tr>
<td>PHSUBSW mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 07 \r</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.</td>
</tr>
<tr>
<td>PHSUBSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1G 07 \r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract 16-bit signed integer horizontally, pack saturated integers to xmm1.</td>
</tr>
<tr>
<td>VPHSUBSW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1G 07 \r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract 16-bit signed integer horizontally, pack saturated integers to ymm1.</td>
</tr>
<tr>
<td>VPHSUBSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL:1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL:1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Operation

PHSUBSW (with 64-bit Operands)

\[
\begin{align*}
\text{mm1}[15-0] & = \text{SaturateToSignedWord}\left(\text{mm1}[15-0] - \text{mm1}[31-16]\right); \\
\text{mm1}[31-16] & = \text{SaturateToSignedWord}\left(\text{mm1}[47-32] - \text{mm1}[63-48]\right); \\
\text{mm1}[47-32] & = \text{SaturateToSignedWord}\left(\text{mm2/m64}[15-0] - \text{mm2/m64}[31-16]\right); \\
\text{mm1}[63-48] & = \text{SaturateToSignedWord}\left(\text{mm2/m64}[47-32] - \text{mm2/m64}[63-48]\right); \\
\end{align*}
\]
**PHSUBSW (With 128-bit Operands)**

- xmm1[15-0] = SaturateToSignedWord(xmm1[15-0] - xmm1[31-16]);
- xmm1[47-32] = SaturateToSignedWord(xmm1[79-64] - xmm1[95-80]);
- xmm1[79-64] = SaturateToSignedWord(xmm2/m128[15-0] - xmm2/m128[31-16]);
- xmm1[95-80] = SaturateToSignedWord(xmm2/m128[47-32] - xmm2/m128[63-48]);
- xmm1[111-96] = SaturateToSignedWord(xmm2/m128[79-64] - xmm2/m128[95-80]);

**VPHSUBSW (VEX.128 Encoded Version)**

- DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
- DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
- DEST[MAXVL-1:128] := 0

**VPHSUBSW (VEX.256 Encoded Version)**

- DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
- DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])

**Intel C/C++ Compiler Intrinsic Equivalent**

PHSUBSW __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
(V)PHSUBSW __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
VPHSUBSW __m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD If VEX.L = 1.
PINSRB/PINSRD/PINSRQ—Insert Byte/Dword/Qword

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 20 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINSRB xmm1, r32/m8, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Insert a byte integer value from r32/m8 into xmm1 at the destination element in xmm1 specified by imm8.</td>
</tr>
<tr>
<td>66 0F 3A 22 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINSRD xmm1, r/m32, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Insert a dword integer value from r/m32 into the xmm1 at the destination element specified by imm8.</td>
</tr>
<tr>
<td>66 REX.W 0F 3A 22 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PINSRQ xmm1, r/m64, imm8</td>
<td>A</td>
<td>V/N.E.</td>
<td>SSE4_1</td>
<td>Insert a qword integer value from r/m64 into the xmm1 at the destination element specified by imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 20 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPINSRB xmm1, xmm2, r32/m8, imm8</td>
<td>B</td>
<td>V^1/V</td>
<td>AVX</td>
<td>Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 22 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPINSRD xmm1, xmm2, r/m32, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W1 22 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPINSRQ xmm1, xmm2, r/m64, imm8</td>
<td>B</td>
<td>V/Ic</td>
<td>AVX</td>
<td>Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0G 20 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPINSRB xmm1, xmm2, r32/m8, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 G 22 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPINSRD xmm1, xmm2, r/m32, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 22 /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPINSRQ xmm1, xmm2, r64/m64, imm8</td>
<td>C</td>
<td>V/N.E.c</td>
<td>AVX512DQ</td>
<td>Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/r (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/r (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/r (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description
Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.
In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L’L must be 0, otherwise the instruction will #UD.

**Operation**

**CASE OF**

PINSRB: SEL := COUNT[3:0];
MASK := (0FFH << (SEL * 8));
TEMP := (((SRC[7:0] << (SEL * 8)) AND MASK);
PINSRD: SEL := COUNT[1:0];
MASK := (0FFFFFFFFH << (SEL * 32));
TEMP := (((SRC << (SEL * 32)) AND MASK) ;
PINSRQ: SEL := COUNT[0]
MASK := (0FFFFFFFFFFFFFFFFH << (SEL * 64));
TEMP := (((SRC << (SEL *64)) AND MASK) ;
ESAC:
DEST := ((DEST AND NOT MASK) OR TEMP);

**VPINSRB (VEX/EVEX Encoded Version)**

SEL := imm8[3:0]
DEST[127:0] := write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0

**VPINSRD (VEX/EVEX Encoded Version)**

SEL := imm8[1:0]
DEST[127:0] := write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0

**VPINSRQ (VEX/EVEX Encoded Version)**

SEL := imm8[0]
DEST[127:0] := write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

PINSRB __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.
Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
Additionally:

#UD If VEX.L = 1 or EVEX.L'L > 0.
PINSRW—Insert Word

**Opcode/ Instruction**

<table>
<thead>
<tr>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C4 /r:ib†</td>
<td>V/V</td>
<td>SSE</td>
<td>Insert the low word from r32 or from m16 into mm at the word position specified by imm8.</td>
</tr>
<tr>
<td>66 0F C4 /r ib</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move the low word of r32 or from m16 into xmm at the word position specified by imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F.w0 C4 /r ib</td>
<td>V^2/V</td>
<td>AVX</td>
<td>Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG C4 /r ib</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Insert the word from r32/m16 at the offset indicated by imm8 into the value from xmm2 and store result in xmm1.</td>
</tr>
</tbody>
</table>

**NOTES:**
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Three operand MMX and SSE instructions:

Copies a word from the source operand and inserts it in the destination operand at the location specified with the count operand. (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Four operand AVX and AVX-512 instructions:

Combines a word from the first source operand with the second source operand, and inserts it in the destination operand at the location specified with the count operand. The second source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The first source and destination operands are XMM registers. The count operand is an 8-bit immediate. When specifying a word location, the 3 least-significant bits specify the location.

Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L/EVEX.L’L must be 0, otherwise the instruction will #UD.
Operation

PINSRW dest, src, imm8 (MMX)
   SEL := imm8[1:0]
   DEST.word[SEL] := src.word[0]

PINSRW dest, src, imm8 (SSE)
   SEL := imm8[2:0]
   DEST.word[SEL] := src.word[0]

VPINSRW dest, src1, src2, imm8 (AVX/AVX512)
   SEL := imm8[2:0]
   DEST := src1
   DEST.word[SEL] := src2.word[0]
   DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW __m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW __m128i _mm_insert_epi16 ( __m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-57, “Type E9NF Class Exception Conditions.”
Additionally:

#UD If VEX.L = 1 or EVEX.L'L > 0.
**PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 04 /r¹ PMADDUBSW mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to mm1.</td>
</tr>
<tr>
<td>66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 04 /r PMADDUBSW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 04 /r PMADDUBSW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 04 /r PMADDUBSW xmm1 (k1){z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 04 /r PMADDUBSW ymm1 (k1){z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 04 /r PMADDUBSW zmm1 (k1){z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.
128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

**PMADDUBSW (With 64-bit Operands)**

\[
\begin{align*}
\text{DEST}[15-0] &= \text{SaturateToSignedWord}(\text{SRC}[15-8]*\text{DEST}[15-8]+\text{SRC}[7-0]*\text{DEST}[7-0]); \\
\text{DEST}[31-16] &= \text{SaturateToSignedWord}(\text{SRC}[31-24]*\text{DEST}[31-24]+\text{SRC}[23-16]*\text{DEST}[23-16]); \\
\text{DEST}[47-32] &= \text{SaturateToSignedWord}(\text{SRC}[47-40]*\text{DEST}[47-40]+\text{SRC}[39-32]*\text{DEST}[39-32]); \\
\text{DEST}[63-48] &= \text{SaturateToSignedWord}(\text{SRC}[63-56]*\text{DEST}[63-56]+\text{SRC}[55-48]*\text{DEST}[55-48]);
\end{align*}
\]

**PMADDUBSW (With 128-bit Operands)**

\[
\begin{align*}
\text{DEST}[15-0] &= \text{SaturateToSignedWord}(\text{SRC}[15-8]*\text{DEST}[15-8]+\text{SRC}[7-0]*\text{DEST}[7-0]); \\
&\text{ // Repeat operation for 2nd through 7th word} \\
\text{SRC1}/\text{DEST}[127-112] &= \text{SaturateToSignedWord}(\text{SRC}[127-120]*\text{DEST}[127-120]+\text{SRC}[119-112]*\text{DEST}[119-112]);
\end{align*}
\]

**VPMADDUBSW (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{DEST}[15:0] := \text{SaturateToSignedWord}(\text{SRC2}[15:8]* \text{SRC1}[15:8]+\text{SRC2}[7:0]*\text{SRC1}[7:0]) \\
&\text{ // Repeat operation for 2nd through 7th word} \\
\text{DEST}[127:112] := \text{SaturateToSignedWord}(\text{SRC2}[127:120]*\text{SRC1}[127:120]+\text{SRC2}[119:112]*\text{SRC1}[119:112]) \\
\text{DEST}[\text{MAXVL}-1:128] := 0
\end{align*}
\]

**VPMADDUBSW (VEX.256 Encoded Version)**

\[
\begin{align*}
\text{DEST}[15:0] := \text{SaturateToSignedWord}(\text{SRC2}[15:8]* \text{SRC1}[15:8]+\text{SRC2}[7:0]*\text{SRC1}[7:0]) \\
&\text{ // Repeat operation for 2nd through 15th word} \\
\text{DEST}[\text{MAXVL}-1:256] := 0
\end{align*}
\]

**VPMADDUBSW (EVEX Encoded Versions)**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[
\begin{align*}
\text{FOR} j := 0 \text{ TO } KL-1 \\
i := j * 16 \\
\text{IF k1[j]} \text{ OR } \text{ *no writemask*} \\
&\text{ THEN } \text{DEST}[i+15:i] := \text{SaturateToSignedWord}(\text{SRC2}[i+15:i+8]* \text{SRC1}[i+15:i+8]+\text{SRC2}[i+7:i]*\text{SRC1}[i+7:i]) \\
&\text{ ELSE} \\
&\text{ *merging-masking* } ; \text{merging-masking} \\
&\text{ THEN } \text{*DEST}[i+15:i] \text{ remains unchanged*} \\
&\text{ ELSE } \text{*zeroing-masking* } ; \text{zeroing-masking} \\
&\text{DEST}[i+15:i] = 0 \\
&\text{ FI} \\
&\text{ FI} \\
\text{ENDFOR;}
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]
Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW __m512i _mm512_maddubs_epi16( __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_maddubs_epi16( __m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_maddubs_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_maddubs_epi16( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_maddubs_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_maddubs_epi16( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16( __mmask8 k, __m128i a, __m128i b);
PMADDUBSW __m64 _mm_maddubs_pi16( __m64 a, __m64 b)
(V)PMADDUBSW __m128i _mm_maddubs_epi16( __m128i a, __m128i b)
VPMADDUBSW __m256i _mm256_maddubs_epi16( __m256i a, __m256i b)

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
## PMADDWD—Multiply and Add Packed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 6F 75 /r^1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Multiply the packed words in mm by the packed words in mm/m64, add adjacent doubleword results, and store in mm.</td>
</tr>
<tr>
<td>PMADDWD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 6F 75 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.</td>
</tr>
<tr>
<td>PMADDWD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 6F 75 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.</td>
</tr>
<tr>
<td>VPMADDWD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 6F 75 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1.</td>
</tr>
<tr>
<td>VPMADDWD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 6F 75 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMADDWD xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG 6F 75 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMADDWD ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG 6F 75 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMADDWD zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).
The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

**Operation**

**PMADDWD (With 64-bit Operands)**

\[
\begin{align*}
\text{DEST}[31:0] & := (\text{DEST}[15:0] \ast \text{SRC}[15:0]) + (\text{DEST}[31:16] \ast \text{SRC}[31:16]) \\
\text{DEST}[63:32] & := (\text{DEST}[47:32] \ast \text{SRC}[47:32]) + (\text{DEST}[63:48] \ast \text{SRC}[63:48])
\end{align*}
\]

**PMADDWD (With 128-bit Operands)**

\[
\begin{align*}
\text{DEST}[31:0] & := (\text{DEST}[15:0] \ast \text{SRC}[15:0]) + (\text{DEST}[31:16] \ast \text{SRC}[31:16]) \\
\text{DEST}[63:32] & := (\text{DEST}[47:32] \ast \text{SRC}[47:32]) + (\text{DEST}[63:48] \ast \text{SRC}[63:48]) \\
\text{DEST}[95:64] & := (\text{DEST}[79:64] \ast \text{SRC}[79:64]) + (\text{DEST}[95:80] \ast \text{SRC}[95:80]) \\
\text{DEST}[127:96] & := (\text{DEST}[111:96] \ast \text{SRC}[111:96]) + (\text{DEST}[127:112] \ast \text{SRC}[127:112]) \\
\text{DEST}[\text{MAXVL}-1:128] & := 0
\end{align*}
\]

**VPMADDWD (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{DEST}[31:0] & := (\text{SRC1}[15:0] \ast \text{SRC2}[15:0]) + (\text{SRC1}[31:16] \ast \text{SRC2}[31:16]) \\
\text{DEST}[63:32] & := (\text{SRC1}[47:32] \ast \text{SRC2}[47:32]) + (\text{SRC1}[63:48] \ast \text{SRC2}[63:48]) \\
\text{DEST}[95:64] & := (\text{SRC1}[79:64] \ast \text{SRC2}[79:64]) + (\text{SRC1}[95:80] \ast \text{SRC2}[95:80]) \\
\text{DEST}[127:96] & := (\text{SRC1}[111:96] \ast \text{SRC2}[111:96]) + (\text{SRC1}[127:112] \ast \text{SRC2}[127:112]) \\
\text{DEST}[\text{MAXVL}-1:128] & := 0
\end{align*}
\]
INSTRUCTION SET REFERENCE, M-U

VPMADDWD (VEX.256 Encoded Version)
DEST[31:0] := (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[95:64] := (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[MAXVL-1:256] := 0

VPMADDWD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := (SRC2[i+31:i+16] * SRC1[i+31:i+16]) + (SRC2[i+15:i] * SRC1[i+15:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] = 0
    FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMADDWD __m512i _mm512_madd_epi16( __m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_madd_epi16( __m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_madd_epi16( __mmask32 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_madd_epi16( __m256i a, __m256i b);
VPMADDWD __m256i _mm256_mask_madd_epi16( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_madd_epi16( __mmask16 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm128_mask_madd_epi16( __m128i s, __mmask8 k, __m128i a, __m128i b);
PMADDWD __m64 __m64_madd_pi16( __m64 m1, __m64 m2);
(VPMADDWD __m128i _mm128_mask_madd_epi16 ( __m128i a, __m128i b)
VPMADDWD __m512i _mm512_madd_epi16 ( __m512i a, __m512i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
### PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EE /r&lt;sup&gt;1&lt;/sup&gt; PMAXSW mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare signed word integers in mm2/m64 and mm1 and return maximum values.</td>
</tr>
<tr>
<td>66 0F 38 3C /r PMAXSB xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE&lt;sub&gt;4_1&lt;/sub&gt;</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>66 0F EE /r PMAXSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.</td>
</tr>
<tr>
<td>66 0F 38 3D /r PMAXSD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE&lt;sub&gt;4_1&lt;/sub&gt;</td>
<td>Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 3C /r VPMAXSW xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;512VL&lt;/sub&gt; AVX&lt;sub&gt;512BW&lt;/sub&gt;</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 3C /r VPMAXSW ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;512VL&lt;/sub&gt; AVX&lt;sub&gt;512BW&lt;/sub&gt;</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 3C /r VPMAXSW zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;512BW&lt;/sub&gt;</td>
<td>Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 3E /r VPMAXSW xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;512VL&lt;/sub&gt; AVX&lt;sub&gt;512BW&lt;/sub&gt;</td>
<td>Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 3E /r VPMAXSW ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;512VL&lt;/sub&gt; AVX&lt;sub&gt;512BW&lt;/sub&gt;</td>
<td>Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 3E /r VPMAXSW zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX&lt;sub&gt;512BW&lt;/sub&gt;</td>
<td>Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>
**PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

VEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

VEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
Operation

PMAXSw (64-bit Operands)

IF DEST[15:0] > SRC[15:0] THEN
  DEST[15:0] := DEST[15:0];
ELSE
  DEST[15:0] := SRC[15:0]; Fi;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
  DEST[63:48] := DEST[63:48];
ELSE
  DEST[63:48] := SRC[63:48]; Fi;

PMAXSB (128-bit Legacy SSE Version)

IF DEST[7:0] > SRC[7:0] THEN
  DEST[7:0] := DEST[7:0];
ELSE
  DEST[7:0] := SRC[7:0]; Fi;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
  DEST[127:120] := DEST[127:120];
ELSE
  DEST[127:120] := SRC[127:120]; Fi;

DEST[MAXVL-1:128] := 0

VPMAXSB (VEX.128 Encoded Version)

IF SRC1[7:0] > SRC2[7:0] THEN
  DEST[7:0] := SRC1[7:0];
ELSE
  DEST[7:0] := SRC2[7:0]; Fi;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
  DEST[127:120] := SRC1[127:120];
ELSE
  DEST[127:120] := SRC2[127:120]; Fi;
DEST[MAXVL-1:128] := 0

VPMAXSB (VEX.256 Encoded Version)

IF SRC1[7:0] > SRC2[7:0] THEN
  DEST[7:0] := SRC1[7:0];
ELSE
  DEST[7:0] := SRC2[7:0]; Fi;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
  DEST[255:248] := SRC1[255:248];
ELSE
  DEST[255:248] := SRC2[255:248]; Fi;
DEST[MAXVL-1:256] := 0
VPMAXSB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask* THEN
    IF SRC1[i+7:i] > SRC2[i+7:i]
      THEN DEST[i+7:i] := SRC1[i+7:i];
      ELSE DEST[i+7:i] := SRC2[i+7:i];
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+7:i] := 0
      FI
    FI;
  ENDFOR;
  DEST[MAXVL-1:VL] := 0

PMAXSw (128-bit Legacy SSE Version)
IF DEST[15:0] > SRC[15:0] THEN
  DEST[15:0] := DEST[15:0];
ELSE
  DEST[15:0] := SRC[15:0];
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:112] := DEST[127:112];
ELSE
  DEST[127:112] := SRC[127:112];
DEST[MAXVL-1:128] (Unmodified)

VPMAXSw (VEX.128 Encoded Version)
IF SRC1[15:0] > SRC2[15:0] THEN
  DEST[15:0] := SRC1[15:0];
ELSE
  DEST[15:0] := SRC2[15:0];
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:112] := SRC1[127:112];
ELSE
  DEST[127:112] := SRC2[127:112];
DEST[MAXVL-1:128] := 0

VPMAXSw (VEX.256 Encoded Version)
IF SRC1[15:0] > SRC2[15:0] THEN
  DEST[15:0] := SRC1[15:0];
ELSE
  DEST[15:0] := SRC2[15:0];
(* Repeat operation for 2nd through 15th words in source and destination operands *)
  DEST[255:240] := SRC1[255:240];
ELSE
  DEST[255:240] := SRC2[255:240];
DEST[MAXVL-1:256] := 0
VPMAXSw (EVEX Encoded Versions)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \( j := 0 \) TO KL-1
\[
i := j \cdot 16
\]
IF \( k1[j] \) OR *no writemask* THEN
IF \( SRC1[i+15:i] > SRC2[i+15:i] \)
THEN \( DEST[i+15:i] := SRC1[i+15:i] \);
ELSE \( DEST[i+15:i] := SRC2[i+15:i] \);
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+15:i] remains unchanged*
ELSE ; zeroing-masking
\( DEST[i+15:i] := 0 \)
FI
ENDFOR;
\( DEST[MAXVL-1:VL] := 0 \)

PMAXSD (128-bit Legacy SSE Version)

IF \( DEST[31:0] > SRC[31:0] \)
THEN \( DEST[31:0] := DEST[31:0] \);
ELSE
\( DEST[31:0] := SRC[31:0] \);
(Exception repeat operation for 2nd through 7th words in source and destination operands *)
IF \( DEST[127:96] > SRC[127:96] \)
ELSE
\( DEST[MAXVL-1:128] \) (Unmodified)

VPMAXSD (VEX.128 Encoded Version)

IF \( SRC1[31:0] > SRC2[31:0] \)
THEN \( DEST[31:0] := SRC1[31:0] \);
ELSE
\( DEST[31:0] := SRC2[31:0] \);
(Exception repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF \( SRC1[127:96] > SRC2[127:96] \)
ELSE
\( DEST[MAXVL-1:128] := 0 \)

VPMAXSD (VEX.256 Encoded Version)

IF \( SRC1[31:0] > SRC2[31:0] \)
THEN \( DEST[31:0] := SRC1[31:0] \);
ELSE
\( DEST[31:0] := SRC2[31:0] \);
(Exception repeat operation for 2nd through 7th dwords in source and destination operands *)
IF \( SRC1[255:224] > SRC2[255:224] \)
ELSE
\( DEST[MAXVL-1:256] := 0 \)
VPMAXSD (EVEX Encoded Versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                        ELSE DEST[i+31:i] := SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] := SRC1[i+31:i];
                        ELSE DEST[i+31:i] := SRC2[i+31:i];
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                        ELSE DEST[i+31:i] := 0 ; zeroing-masking
                FI
            FI;
        ENDFOR
    DEST[MAXVL-1:VL] := 0

VPMAXSQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                        ELSE DEST[i+63:i] := SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] := SRC1[i+63:i];
                        ELSE DEST[i+63:i] := SRC2[i+63:i];
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                        ELSE DEST[i+63:i] := 0 ; zeroing-masking
                FI
            FI;
        ENDFOR;
    DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8(__m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b);

VPMAXSW __m512i _mm512_max_epi16(__m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b);

VPMAXSW __m256i _mm256_max_epi16(__m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);

VPMAXSW __m128i _mm_mask_max_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi8(__mmask8 k, __m128i a, __m128i b);

VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);

VPMAXSD __m256i _mm256_max_epi32(__m256i a, __m256i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);

VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPMAXSD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
## PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F DE /r1</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Compare unsigned byte integers in mm2/m64 and mm1 and returns maximum values.</td>
</tr>
<tr>
<td>PMAXUB mm1, mm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F DE /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>PMAXUB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 3E/r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.</td>
</tr>
<tr>
<td>PMAXUW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F DE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VPMAXUB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38 3E/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned word integers in xmm2 and xmm3/m256 and store packed maximum values in xmm1.</td>
</tr>
<tr>
<td>VPMAXUW xmm1, xmm2, xmm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F DE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed unsigned byte integers in ymm2 and ymm3/m128 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VPMAXUB ymm1, ymm2, ymm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38 3E/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
</tr>
<tr>
<td>VPMAXUW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG DE /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXUB xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG DE /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXUB ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG DE /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXUB zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 3E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXUW xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 3E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXUW ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 3E /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMAXUW zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:

Description
Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be a YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUB (64-bit Operands)

\[
\text{IF } \text{DEST}[7:0] > \text{SRC}[17:0] \text{ THEN}\n\]
\[
\text{DEST}[7:0] := \text{DEST}[7:0];\n\]
\[
\text{ELSE}\n\]
\[
\text{DEST}[7:0] := \text{SRC}[7:0]; \text{ FI};\n\]
\[
\text{(* Repeat operation for 2nd through 7th bytes in source and destination operands *)}\n\]
\[
\text{IF } \text{DEST}[63:56] > \text{SRC}[63:56] \text{ THEN}\n\]
\[
\text{DEST}[63:56] := \text{DEST}[63:56];\n\]
\[
\text{ELSE}\n\]
\[
\text{DEST}[63:56] := \text{SRC}[63:56]; \text{ FI};\n\]

PMAXUB (128-bit Legacy SSE Version)

\[
\text{IF } \text{DEST}[7:0] > \text{SRC}[7:0] \text{ THEN}\n\]
\[
\text{DEST}[7:0] := \text{DEST}[7:0];\n\]
\[
\text{ELSE}\n\]
\[
\text{DEST}[15:0] := \text{SRC}[7:0]; \text{ FI};\n\]
\[
\text{(* Repeat operation for 2nd through 15th bytes in source and destination operands *)}\n\]
\[
\text{IF } \text{DEST}[127:120] > \text{SRC}[127:120] \text{ THEN}\n\]
\[
\text{DEST}[127:120] := \text{DEST}[127:120];\n\]
\[
\text{ELSE}\n\]
\[
\text{DEST}[127:120] := \text{SRC}[127:120]; \text{ FI};\n\]
\[
\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}\n\]
PMAXUB (VEX.128 Encoded Version)

IF SRC1[7:0] > SRC2[7:0] THEN
   DEST[7:0] := SRC1[7:0];
ELSE
   DEST[7:0] := SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
   DEST[127:120] := SRC1[127:120];
ELSE
   DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0

PMAXUB (VEX.256 Encoded Version)

IF SRC1[7:0] > SRC2[7:0] THEN
   DEST[7:0] := SRC1[7:0];
ELSE
   DEST[15:0] := SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
   DEST[255:248] := SRC1[255:248];
ELSE
   DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:128] := 0

PMAXUB (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
   i := j * 8
   IF k1[j] OR *no writemask* THEN
      IF SRC1[i+7:i] > SRC2[i+7:i]
         THEN DEST[i+7:i] := SRC1[i+7:i];
      ELSE DEST[i+7:i] := SRC2[i+7:i];
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+7:i] := 0
      FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

PMAXUW (128-bit Legacy SSE Version)

IF DEST[15:0] > SRC[15:0] THEN
   DEST[15:0] := DEST[15:0];
ELSE
   DEST[15:0] := SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
   DEST[127:112] := DEST[127:112];
ELSE
   DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
VPMAXUW (VEX.128 Encoded Version)
  IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] := SRC1[15:0];
  ELSE
    DEST[15:0] := SRC2[15:0]; FI;

  (* Repeat operation for 2nd through 7th words in source and destination operands *)
    DEST[127:112] := SRC1[127:112];
  ELSE
    DEST[127:112] := SRC2[127:112]; FI;

  DEST[MAXVL-1:128] := 0

VPMAXUW (VEX.256 Encoded Version)
  IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] := SRC1[15:0];
  ELSE
    DEST[15:0] := SRC2[15:0]; FI;

  (* Repeat operation for 2nd through 15th words in source and destination operands *)
    DEST[255:240] := SRC1[255:240];
  ELSE
    DEST[255:240] := SRC2[255:240]; FI;

  DEST[MAXVL-1:128] := 0

VPMAXUW (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask* THEN
    IF SRC1[i+15:i] > SRC2[i+15:i]
      THEN DEST[i+15:i] := SRC1[i+15:i];
      ELSE DEST[i+15:i] := SRC2[i+15:i];
      FI;
    ELSE
      IF *merging-masking* THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+15:i] := 0
        FI
      FI;
  ENDFOR;

  DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUB __m512i _mm512_max_epu8(__m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8(__mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16(__m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16(__mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8(__mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_pu8(__m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16(__m128i a, __m128i b);
VPMAXUB __m256i _mm256_max_epu8(__m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16(__m256i a, __m256i b);
PMAXUB __m64 _mm_max_pu8(__m64 a, __m64 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
## PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 3F /r</td>
<td>SSE4_1</td>
<td>Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 3F /r</td>
<td>AVX</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 3F /r</td>
<td>AVX2</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 3F /r</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 3F /r</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 3F /r</td>
<td>AVX512F</td>
<td>Compare packed unsigned dword integers in zmm2 and zmm3/m12/32bcst and store packed maximum values in zmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 3F /r</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 3F /r</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 3F /r</td>
<td>AVX512F</td>
<td>Compare packed unsigned qword integers in zmm2 and zmm3/m122/m64bcst and store packed maximum values in zmm1 under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.
EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**PMAXUD (128-bit Legacy SSE Version)**

\[
\begin{align*}
\text{IF } \text{DEST}[31:0] > \text{SRC}[31:0] \text{ THEN} \\
\text{DEST}[31:0] := \text{DEST}[31:0]; \\
\text{ELSE} \\
\text{DEST}[31:0] := \text{SRC}[31:0]; \text{ Fl;}
\end{align*}
\]

(* Repeat operation for 2nd through 7th words in source and destination operands *)

\[
\begin{align*}
\text{IF } \text{DEST}[127:96] > \text{SRC}[127:96] \text{ THEN} \\
\text{DEST}[127:96] := \text{DEST}[127:96]; \\
\text{ELSE} \\
\text{DEST}[127:96] := \text{SRC}[127:96]; \text{ Fl;}
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:128] (\text{Unmodified})
\]

**VPMAXUD (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{IF } \text{SRC1}[31:0] > \text{SRC2}[31:0] \text{ THEN} \\
\text{DEST}[31:0] := \text{SRC1}[31:0]; \\
\text{ELSE} \\
\text{DEST}[31:0] := \text{SRC2}[31:0]; \text{ Fl;}
\end{align*}
\]

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

\[
\begin{align*}
\text{IF } \text{SRC1}[127:96] > \text{SRC2}[127:96] \text{ THEN} \\
\text{DEST}[127:96] := \text{SRC1}[127:96]; \\
\text{ELSE} \\
\text{DEST}[127:96] := \text{SRC2}[127:96]; \text{ Fl;}
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

**VPMAXUD (VEX.256 Encoded Version)**

\[
\begin{align*}
\text{IF } \text{SRC1}[31:0] > \text{SRC2}[31:0] \text{ THEN} \\
\text{DEST}[31:0] := \text{SRC1}[31:0]; \\
\text{ELSE} \\
\text{DEST}[31:0] := \text{SRC2}[31:0]; \text{ Fl;}
\end{align*}
\]

(* Repeat operation for 2nd through 7th dwords in source and destination operands *)

\[
\begin{align*}
\text{IF } \text{SRC1}[255:224] > \text{SRC2}[255:224] \text{ THEN} \\
\text{DEST}[255:224] := \text{SRC1}[255:224]; \\
\text{ELSE} \\
\text{DEST}[255:224] := \text{SRC2}[255:224]; \text{ Fl;}
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:256] := 0
\]
VPMAXUD (EVEX Encoded Versions)

$(KL, VL) = (4, 128), (8, 256), (16, 512)$

FOR $j := 0$ TO $KL - 1$
  $i := j \times 32$
  IF $k1[j]$ OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*) THEN
      IF $SRC1[i+31:i] > SRC2[31:0]$
        THEN $DEST[i+31:i] := SRC1[i+31:i]$
        ELSE $DEST[i+31:i] := SRC2[31:0]$
      FI;
    ELSE
      IF $SRC1[i+31:i] > SRC2[i+31:i]$
        THEN $DEST[i+31:i] := SRC1[i+31:i]$
        ELSE $DEST[i+31:i] := SRC2[i+31:i]$
      FI;
    ELSE
      IF *merging-masking* THEN $DEST[i+31:i] remains unchanged*
      ELSE $zeroing-masking$
        THEN $DEST[i+31:i] := 0$
      FI
    FI;
  ENDFOR;
  $DEST[MAXVL-1:VL] := 0$

VPMAXUQ (EVEX Encoded Versions)

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

FOR $j := 0$ TO $KL - 1$
  $i := j \times 64$
  IF $k1[j]$ OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*) THEN
      IF $SRC1[i+63:i] > SRC2[63:0]$
        THEN $DEST[i+63:i] := SRC1[i+63:i]$
        ELSE $DEST[i+63:i] := SRC2[63:0]$
      FI;
    ELSE
      IF $SRC1[i+31:i] > SRC2[i+31:i]$
        THEN $DEST[i+63:i] := SRC1[i+63:i]$
        ELSE $DEST[i+63:i] := SRC2[i+63:i]$
      FI;
    ELSE
      IF *merging-masking* THEN $DEST[i+63:i] remains unchanged*
      ELSE $zeroing-masking$
        THEN $DEST[i+63:i] := 0$
      FI
    FI;
  ENDFOR;
  $DEST[MAXVL-1:VL] := 0$
Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32( __m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32( __m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32( __mmask16 k, __m512i a, __m512i b);

VPMAXUQ __m512i _mm512_max_epu64( __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64( __mmask8 k, __m512i a, __m512i b);

VPMAXUD __m256i _mm256_max_epu32( __m256i a, __m256i b);
VPMAXUD __m256i _mm256_mask_max_epu32( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32( __mmask16 k, __m256i a, __m256i b);

VPMAXUQ __m256i _mm256_max_epu64( __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64( __mmask8 k, __m256i a, __m256i b);

VPMAXUD __m128i _mm_mask_max_epu32( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32( __mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
### PMINSB/PMINSW—Minimum of Packed Signed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EA /r1 PMINSW mm1, mm2/m64</td>
<td>A V/V</td>
<td>SSE</td>
<td>Compare signed word integers in mm2/m64 and mm1 and return minimum values.</td>
<td></td>
</tr>
<tr>
<td>66 0F 38 38 /r PMINSW xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F EA /r PMINSW xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 38 /r VPMINSB xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 38 /r VPMINSB ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 38 /r VPMINSB zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG EA /r VPMINSW xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG EA /r VPMINSW ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG EA /r VPMINSW zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**


### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMtr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description
Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be a YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSW (64-bit Operands)
IF DEST[15:0] < SRC[15:0] THEN
   DEST[15:0] := DEST[15:0];
ELSE
   DEST[15:0] := SRC[15:0]; Fl;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
   DEST[63:48] := DEST[63:48];
ELSE
   DEST[63:48] := SRC[63:48]; Fl;
PMINSB (128-bit Legacy SSE Version)
IF DEST[7:0] < SRC[7:0] THEN
   DEST[7:0] := DEST[7:0];
ELSE
   DEST[15:0] := SRC[7:0]; Fl;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
   DEST[127:120] := DEST[127:120];
ELSE
   DEST[127:120] := SRC[127:120]; Fl;
DEST[MAXVL-1:128] := 0

VPMINSB (VEX.128 Encoded Version)
IF SRC1[7:0] < SRC2[7:0] THEN
   DEST[7:0] := SRC1[7:0];
ELSE
   DEST[7:0] := SRC2[7:0]; Fl;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
   DEST[127:120] := SRC1[127:120];
ELSE
   DEST[127:120] := SRC2[127:120]; Fl;
DEST[MAXVL-1:128] := 0
VPMINSB (VEX.256 Encoded Version)
IF SRC1[7:0] < SRC2[7:0] THEN
   DEST[7:0] := SRC1[7:0];
ELSE
   DEST[15:0] := SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
   DEST[255:248] := SRC1[255:248];
ELSE
   DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0

VPMINSB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
   i := j * 8
IF k1[j] OR *no writemask* THEN
   IF SRC1[i+7:i] < SRC2[i+7:i]
      THEN DEST[i+7:i] := SRC1[i+7:i];
   ELSE DEST[i+7:i] := SRC2[i+7:i];
   FI;
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
   ELSE ; zeroing-masking
      DEST[i+7:i] := 0
   FI
ENDIF;
ENDFOR;
DEST[MAXVL-1:VL] := 0

PMINSW (128-bit Legacy SSE Version)
IF DEST[15:0] < SRC[15:0] THEN
   DEST[15:0] := DEST[15:0];
ELSE
   DEST[15:0] := SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
   DEST[127:112] := DEST[127:112];
ELSE
   DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] := 0

VPMINSW (VEX.128 Encoded Version)
IF SRC1[15:0] < SRC2[15:0] THEN
   DEST[15:0] := SRC1[15:0];
ELSE
   DEST[15:0] := SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
   DEST[127:112] := SRC1[127:112];
ELSE
   DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0
VPMINSW (VEX.256 Encoded Version)
IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] := SRC1[15:0];
ELSE
    DEST[15:0] := SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
    DEST[255:240] := SRC1[255:240];
ELSE
    DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0

VPMINSW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] := SRC1[i+15:i];
        ELSE DEST[i+15:i] := SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+15:i] := 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMINSB __m512i _mm512_min_epi8(__m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi8(__mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16( __m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16( __mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_min_epi8(__m256i a, __m256i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16( __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8(__mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask4 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16(__mmask4 k, __m128i a, __m128i b);
VPMINSW __m64 _mm_min_pi16(__m64 a, __m64 b)
SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."
Additionally:
#MF (64-bit operations only) If there is a pending x87 FPU exception.
PMINS/PMINSQ—Minimum of Packed Signed Integers

### Description

Perform a SIMD compare of the packed signed dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

**Opcode/Description**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/E n</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 39 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4.1</td>
<td>Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 39 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 39 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 39 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 39 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 39 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed signed dword integers in zmm2 and zmm3/m12/m32bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 39 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 39 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 39 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Compare packed signed dword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>
**Operation**

**PMINSD (128-bit Legacy SSE Version)**

IF DEST[31:0] < SRC[31:0] THEN
    DEST[31:0] := DEST[31:0];
ELSE
    DEST[31:0] := SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)

    DEST[127:96] := DEST[127:96];
ELSE
    DEST[127:96] := SRC[127:96]; FI;

DEST[MAXVL-1:128] (Unmodified)

**VPMINSD (VEX.128 Encoded Version)**

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] := SRC1[31:0];
ELSE
    DEST[31:0] := SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

    DEST[127:96] := SRC1[127:96];
ELSE
    DEST[127:96] := SRC2[127:96]; FI;

DEST[MAXVL-1:128] := 0

**VPMINSD (VEX.256 Encoded Version)**

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] := SRC1[31:0];
ELSE
    DEST[31:0] := SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)

    DEST[255:224] := SRC1[255:224];
ELSE
    DEST[255:224] := SRC2[255:224]; FI;

DEST[MAXVL-1:256] := 0
**VPMINSQ (EVEX Encoded Versions)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)
\[i := j \times 64\]

IF \(k1[j] \) OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

IF SRC1[i+63:i] < SRC2[63:0]

THEN DEST[i+63:i] := SRC1[i+63:i];
ELSE DEST[i+63:i] := SRC2[63:0];
FI;
ELSE

IF SRC1[i+63:i] < SRC2[i+63:i]

THEN DEST[i+63:i] := SRC1[i+63:i];
ELSE DEST[i+63:i] := SRC2[i+63:i];
FI;
FI;
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking

DEST[i+63:i] := 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

**VPMINSQ (EVEX Encoded Versions)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)
\[i := j \times 32\]

IF \(k1[j] \) OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN

IF SRC1[i+31:i] < SRC2[31:0]

THEN DEST[i+31:i] := SRC1[i+31:i];
ELSE DEST[i+31:i] := SRC2[31:0];
FI;
ELSE

IF SRC1[i+31:i] < SRC2[i+31:i]

THEN DEST[i+31:i] := SRC1[i+31:i];
ELSE DEST[i+31:i] := SRC2[i+31:i];
FI;
FI;
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking

DEST[i+31:i] := 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPMINSI __m512i _mm512_min_epi32(_m512i a, __m512i b);
VPMINSI __m512i _mm512_mask_min_epi32(_m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSI __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(_m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
### PMINUB/PMINUW—Minimum of Packed Unsigned Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| NP 0F DA /r\(^1\)  
PMINUB mm1, mm2/m64 | A | V/V | SSE | Compare unsigned byte integers in mm2/m64 and mm1 and returns minimum values. |
| 66 0F DA /r  
PMINUB xmm1, xmm2/m128 | A | V/V | SSE2 | Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1. |
| 66 0F 38 3A/r  
PMINUW xmm1, xmm2/m128 | A | V/V | SSE4_1 | Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1. |
| VEX.128.66.0F DA /r  
VPMINUB xmm1, xmm2, xmm3/m128 | B | V/V | AVX | Compare packed unsigned byte integers in xmm3/m128 and xmm2 and store packed minimum values in xmm1. |
| VEX.128.66.0F38 3A/r  
VPMINUW xmm1, xmm2/m128 | B | V/V | AVX | Compare packed unsigned word integers in xmm3/m128 and xmm2 and store packed minimum values in xmm1. |
| VEX.256.66.0F DA /r  
VPMINUB ymm1, ymm2, ymm3/m256 | B | V/V | AVX2 | Compare packed unsigned byte integers in ymm3/m256 and ymm2 and store packed minimum values in ymm1. |
| VEX.256.66.0F38 3A/r  
VPMINUW ymm1, ymm2/m256 | B | V/V | AVX2 | Compare packed unsigned word integers in ymm3/m256 and ymm2 and store packed minimum values in ymm1. |
| EVEX.128.66.0F DA /r  
VPMINUB xmm1 [k1]{z}, xmm2, xmm3/m128 | C | V/V | AVX512VL AVX512BW | Compare packed unsigned byte integers in xmm3/m128 and xmm2 and store packed minimum values in xmm1 under writemask k1. |
| EVEX.256.66.0F DA /r  
VPMINUB ymm1 [k1]{z}, ymm2, ymm3/m256 | C | V/V | AVX512VL AVX512BW | Compare packed unsigned byte integers in ymm3/m256 and ymm2 and store packed minimum values in ymm1 under writemask k1. |
| EVEX.512.66.0F DA /r  
VPMINUB zmm1 [k1]{z}, zmm2, zmm3/m512 | C | V/V | AVX512BW | Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1. |
| EVEX.128.66.0F38 3A/r  
VPMINUW xmm1[k1]{z}, xmm2, xmm3/m128 | C | V/V | AVX512VL AVX512BW | Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1. |
| EVEX.256.66.0F38 3A/r  
VPMINUW ymm1[k1]{z}, ymm2, ymm3/m256 | C | V/V | AVX512VL AVX512BW | Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1. |
| EVEX.512.66.0F38 3A/r  
VPMINUW zmm1[k1]{z}, zmm2, zmm3/m512 | C | V/V | AVX512BW | Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1. |

**NOTES:**
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**PMINUB (64-bit Operands)**

\[
\text{IF } \text{DEST}[7:0] < \text{SRC}[17:0] \text{ THEN} \\
\quad \text{DEST}[7:0] := \text{DEST}[7:0]; \\
\text{ELSE} \\
\quad \text{DEST}[7:0] := \text{SRC}[7:0]; \quad \text{FI}; \\
\quad (* \text{Repeat operation for 2nd through 7th bytes in source and destination operands} *) \\
\text{IF } \text{DEST}[63:56] < \text{SRC}[63:56] \text{ THEN} \\
\quad \text{DEST}[63:56] := \text{DEST}[63:56]; \\
\text{ELSE} \\
\quad \text{DEST}[63:56] := \text{SRC}[63:56]; \quad \text{FI};
\]

**PMINUB (128-bit Operands)**

\[
\text{IF } \text{DEST}[7:0] < \text{SRC}[7:0] \text{ THEN} \\
\quad \text{DEST}[7:0] := \text{DEST}[7:0]; \\
\text{ELSE} \\
\quad \text{DEST}[15:0] := \text{SRC}[7:0]; \quad \text{FI}; \\
\quad (* \text{Repeat operation for 2nd through 15th bytes in source and destination operands} *) \\
\text{IF } \text{DEST}[127:120] < \text{SRC}[127:120] \text{ THEN} \\
\quad \text{DEST}[127:120] := \text{DEST}[127:120]; \\
\text{ELSE} \\
\quad \text{DEST}[127:120] := \text{SRC}[127:120]; \quad \text{FI}; \\
\quad \text{DEST}[\text{MAXVL-1:128}] \quad \text{(Unmodified)}
\]
VPMINUB (VEX.128 Encoded Version)
IF SRC1[7:0] < SRC2[7:0] THEN
  DEST[7:0] := SRC1[7:0];
ELSE
  DEST[7:0] := SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
  DEST[127:120] := SRC1[127:120];
ELSE
  DEST[127:120] := SRC2[127:120]; FI;
DEST[MAXVL-1:128] := 0

VPMINUB (VEX.256 Encoded Version)
IF SRC1[7:0] < SRC2[7:0] THEN
  DEST[7:0] := SRC1[7:0];
ELSE
  DEST[15:0] := SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
  DEST[255:248] := SRC1[255:248];
ELSE
  DEST[255:248] := SRC2[255:248]; FI;
DEST[MAXVL-1:256] := 0

VPMINUB (EVEX Encoded Versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask* THEN
    IF SRC1[i+7:i] < SRC2[i+7:i]
      THEN DEST[i+7:i] := SRC1[i+7:i];
    ELSE DEST[i+7:i] := SRC2[i+7:i];
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN "DEST[i+7:i] remains unchanged"
    ELSE ; zeroing-masking
      DEST[i+7:i] := 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

PMINUW (128-bit Operands)
IF DEST[15:0] < SRC[15:0] THEN
  DEST[15:0] := DEST[15:0];
ELSE
  DEST[15:0] := SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:112] := DEST[127:112];
ELSE
  DEST[127:112] := SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
VPMINUw (VEX.128 Encoded Version)
IF SRC1[15:0] < SRC2[15:0] THEN
  DEST[15:0] := SRC1[15:0];
ELSE
  DEST[15:0] := SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
  DEST[127:112] := SRC1[127:112];
ELSE
  DEST[127:112] := SRC2[127:112]; FI;
DEST[MAXVL-1:128] := 0

VPMINUw (VEX.256 Encoded Version)
IF SRC1[15:0] < SRC2[15:0] THEN
  DEST[15:0] := SRC1[15:0];
ELSE
  DEST[15:0] := SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
  DEST[255:240] := SRC1[255:240];
ELSE
  DEST[255:240] := SRC2[255:240]; FI;
DEST[MAXVL-1:256] := 0

VPMINUw (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask* THEN
    IF SRC1[i+15:i] < SRC2[i+15:i]
      THEN DEST[i+15:i] := SRC1[i+15:i];
      ELSE DEST[i+15:i] := SRC2[i+15:i];
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPMINUB _m512i _mm512_min_epu8(__m512i a, __m512i b);
VPMINUB _m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB _m512i _mm512_maskz_min_epu8(__mmask64 k, __m512i a, __m512i b);
VPMINUW _m512i _mm512_min_epu16(__m512i a, __m512i b);
VPMINUW _m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW _m512i _mm512_maskz_min_epu16(__mmask32 k, __m512i a, __m512i b);
VPMINUW _m256i _mm256_min_epu8(__m256i a, __m256i b);
VPMINUW _m256i _mm256_mask_min_epu8(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW _m256i _mm256_maskz_min_epu8(__mmask16 k, __m256i a, __m256i b);
VPMINUW _m128i _mm128_min_epu8(__m128i a, __m128i b);
VPMINUW _m128i _mm128_mask_min_epu8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW _m128i _mm128_maskz_min_epu8(__mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 (__m128i a, __m128i b);
VPMINUW _m256i _mm256_min_epu8 (__m256i a, __m256i b);
VPMINUW _m256i _mm256_min_epu16 (__m256i a, __m256i b);
PMINUB __m64 _m_min_epu8 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 3B r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>PMINUD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 3B r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.</td>
</tr>
<tr>
<td>VPMINUD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 3B r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.</td>
</tr>
<tr>
<td>VPMINUD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 3B r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINUD xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 3B r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINUD ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 3B r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINUD zmm1 [k1][z], zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 3B r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINUQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 3B r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINUQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 3B r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>VPMINUQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM(reg (r, w))</td>
<td>ModRM(r/m (r))</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM(reg (w))</td>
<td>VEX.vvvv (r)</td>
<td>ModRM(r/m (r))</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM(reg (w))</td>
<td>VEX.vvvv (r)</td>
<td>ModRM(r/m (r))</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.
EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**PMINUD (128-bit Legacy SSE Version)**

PMINUD instruction for 128-bit operands:

IF DEST[31:0] < SRC[31:0] THEN
    DEST[31:0] := DEST[31:0];
ELSE
    DEST[31:0] := SRC[31:0]; FI;

(* Repeat operation for 2nd through 7th words in source and destination operands *)

    DEST[127:96] := DEST[127:96];
ELSE
    DEST[127:96] := SRC[127:96]; FI;

DEST[MAXVL-1:128] := 0

**VPMINUD (VEX.128 Encoded Version)**

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] := SRC1[31:0];
ELSE
    DEST[31:0] := SRC2[31:0]; FI;

(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)

    DEST[127:96] := SRC1[127:96];
ELSE
    DEST[127:96] := SRC2[127:96]; FI;

DEST[MAXVL-1:128] := 0

**VPMINUD (VEX.256 Encoded Version)**

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] := SRC1[31:0];
ELSE
    DEST[31:0] := SRC2[31:0]; FI;

(* Repeat operation for 2nd through 7th dwords in source and destination operands *)

    DEST[255:224] := SRC1[255:224];
ELSE
    DEST[255:224] := SRC2[255:224]; FI;

DEST[MAXVL-1:256] := 0
VPMINUQ (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[i+63:i] < SRC2[i+63:i]
          THEN DEST[i+63:i] := SRC1[i+63:i];
          ELSE DEST[i+63:i] := SRC2[i+63:i];
        FI;
      ELSE
        IF SRC1[i+63:i] < SRC2[i+63:i]
          THEN DEST[i+63:i] := SRC1[i+63:i];
          ELSE DEST[i+63:i] := SRC2[i+63:i];
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI;
      FI;
  ENDFOR;
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VPMINUD __m512i _mm512_min_epu32(__m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64(__m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_min_epu32(__m256i a, __m256i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32(__mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_min_epu64(__m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32(__m128i a, __m128i b);
VPMINUQ __m256i _mm256_min_epu32(__m256i a, __m256i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
PMOVMSKB—Move Byte Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D7 /r¹ PMOVMSKB reg, mm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Move a byte mask of mm to reg. The upper bits of r32 or r64 are zeroed</td>
</tr>
<tr>
<td>66 0F D7 /r PMOVMSKB reg, xmm</td>
<td>RM</td>
<td>V/V</td>
<td>SSE2</td>
<td>Move a byte mask of xmm to reg. The upper bits of r32 or r64 are zeroed</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D7 /r VPMOVMSKB reg, xmm1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D7 /r VPMOVMSKB reg, ymm1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Move a 32-bit mask of ymm1 to reg. The upper bits of r64 are filled with zeros.</td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.
128-bit Legacy SSE version: The source operand is an XMM register.
VEX.128 encoded version: The source operand is an XMM register.
VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMSKB (with 64-bit Source Operand and r32)

\[
\begin{align*}
\text{r32}[0] & := \text{SRC}[7]; \\
\text{r32}[1] & := \text{SRC}[15]; \\
(* \text{ Repeat operation for bytes 2 through 6 } *) \\
\text{r32}[7] & := \text{SRC}[63]; \\
\text{r32}[31:8] & := \text{ZERO_FILL};
\end{align*}
\]
(V)PMOVMSKB (with 128-bit Source Operand and r32)
   r32[0] := SRC[7];
   r32[1] := SRC[15];
   (* Repeat operation for bytes 2 through 14 *)
   r32[15] := SRC[127];
   r32[31:16] := ZERO_FILL;

VPMOVMSKB (with 256-bit Source Operand and r32)
   r32[0] := SRC[7];
   r32[1] := SRC[15];
   (* Repeat operation for bytes 3rd through 31*)
   r32[31] := SRC[255];

PMOVMSKB (with 64-bit Source Operand and r64)
   r64[0] := SRC[7];
   r64[1] := SRC[15];
   (* Repeat operation for bytes 2 through 6 *)
   r64[7] := SRC[63];
   r64[63:8] := ZERO_FILL;

(V)PMOVMSKB (with 128-bit Source Operand and r64)
   r64[0] := SRC[7];
   r64[1] := SRC[15];
   (* Repeat operation for bytes 2 through 14 *)
   r64[15] := SRC[127];
   r64[63:16] := ZERO_FILL;

VPMOVMSKB (with 256-bit Source Operand and r64)
   r64[0] := SRC[7];
   r64[1] := SRC[15];
   (* Repeat operation for bytes 2 through 31*)
   r64[31] := SRC[255];
   r64[63:32] := ZERO_FILL;

Intel C/C++ Compiler Intrinsic Equivalent
PMOVMSKB int _mm_movemask_pi8(__m64 a)
(V)PMOVMSKB int _mm_movemask_epi8 (__m128i a)
VPMOVMSKB int _mm256_movemask_epi8 (__m256i a)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
See Table 2-24, “Type 7 Class Exception Conditions,” additionally:
#UD If VEX.vvvv ≠ 1111B.
## PMOVSX—Packed Move With Sign Extend

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0f 38 20 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSBW xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 21 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSBD xmm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 22 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSBQ xmm1, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 23 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 24 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSD WD xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 25 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSD WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 27 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 28 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 16-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 29 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 2A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 2B /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 2C /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 2D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 16-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 2E /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0f 38 2F /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>PMOVSDQ WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 20 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
</tr>
<tr>
<td>VPMOVSBW xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 21 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>VPMOVSBD xmm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 22 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>VPMOVSBQ xmm1, xmm2/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 23 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>VPMOVSDQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 24 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
</tr>
<tr>
<td>VPMOVSD WD xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 25 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
</tr>
<tr>
<td>VPMOVSD WQ xmm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 20 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSBW ymm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 21 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSBD ymm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 22 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSBQ ymm1, xmm2/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 23 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSDQ ymm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 24 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSD WD ymm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 25 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSD Q ymm1, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 20 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSBW xmm1 {k1}{z}, xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 20 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
</tr>
<tr>
<td>VPMOVSBW ymm1 {k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 20 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.</td>
</tr>
<tr>
<td>VPMOVSBW zmm1 {k1}{z}, xmm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode/Instruction</td>
<td>Op / En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>------------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 21 /r VPMOVXSXB xmm1 {k1}{z}, xmm2/m32</td>
<td>C  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 21 /r VPMOVXSXB ymm1 {k1}[z], xmm2/m64</td>
<td>C  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 21 /r VPMOVXSXB zmm1 {k1}[z], xmm2/m128</td>
<td>C  V/V</td>
<td>AVX512F</td>
<td>Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 22 /r VPMOVXSXB xmm1 {k1}{z}, xmm2/m16</td>
<td>D  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 22 /r VPMOVXSXB ymm1 {k1}[z], xmm2/m32</td>
<td>D  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 22 /r VPMOVXSXB zmm1 {k1}[z], xmm2/m64</td>
<td>D  V/V</td>
<td>AVX512F</td>
<td>Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 23 /r VPMOVXSXW xmm1 {k1}{z}, xmm2/m64</td>
<td>B  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 23 /r VPMOVXSXW ymm1 {k1}[z], xmm2/m128</td>
<td>B  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 23 /r VPMOVXSXW zmm1 {k1}[z], ymm2/m256</td>
<td>B  V/V</td>
<td>AVX512F</td>
<td>Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 24 /r VPMOVXSXW Q xmm1 {k1}[z], xmm2/m32</td>
<td>C  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 24 /r VPMOVXSXW Q ymm1 {k1}[z], xmm2/m64</td>
<td>C  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 24 /r VPMOVXSXW Q zmm1 {k1}[z], ymm2/m256</td>
<td>C  V/V</td>
<td>AVX512F</td>
<td>Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m256 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 25 /r VPMOVXSX D xmm1 {k1}[z], xmm2/m64</td>
<td>B  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 25 /r VPMOVXSX D ymm1 {k1}[z], xmm2/m128</td>
<td>B  V/V</td>
<td>AVX512VL AVX512F</td>
<td>Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 25 /r VPMOVXSX D zmm1 {k1}[z], ymm2/m256</td>
<td>B  V/V</td>
<td>AVX512F</td>
<td>Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
### Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)

\[
\begin{align*}
\text{DEST}[15:0] &= \text{SignExtend}(\text{SRC}[7:0]); \\
\text{DEST}[31:16] &= \text{SignExtend}(\text{SRC}[15:8]); \\
\text{DEST}[47:32] &= \text{SignExtend}(\text{SRC}[23:16]); \\
\text{DEST}[63:48] &= \text{SignExtend}(\text{SRC}[31:24]); \\
\text{DEST}[79:64] &= \text{SignExtend}(\text{SRC}[39:32]); \\
\text{DEST}[95:64] &= \text{SignExtend}(\text{SRC}[47:32]); \\
\text{DEST}[127:96] &= \text{SignExtend}(\text{SRC}[55:48]); \\
\end{align*}
\]

#### Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)

\[
\begin{align*}
\text{DEST}[31:0] &= \text{SignExtend}(\text{SRC}[7:0]); \\
\text{DEST}[63:32] &= \text{SignExtend}(\text{SRC}[15:8]); \\
\text{DEST}[95:64] &= \text{SignExtend}(\text{SRC}[23:16]); \\
\text{DEST}[127:96] &= \text{SignExtend}(\text{SRC}[31:24]); \\
\end{align*}
\]

#### Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)

\[
\begin{align*}
\text{DEST}[63:0] &= \text{SignExtend}(\text{SRC}[7:0]); \\
\text{DEST}[127:64] &= \text{SignExtend}(\text{SRC}[15:8]); \\
\end{align*}
\]

#### Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)

\[
\begin{align*}
\text{DEST}[31:0] &= \text{SignExtend}(\text{SRC}[15:0]); \\
\text{DEST}[63:32] &= \text{SignExtend}(\text{SRC}[31:16]); \\
\text{DEST}[95:64] &= \text{SignExtend}(\text{SRC}[47:32]); \\
\text{DEST}[127:96] &= \text{SignExtend}(\text{SRC}[63:48]); \\
\end{align*}
\]

#### Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)

\[
\begin{align*}
\text{DEST}[63:0] &= \text{SignExtend}(\text{SRC}[15:0]); \\
\text{DEST}[127:64] &= \text{SignExtend}(\text{SRC}[31:16]); \\
\end{align*}
\]
Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)
DEST[63:0] := SignExtend(SRC[31:0]);
DEST[127:64] := SignExtend(SRC[63:32]);

VPMOVXSXBW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
   Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
   Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
         ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPMOVXSXBD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
   Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
   Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
         ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPMOVSBQ (EVEX Encoded Versions)

(\(KL, VL\)) = (2, 128), (4, 256), (8, 512)
Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[127:0], SRC[15:0])
IF \(VL >= 256\)
   Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[255:128], SRC[31:16])
FI;
IF \(VL >= 512\)
   Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[383:256], SRC[47:32])
   Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[511:384], SRC[63:48])
FI;
FOR \(j := 0\) TO \(KL-1\)
   \(i := j * 64\)
   IF \(k1[j]\) OR *no writemask*
      THEN \(DEST[i+63:i] := TEMP\_DEST[i+63:i]\)
      ELSE
         IF *merging-masking* ; merging-masking
            THEN \(DEST[i+63:i] \text{ remains unchanged}\)
            ELSE *zeroing-masking* ; zeroing-masking
               \(DEST[i+63:i] := 0\)
         FI
   FI;
ENDFOR
\(DEST[MAXVL-1:VL] := 0\)

VPMOVSD (EVEX Encoded Versions)

(\(KL, VL\)) = (4, 128), (8, 256), (16, 512)
Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[127:0], SRC[63:0])
IF \(VL >= 256\)
   Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[255:128], SRC[127:64])
FI;
IF \(VL >= 512\)
   Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[383:256], SRC[191:128])
   Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[511:384], SRC[256:192])
FI;
FOR \(j := 0\) TO \(KL-1\)
   \(i := j * 32\)
   IF \(k1[j]\) OR *no writemask*
      THEN \(DEST[i+31:i] := TEMP\_DEST[i+31:i]\)
      ELSE
         IF *merging-masking* ; merging-masking
            THEN \(DEST[i+31:i] \text{ remains unchanged}\)
            ELSE "zeroing-masking" ; zeroing-masking
               \(DEST[i+31:i] := 0\)
         FI
   FI;
ENDFOR
\(DEST[MAXVL-1:VL] := 0\)
VPMOVVSXWQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed_Sign_Extend_WD_Word_to_QWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_WD_Word_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_WD_Word_to_QWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_WD_Word_to_QWORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPMOVVSXDQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed_Sign_Extend_QD_DWord_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_QD_DWord_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_QD_DWord_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_QD_DWord_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPMOVVSXBW (VEX.256 Encoded Version)

Packed_Sign_Extend_BYTE_to_WD_Word(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WD_Word(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:128] := 0
VPMOVSXBD (VEX.256 Encoded Version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0

VPMOVSXBBQ (VEX.256 Encoded Version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] := 0

VPMOVSXWD (VEX.256 Encoded Version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] := 0

VPMOVSXWQ (VEX.256 Encoded Version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] := 0

VPMOVSXBD (VEX.128 Encoded Version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0

VPMOVSXBBQ (VEX.128 Encoded Version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0

VPMOVSXWD (VEX.128 Encoded Version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0

VPMOVSXWQ (VEX.128 Encoded Version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0

VPMOVSXDQ (VEX.128 Encoded Version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] := 0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSBQ
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSWD
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSWQ
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVXDQ
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVXSXBw __m512i _mm512_cvtepi8_epi16(__m512i a);
VPMOVXSXBw __m512i _mm512_mask_cvtepi8_epi16(__m512i a, __mmask32 k, __m512i b);
VPMOVXSXBw __m512i _mm512_maskz_cvtepi8_epi16( __mmask32 k, __m512i b);
VPMOVXSXBD __m512i _mm512_cvtepi8_epi32(__m512i a);
VPMOVXSXBD __m512i _mm512_mask_cvtepi8_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVXSXBD __m512i _mm512_maskz_cvtepi8_epi32( __mmask16 k, __m512i b);
VPMOVXSXBQ __m512i _mm512_cvtepi8_epi64(__m512i a);
VPMOVXSXBQ __m512i _mm512_mask_cvtepi8_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVXSXBQ __m512i _mm512_maskz_cvtepi8_epi64( __mmask8 k, __m512i b);
VPMOVXSXDQ __m512i _mm512_cvtepi32_epi64(__m512i a);
VPMOVXSXDQ __m512i _mm512_mask_cvtepi32_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVXSXDQ __m512i _mm512_maskz_cvtepi32_epi64( __mmask8 k, __m512i b);
VPMOVSXWQ __m512i _mm512_cvtepi16_epi64(__m512i a);
VPMOVSXWQ __m512i _mm512_mask_cvtepi16_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXWQ __m512i _mm512_maskz_cvtepi16_epi64( __mmask8 k, __m512i b);
VPMOVSWBw __m256i _mm256_cvtepi8_epi16(__m256i a);
VPMOVSWBw __m256i _mm256_mask_cvtepi8_epi16(__m256i a, __mmask16 k, __m256i b);
VPMOVSWBw __m256i _mm256_maskz_cvtepi8_epi16( __mmask16 k, __m256i b);
VPMOVSWBD __m256i _mm256_cvtepi8_epi32(__m256i a);
VPMOVSWBD __m256i _mm256_mask_cvtepi8_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVSWBD __m256i _mm256_maskz_cvtepi8_epi32( __mmask8 k, __m256i b);
VPMOVSWBQ __m256i _mm256_cvtepi8_epi64(__m256i a);
VPMOVSWBQ __m256i _mm256_mask_cvtepi8_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSWBQ __m256i _mm256_maskz_cvtepi8_epi64( __mmask8 k, __m256i b);
VPMOVSWD __m256i _mm256_cvtepi16_epi32(__m256i a);
VPMOVSWD __m256i _mm256_mask_cvtepi16_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVSWD __m256i _mm256_maskz_cvtepi16_epi32( __mmask8 k, __m256i b);
PMOVSX—Packed Move With Sign Extend

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”

EVEX-encoded instruction, see Table 2-51, “Type E5 Class Exception Conditions.”

Additionally:

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
### PMOVZX—Packed Move With Zero Extend

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMOVZXBw xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMOVZXBD xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMOVZXBQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMOVZXWD xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMOVZXWQ xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>PMOVZXDQ xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>SSE4_1</td>
<td>Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMOVZXBw xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMOVZXBD xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMOVZXBQ xmm1, xmm2/m16</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMOVZXWD xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMOVZXWQ xmm1, xmm2/m32</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VPMOVZXDQ xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX</td>
<td>Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Note: AVX and AVX2 support is indicated where applicable.
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En.</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 35 /r VPMOVZXWDQ ymm1, xmm2/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.30.WIG /r VPMOVZXBW xmm1 [k1]{z}, xmm2/m64</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 [k1]{z}, xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW ymm1 [k1]{z}, ymm2/m256</td>
<td>B V/V</td>
<td>AVX512BW</td>
<td>Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.31.WIG /r VPMOVZXBD xmm1 [k1]{z}, xmm2/m32</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 [k1]{z}, xmm2/m64</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 [k1]{z}, ymm2/m256</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.32.WIG /r VPMOVZXBD xmm1 [k1]{z}, xmm2/m16</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 32 /r VPMOVZXBD ymm1 [k1]{z}, xmm2/m32</td>
<td>D V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 32 /r VPMOVZXBD zmm1 [k1]{z}, xmm2/m64</td>
<td>D V/V</td>
<td>AVX512F</td>
<td>Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.33.WIG /r VPMOVZXBD xmm1 [k1]{z}, xmm2/m64</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 33 /r VPMOVZXBD ymm1 [k1]{z}, xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 33 /r VPMOVZXBD zmm1 [k1]{z}, xmm2/m256</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.34.WIG /r VPMOVZXWQ xmm1 [k1]{z}, xmm2/m32</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 [k1]{z}, xmm2/m64</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 [k1]{z}, xmm2/m64</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Zero extend 8 packed 16-bit integers in xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
**Description**

Legacy, VEX, and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**Packed_Zero_Extend_BYTE_to_WORD***(DEST, SRC)***

\[
\text{DEST}[15:0] := \text{ZeroExtend}(\text{SRC}[7:0]); \\
\text{DEST}[31:16] := \text{ZeroExtend}(\text{SRC}[15:8]); \\
\text{DEST}[47:32] := \text{ZeroExtend}(\text{SRC}[23:16]); \\
\text{DEST}[63:48] := \text{ZeroExtend}(\text{SRC}[31:24]); \\
\text{DEST}[79:64] := \text{ZeroExtend}(\text{SRC}[39:32]); \\
\text{DEST}[95:80] := \text{ZeroExtend}(\text{SRC}[47:40]); \\
\text{DEST}[111:96] := \text{ZeroExtend}(\text{SRC}[55:48]); \\
\text{DEST}[127:112] := \text{ZeroExtend}(\text{SRC}[63:56]);
\]

**Packed_Zero_Extend_BYTE_to_DWORD***(DEST, SRC)***

\[
\text{DEST}[31:0] := \text{ZeroExtend}(\text{SRC}[7:0]); \\
\text{DEST}[63:32] := \text{ZeroExtend}(\text{SRC}[15:8]); \\
\text{DEST}[95:64] := \text{ZeroExtend}(\text{SRC}[23:16]); \\
\text{DEST}[127:96] := \text{ZeroExtend}(\text{SRC}[31:24]);
\]
Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)
DEST[63:0] := ZeroExtend(SRC[7:0]);
DEST[127:64] := ZeroExtend(SRC[15:8]);

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)
DEST[31:0] := ZeroExtend(SRC[15:0]);
DEST[95:64] := ZeroExtend(SRC[47:32]);

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)
DEST[63:0] := ZeroExtend(SRC[15:0]);
DEST[127:64] := ZeroExtend(SRC[31:16]);

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)
DEST[63:0] := ZeroExtend(SRC[31:0]);
DEST[127:64] := ZeroExtend(SRC[63:32]);

VPMOVZXBW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
  Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
  Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
  Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TEMP_DEST[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
INSTRUCTION SET REFERENCE, M-U

**VPMOVZXBD (EVEX Encoded Versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed.ZERO_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])

IF VL >= 256
   Packed.ZERO_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;

IF VL >= 512
   Packed.ZERO_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
   Packed.ZERO_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;

FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i] := 0
   FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPMOVZXBQ (EVEX Encoded Versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed.ZERO_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])

IF VL >= 256
   Packed.ZERO_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;

IF VL >= 512
   Packed.ZERO_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
   Packed.ZERO_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;

FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+63:i] := 0
   FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
VPMOVCXD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
   Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
   Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
   Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPMOVCXQ (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
   Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
   Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
   Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
               DEST[i+63:i] := 0
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPMOVZXQ (EVEX Encoded Versions)

(\(KL, VL\)) = (2, 128), (4, 256), (8, 512)

\(\text{Packed\_Zero\_Extend\_DWORD\_to\_QWORD}(\text{TEMP\_DEST}[127:0], \text{SRC}[63:0])\)

\(\text{IF } VL \geq 256\)

\(\text{Packed\_Zero\_Extend\_DWORD\_to\_QWORD}(\text{TEMP\_DEST}[255:128], \text{SRC}[127:64])\)

\(\text{FI};\)

\(\text{IF } VL \geq 512\)

\(\text{Packed\_Zero\_Extend\_DWORD\_to\_QWORD}(\text{TEMP\_DEST}[511:384], \text{SRC}[255:192])\)

\(\text{Packed\_Zero\_Extend\_DWORD\_to\_QWORD}(\text{TEMP\_DEST}[383:256], \text{SRC}[191:128])\)

\(\text{FI};\)

\(\text{FOR } j := 0 \text{ TO } KL-1\)

\(i := j \times 64\)

\(\text{IF } k1[j] \text{ OR } \text{*no writemask*}\)

\(\text{THEN } \text{DEST}[i+63:i] := \text{TEMP\_DEST}[i+63:i]\)

\(\text{ELSE}\)

\(\text{IF } \text{*merging-masking*}; \text{merging-masking}\)

\(\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged*}\)

\(\text{ELSE } \text{*zeroing-masking*}; \text{zeroing-masking}\)

\(\text{DEST}[i+63:i] := 0\)

\(\text{FI}\)

\(\text{FI}\)

\(\text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

VPMOVZXQ (VEX.256 Encoded Version)

\(\text{Packed\_Zero\_Extend\_BYTE\_to\_WORD}(\text{DEST}[127:0], \text{SRC}[63:0])\)

\(\text{Packed\_Zero\_Extend\_BYTE\_to\_WORD}(\text{DEST}[255:128], \text{SRC}[127:64])\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)

VPMOVZXQ (VEX.256 Encoded Version)

\(\text{Packed\_Zero\_Extend\_BYTE\_to\_DWORD}(\text{DEST}[127:0], \text{SRC}[31:0])\)

\(\text{Packed\_Zero\_Extend\_BYTE\_to\_DWORD}(\text{DEST}[255:128], \text{SRC}[63:32])\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)

VPMOVZXQ (VEX.256 Encoded Version)

\(\text{Packed\_Zero\_Extend\_BYTE\_to\_QWORD}(\text{DEST}[127:0], \text{SRC}[15:0])\)

\(\text{Packed\_Zero\_Extend\_BYTE\_to\_QWORD}(\text{DEST}[255:128], \text{SRC}[31:16])\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)

VPMOVZXQ (VEX.256 Encoded Version)

\(\text{Packed\_Zero\_Extend\_WORD\_to\_DWORD}(\text{DEST}[127:0], \text{SRC}[63:0])\)

\(\text{Packed\_Zero\_Extend\_WORD\_to\_DWORD}(\text{DEST}[255:128], \text{SRC}[127:64])\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)

VPMOVZXQ (VEX.256 Encoded Version)

\(\text{Packed\_Zero\_Extend\_WORD\_to\_QWORD}(\text{DEST}[127:0], \text{SRC}[31:0])\)

\(\text{Packed\_Zero\_Extend\_WORD\_to\_QWORD}(\text{DEST}[255:128], \text{SRC}[63:32])\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)

VPMOVZXQ (VEX.256 Encoded Version)

\(\text{Packed\_Zero\_Extend\_DWORD\_to\_QWORD}(\text{DEST}[127:0], \text{SRC}[63:0])\)

\(\text{Packed\_Zero\_Extend\_DWORD\_to\_QWORD}(\text{DEST}[255:128], \text{SRC}[127:64])\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)
VPMOVZXBw (VEX.128 Encoded Version)
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] := 0

VPMOVZXBD (VEX.128 Encoded Version)
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] := 0

VPMOVZXBQ (VEX.128 Encoded Version)
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] := 0

VPMOVZXWD (VEX.128 Encoded Version)
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] := 0

VPMOVZXWQ (VEX.128 Encoded Version)
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] := 0

VPMOVZXDQ (VEX.128 Encoded Version)
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVZXBw __m512i _mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBD __m512i _mm512_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBw __m512i _mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i b);
VPMOVZXWD __m512i _mm512_cvtepu8_epi32(__m128i a);

PMOVZXBw
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)
VPMOVZXBW __m128i _mm_mask_cvtepu8_epi16(__m128i a, __mmask8 k, __m128i b);  
VPMOVZXBW __m128i _mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_mask_cvtepu8_epi32(__m128i a, __mmask16 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32(__mmask16 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);  
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i b);  
VPMOVZXBD __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i b);  

SIMD Floating-Point Exceptions
None.

PMOVZXBW—Packed Move With Zero Extend
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-22, “Type 5 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-51, “Type E5 Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
PMULDQ—Multiply Packed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 28 /r PMULDQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 28 /r VPMULDQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 28 /r VPMULDQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 28 /r VPMULDQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.
EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM register. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

**Operation**

**VPMULQD (EVEX Encoded Versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[\text{FOR } j := 0 \text{ TO } KL-1\]

\[i := j \times 64\]

IF \(k1[j]\) OR *no writemask*

THEN

IF (EVEX.b = 1) AND (SRC2 *is memory*)

THEN \(\text{DEST[i+63:i]} := \text{SignExtend64( SRC1[i+31:i]}) \times \text{SignExtend64( SRC2[31:0])}\)

ELSE \(\text{DEST[i+63:i]} := \text{SignExtend64( SRC1[i+31:i]}) \times \text{SignExtend64( SRC2[i+31:i])}\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN \(\text{DEST[i+63:i]} \text{ remains unchanged}\)

ELSE *zeroing-masking* ; zeroing-masking

\(\text{DEST[i+63:i]} := 0\)

FI

ENDFOR

\(\text{DEST[MAXVL-1:VL]} := 0\)

**VPMULQD (VEX.256 Encoded Version)**

\(\text{DEST[63:0]} := \text{SignExtend64( SRC1[31:0])} \times \text{SignExtend64( SRC2[31:0])}\)

\(\text{DEST[127:64]} := \text{SignExtend64( SRC1[95:64])} \times \text{SignExtend64( SRC2[95:64])}\)

\(\text{DEST[191:128]} := \text{SignExtend64( SRC1[159:128])} \times \text{SignExtend64( SRC2[159:128])}\)

\(\text{DEST[255:192]} := \text{SignExtend64( SRC1[223:192])} \times \text{SignExtend64( SRC2[223:192])}\)

\(\text{DEST[MAXVL-1:256]} := 0\)

**VPMULQD (VEX.128 Encoded Version)**

\(\text{DEST[63:0]} := \text{SignExtend64( SRC1[31:0])} \times \text{SignExtend64( SRC2[31:0])}\)

\(\text{DEST[127:64]} := \text{SignExtend64( SRC1[95:64])} \times \text{SignExtend64( SRC2[95:64])}\)

\(\text{DEST[MAXVL-1:128]} := 0\)

**PMULQD (128-bit Legacy SSE Version)**

\(\text{DEST[63:0]} := \text{SignExtend64( DEST[31:0])} \times \text{SignExtend64( SRC[31:0])}\)

\(\text{DEST[127:64]} := \text{SignExtend64( DEST[95:64])} \times \text{SignExtend64( SRC[95:64])}\)

\(\text{DEST[MAXVL-1:128]} \text{ (Unmodified)}\)

**Intel C/C++ Compiler Intrinsic Equivalent**

\[\text{VPMULQD} \text{ __m512i _mm512_mul_epi32(__m512i a, __m512i b)};\]

\[\text{VPMULQD} \text{ __m512i _mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b)};\]

\[\text{VPMULQD} \text{ __m512i _mm512_maskz_mul_epi32(__mmask8 k, __m512i a, __m512i b)};\]

\[\text{VPMULQD} \text{ __m256i _mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b)};\]

\[\text{VPMULQD} \text{ __m256i _mm256_maskz_mul_epi32(__mmask8 k, __m256i a, __m256i b)};\]

\[\text{VPMULQD} \text{ __m128i _mm128_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b)};\]

\[\text{VPMULQD} \text{ __m128i _mm128_maskz_mul_epi32(__mmask8 k, __m128i a, __m128i b)};\]

\[\text{VPMULQD} \text{ __m128i _mm_mul_epi32(__m128i a, __m128i b)};\]
VPMULDQ __m256i _mm256_mul_epi32(__m256i a, __m256i b);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
PMULHRSW—Packed Multiply High With Round and Scale

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 0B /r¹ PMULHRSW mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to mm1.</td>
</tr>
<tr>
<td>66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

NOTES:

 Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.
128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**Operation**

**PMULHRSW (With 64-bit Operands)**

\[
\begin{align*}
\text{temp0}[31:0] &= \text{INT32} \left( (\text{DEST}[15:0] \ast \text{SRC}[15:0]) \gg 14 \right) + 1; \\
\text{temp1}[31:0] &= \text{INT32} \left( (\text{DEST}[31:16] \ast \text{SRC}[31:16]) \gg 14 \right) + 1; \\
\text{temp2}[31:0] &= \text{INT32} \left( (\text{DEST}[47:32] \ast \text{SRC}[47:32]) \gg 14 \right) + 1; \\
\text{temp3}[31:0] &= \text{INT32} \left( (\text{DEST}[63:48] \ast \text{SRC}[63:48]) \gg 14 \right) + 1; \\
\text{DEST}[15:0] &= \text{temp0}[16:1]; \\
\text{DEST}[31:16] &= \text{temp1}[16:1]; \\
\text{DEST}[47:32] &= \text{temp2}[16:1]; \\
\text{DEST}[63:48] &= \text{temp3}[16:1];
\end{align*}
\]

**PMULHRSW (With 128-bit Operands)**

\[
\begin{align*}
\text{temp0}[31:0] &= \text{INT32} \left( (\text{DEST}[15:0] \ast \text{SRC}[15:0]) \gg 14 \right) + 1; \\
\text{temp1}[31:0] &= \text{INT32} \left( (\text{DEST}[31:16] \ast \text{SRC}[31:16]) \gg 14 \right) + 1; \\
\text{temp2}[31:0] &= \text{INT32} \left( (\text{DEST}[47:32] \ast \text{SRC}[47:32]) \gg 14 \right) + 1; \\
\text{temp3}[31:0] &= \text{INT32} \left( (\text{DEST}[63:48] \ast \text{SRC}[63:48]) \gg 14 \right) + 1; \\
\text{temp4}[31:0] &= \text{INT32} \left( (\text{DEST}[79:64] \ast \text{SRC}[79:64]) \gg 14 \right) + 1; \\
\text{temp5}[31:0] &= \text{INT32} \left( (\text{DEST}[95:80] \ast \text{SRC}[95:80]) \gg 14 \right) + 1; \\
\text{temp6}[31:0] &= \text{INT32} \left( (\text{DEST}[111:96] \ast \text{SRC}[111:96]) \gg 14 \right) + 1; \\
\text{temp7}[31:0] &= \text{INT32} \left( (\text{DEST}[127:112] \ast \text{SRC}[127:112]) \gg 14 \right) + 1; \\
\text{DEST}[15:0] &= \text{temp0}[16:1]; \\
\text{DEST}[31:16] &= \text{temp1}[16:1]; \\
\text{DEST}[47:32] &= \text{temp2}[16:1]; \\
\text{DEST}[63:48] &= \text{temp3}[16:1]; \\
\text{DEST}[79:64] &= \text{temp4}[16:1]; \\
\text{DEST}[95:80] &= \text{temp5}[16:1]; \\
\text{DEST}[111:96] &= \text{temp6}[16:1]; \\
\text{DEST}[127:112] &= \text{temp7}[16:1];
\end{align*}
\]

**VPMULHRSW (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{temp0}[31:0] &= \text{INT32} \left( (\text{SRC1}[15:0] \ast \text{SRC2}[15:0]) \gg 14 \right) + 1 \\
\text{temp1}[31:0] &= \text{INT32} \left( (\text{SRC1}[31:16] \ast \text{SRC2}[31:16]) \gg 14 \right) + 1 \\
\text{temp2}[31:0] &= \text{INT32} \left( (\text{SRC1}[47:32] \ast \text{SRC2}[47:32]) \gg 14 \right) + 1 \\
\text{temp3}[31:0] &= \text{INT32} \left( (\text{SRC1}[63:48] \ast \text{SRC2}[63:48]) \gg 14 \right) + 1 \\
\text{temp4}[31:0] &= \text{INT32} \left( (\text{SRC1}[79:64] \ast \text{SRC2}[79:64]) \gg 14 \right) + 1 \\
\text{temp5}[31:0] &= \text{INT32} \left( (\text{SRC1}[95:80] \ast \text{SRC2}[95:80]) \gg 14 \right) + 1 \\
\text{temp6}[31:0] &= \text{INT32} \left( (\text{SRC1}[111:96] \ast \text{SRC2}[111:96]) \gg 14 \right) + 1 \\
\text{temp7}[31:0] &= \text{INT32} \left( (\text{SRC1}[127:112] \ast \text{SRC2}[127:112]) \gg 14 \right) + 1 \\
\text{DEST}[15:0] &= \text{temp0}[16:1]; \\
\text{DEST}[31:16] &= \text{temp1}[16:1]; \\
\text{DEST}[47:32] &= \text{temp2}[16:1];
\end{align*}
\]
PMULHRSW—Packed Multiply High With Round and Scale

DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[MAXVL-1:128] := 0

VPMULHRSW (VEX.256 Encoded Version)

\[
\text{temp0}[31:0] := \text{INT32}\left(\left(\text{SRC1}[15:0] * \text{SRC2}[15:0]\right) \gg 14\right) + 1
\]
\[
\text{temp1}[31:0] := \text{INT32}\left(\left(\text{SRC1}[31:16] * \text{SRC2}[31:16]\right) \gg 14\right) + 1
\]
\[
\text{temp2}[31:0] := \text{INT32}\left(\left(\text{SRC1}[47:32] * \text{SRC2}[47:32]\right) \gg 14\right) + 1
\]
\[
\text{temp3}[31:0] := \text{INT32}\left(\left(\text{SRC1}[63:48] * \text{SRC2}[63:48]\right) \gg 14\right) + 1
\]
\[
\text{temp4}[31:0] := \text{INT32}\left(\left(\text{SRC1}[79:64] * \text{SRC2}[79:64]\right) \gg 14\right) + 1
\]
\[
\text{temp5}[31:0] := \text{INT32}\left(\left(\text{SRC1}[95:80] * \text{SRC2}[95:80]\right) \gg 14\right) + 1
\]
\[
\text{temp6}[31:0] := \text{INT32}\left(\left(\text{SRC1}[111:96] * \text{SRC2}[111:96]\right) \gg 14\right) + 1
\]
\[
\text{temp7}[31:0] := \text{INT32}\left(\left(\text{SRC1}[127:112] * \text{SRC2}[127:112]\right) \gg 14\right) + 1
\]
\[
\text{temp8}[31:0] := \text{INT32}\left(\left(\text{SRC1}[143:128] * \text{SRC2}[143:128]\right) \gg 14\right) + 1
\]
\[
\text{temp9}[31:0] := \text{INT32}\left(\left(\text{SRC1}[159:144] * \text{SRC2}[159:144]\right) \gg 14\right) + 1
\]
\[
\text{temp10}[31:0] := \text{INT32}\left(\left(\text{SRC1}[175:160] * \text{SRC2}[175:160]\right) \gg 14\right) + 1
\]
\[
\text{temp11}[31:0] := \text{INT32}\left(\left(\text{SRC1}[191:176] * \text{SRC2}[191:176]\right) \gg 14\right) + 1
\]
\[
\text{temp12}[31:0] := \text{INT32}\left(\left(\text{SRC1}[207:192] * \text{SRC2}[207:192]\right) \gg 14\right) + 1
\]
\[
\text{temp13}[31:0] := \text{INT32}\left(\left(\text{SRC1}[223:208] * \text{SRC2}[223:208]\right) \gg 14\right) + 1
\]
\[
\text{temp14}[31:0] := \text{INT32}\left(\left(\text{SRC1}[239:224] * \text{SRC2}[239:224]\right) \gg 14\right) + 1
\]
\[
\text{temp15}[31:0] := \text{INT32}\left(\left(\text{SRC1}[255:240] * \text{SRC2}[255:240]\right) \gg 14\right) + 1
\]

DEST[15:0] := temp0[16:1]
DEST[31:16] := temp1[16:1]
DEST[63:48] := temp3[16:1]
DEST[79:64] := temp4[16:1]
DEST[95:80] := temp5[16:1]
DEST[111:96] := temp6[16:1]
DEST[127:112] := temp7[16:1]
DEST[143:128] := temp8[16:1]
DEST[159:144] := temp9[16:1]
DEST[175:160] := temp10[16:1]
DEST[MAXVL-1:256] := 0
VPMULHRSw (EVEXEncoded Version)

(QL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
       THEN
           temp[31:0] := ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
           DEST[i+15:i] := tmp[16:1]
       ELSE
           IF *merging-masking*
               THEN *DEST[i+15:i] remains unchanged*
           ELSE *zeroing-masking*
               DEST[i+15:i] := 0
           FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
PMULHRSW __m64 _mm_mulhrs_pi16 ( __m64 a, __m64 b)
(V)PMULHRSW __m128i __mm_mulhrs_epi16 ( __m128i a, __m128i b)
VPMULHRSW __m256i _mm256_mulhrs_epi16 ( __m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
### PMULHUW—Multiply Packed Unsigned Integers and Store High Result

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E4 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.</td>
</tr>
<tr>
<td>66 0F E4 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E4 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E4 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG E4 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG E4 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG E4 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

---

**Operation**

**PMULHUW (With 64-bit Operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] & := \text{DEST}[15:0] \times \text{SRC}[15:0]; (* \text{Unsigned multiplication} *) \\
\text{TEMP1}[31:0] & := \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] & := \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] & := \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{DEST}[15:0] & := \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] & := \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] & := \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] & := \text{TEMP3}[31:16];
\end{align*}
\]

**PMULHUW (With 128-bit Operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] & := \text{DEST}[15:0] \times \text{SRC}[15:0]; (* \text{Unsigned multiplication} *) \\
\text{TEMP1}[31:0] & := \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] & := \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] & := \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{TEMP4}[31:0] & := \text{DEST}[79:64] \times \text{SRC}[79:64]; \\
\text{TEMP5}[31:0] & := \text{DEST}[95:80] \times \text{SRC}[95:80]; \\
\text{TEMP6}[31:0] & := \text{DEST}[111:96] \times \text{SRC}[111:96]; \\
\text{TEMP7}[31:0] & := \text{DEST}[127:112] \times \text{SRC}[127:112]; \\
\text{DEST}[15:0] & := \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] & := \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] & := \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] & := \text{TEMP3}[31:16]; \\
\text{DEST}[79:64] & := \text{TEMP4}[31:16]; \\
\text{DEST}[95:80] & := \text{TEMP5}[31:16]; \\
\text{DEST}[111:96] & := \text{TEMP6}[31:16]; \\
\text{DEST}[127:112] & := \text{TEMP7}[31:16];
\end{align*}
\]
VPMULHUw (VEX.128 Encoded Version)
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0

PMULHUw (VEX.256 Encoded Version)
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0]
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[MAXVL-1:256] := 0
PMULHUW (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
      DEST[i+15:i] := tmp[31:16]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16( __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16( __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16( __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm128_mulhi_epu16 ( __m128i a, __m128i b);
VPMULHUW __m128i _mm128_maskz_mulhi_epu16( __mmask8 k, __m128i a, __m128i b);
PMULHUW __m64 _mm64_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW __m128i _mm_mulhi_epu16 ( __m128i a, __m128i b)
VPMULHUW __m256i _mm256_mulhi_epu16 ( __m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
PMULHW—Multiply Packed Signed Integers and Store High Result

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E5 /r(^1) PMULHW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Multiply the packed signed word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.</td>
</tr>
<tr>
<td>66 0F E5 /r PMULHW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG E5 /r VPMULHW xmm1 [[k1][z]], xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG E5 /r VPMULHW ymm1 [[k1][z]], ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG E5 /r VPMULHW zmm1 [[k1][z]], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

Op/En | Tuple Type | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
PMULHW—Multiply Packed Signed Integers and Store High Result

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHW (with 64-bit Operands)

\[
\begin{align*}
\text{TEMP0}[31:0] & := \text{DEST}[15:0] \ast \text{SRC}[15:0]; (* \text{Signed multiplication} *) \\
\text{TEMP1}[31:0] & := \text{DEST}[31:16] \ast \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] & := \text{DEST}[47:32] \ast \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] & := \text{DEST}[63:48] \ast \text{SRC}[63:48]; \\
\text{DEST}[15:0] & := \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] & := \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] & := \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] & := \text{TEMP3}[31:16]; \\
\end{align*}
\]

PMULHW (with 128-bit Operands)

\[
\begin{align*}
\text{TEMP0}[31:0] & := \text{DEST}[15:0] \ast \text{SRC}[15:0]; (* \text{Signed multiplication} *) \\
\text{TEMP1}[31:0] & := \text{DEST}[31:16] \ast \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] & := \text{DEST}[47:32] \ast \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] & := \text{DEST}[63:48] \ast \text{SRC}[63:48]; \\
\text{TEMP4}[31:0] & := \text{DEST}[79:64] \ast \text{SRC}[79:64]; \\
\text{TEMP5}[31:0] & := \text{DEST}[95:80] \ast \text{SRC}[95:80]; \\
\text{TEMP6}[31:0] & := \text{DEST}[111:96] \ast \text{SRC}[111:96]; \\
\text{TEMP7}[31:0] & := \text{DEST}[127:112] \ast \text{SRC}[127:112]; \\
\text{DEST}[15:0] & := \text{TEMP0}[31:16]; \\
\text{DEST}[31:16] & := \text{TEMP1}[31:16]; \\
\text{DEST}[47:32] & := \text{TEMP2}[31:16]; \\
\text{DEST}[63:48] & := \text{TEMP3}[31:16]; \\
\text{DEST}[79:64] & := \text{TEMP4}[31:16]; \\
\text{DEST}[95:80] & := \text{TEMP5}[31:16]; \\
\text{DEST}[111:96] & := \text{TEMP6}[31:16]; \\
\text{DEST}[127:112] & := \text{TEMP7}[31:16]; \\
\end{align*}
\]

VPMULHW (VEX.128 Encoded Version)

\[
\begin{align*}
\text{TEMP0}[31:0] & := \text{SRC1}[15:0] \ast \text{SRC2}[15:0]; (* \text{Signed Multiplication} *) \\
\text{TEMP1}[31:0] & := \text{SRC1}[31:16] \ast \text{SRC2}[31:16] \\
\text{TEMP2}[31:0] & := \text{SRC1}[47:32] \ast \text{SRC2}[47:32] \\
\text{TEMP3}[31:0] & := \text{SRC1}[63:48] \ast \text{SRC2}[63:48] \\
\text{TEMP4}[31:0] & := \text{SRC1}[79:64] \ast \text{SRC2}[79:64] \\
\text{TEMP5}[31:0] & := \text{SRC1}[95:80] \ast \text{SRC2}[95:80] \\
\text{TEMP6}[31:0] & := \text{SRC1}[111:96] \ast \text{SRC2}[111:96] \\
\text{TEMP7}[31:0] & := \text{SRC1}[127:112] \ast \text{SRC2}[127:112] \\
\text{DEST}[15:0] & := \text{TEMP0}[31:16] \\
\text{DEST}[31:16] & := \text{TEMP1}[31:16] \\
\text{DEST}[47:32] & := \text{TEMP2}[31:16] \\
\text{DEST}[63:48] & := \text{TEMP3}[31:16] \\
\text{DEST}[79:64] & := \text{TEMP4}[31:16] \\
\text{DEST}[95:80] & := \text{TEMP5}[31:16] \\
\end{align*}
\]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[MAXVL-1:128] := 0

PMULhw (VEX.256 Encoded Version)
TEMP0[31:0] := SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP4[31:0] := SRC1[79:64] * SRC2[79:64]
TEMP6[31:0] := SRC1[111:96] * SRC2[111:96]
DEST[15:0] := TEMP0[31:16]
DEST[31:16] := TEMP1[31:16]
DEST[63:48] := TEMP3[31:16]
DEST[79:64] := TEMP4[31:16]
DEST[95:80] := TEMP5[31:16]
DEST[111:96] := TEMP6[31:16]
DEST[127:112] := TEMP7[31:16]
DEST[143:128] := TEMP8[31:16]
DEST[159:144] := TEMP9[31:16]
DEST[175:160] := TEMP10[31:16]
DEST[MAXVL-1:256] := 0
PMULHW (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
      DEST[i+15:i] := tmp[31:16]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW __m512i __mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i __mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i __mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i __mm256_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i __mm256_mask_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i __mm128_mask_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i __mm128_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHWren__m64__mm64_mulhi_epi16(__m64 m1, __m64 m2)
(V)PMULHW __m128i __mm128_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW __m256i __mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 40 /r PMULLD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VLA VX512DQ</td>
<td>Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.
EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**VPMULLQ (EVEX Encoded Versions)**

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

FOR $j := 0$ TO $KL-1$

\[ i := j * 64 \]

IF $k1[j]$ OR "*no writemask*" THEN

IF $(EVEX.b = 1)$ AND (SRC2 "is memory")


$DEST[i+63:i] := Temp[63:0]$ \nELSE

IF "merging-masking" ; merging-masking

THEN "$DEST[i+63:i] remains unchanged*"

ELSE ; zeroing-masking

$DEST[i+63:i] := 0$

FI

ENDFOR

$DEST[MAXVL-1:VL] := 0$

**VPMULLD (EVEX Encoded Versions)**

$(KL, VL) = (4, 128), (8, 256), (16, 512)$

FOR $j := 0$ TO $KL-1$

\[ i := j * 32 \]

IF $k1[j]$ OR "*no writemask*" THEN

IF $(EVEX.b = 1)$ AND (SRC2 "is memory")


$DEST[i+31:i] := Temp[31:0]$ \nELSE

IF "merging-masking" ; merging-masking

THEN "$DEST[i+31:i] remains unchanged*"

ELSE ; zeroing-masking

$DEST[i+31:i] := 0$

FI

ENDFOR

$DEST[MAXVL-1:VL] := 0$
VPMULLD (VEX.256 Encoded Version)
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]

DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[159:128] := Temp4[31:0]
DEST[191:160] := Temp5[31:0]
DEST[223:192] := Temp6[31:0]
DEST[255:224] := Temp7[31:0]
DEST[MAXVL-1:256] := 0

VPMULLD (VEX.128 Encoded Version)
Temp0[63:0] := SRC1[31:0] * SRC2[31:0]
Temp2[63:0] := SRC1[95:64] * SRC2[95:64]

DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] := 0

PMULLD (128-bit Legacy SSE Version)
Temp0[63:0] := DEST[31:0] * SRC[31:0]
Temp2[63:0] := DEST[95:64] * SRC[95:64]

DEST[31:0] := Temp0[31:0]
DEST[63:32] := Temp1[31:0]
DEST[95:64] := Temp2[31:0]
DEST[127:96] := Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLQ __m512i _mm512_mullo_epi64(__m512i a, __m512i b);
VPMULLQ __m512i _mm512_mask_mullo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULLQ __m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64(__m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."
**PMULLW—Multiply Packed Signed Integers and Store Low Result**

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D5 /r¹ PMULLW mm, mm/m64</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Multiply the packed signed word integers in mm1 register and mm2/m64, and store the low 16 bits of the results in mm1.</td>
</tr>
<tr>
<td>66 0F D5 /r PMULLW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D5 /r VPMULLW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG D5 /r VPMULLW xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG D5 /r VPMULLW ymm1 [k1][z], ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG D5 /r VPMULLW zmm1 [k1][z], zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg  (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation**

**PMULLW (with 64-bit Operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] &= \text{DEST}[15:0] \times \text{SRC}[15:0]; \\
\text{TEMP1}[31:0] &= \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] &= \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] &= \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{DEST}[15:0] &= \text{TEMP0}[15:0]; \\
\text{DEST}[31:16] &= \text{TEMP1}[15:0]; \\
\text{DEST}[47:32] &= \text{TEMP2}[15:0]; \\
\text{DEST}[63:48] &= \text{TEMP3}[15:0];
\end{align*}
\]

**PMULLW (with 128-bit Operands)**

\[
\begin{align*}
\text{TEMP0}[31:0] &= \text{DEST}[15:0] \times \text{SRC}[15:0]; \\
\text{TEMP1}[31:0] &= \text{DEST}[31:16] \times \text{SRC}[31:16]; \\
\text{TEMP2}[31:0] &= \text{DEST}[47:32] \times \text{SRC}[47:32]; \\
\text{TEMP3}[31:0] &= \text{DEST}[63:48] \times \text{SRC}[63:48]; \\
\text{TEMP4}[31:0] &= \text{DEST}[79:64] \times \text{SRC}[79:64]; \\
\text{TEMP5}[31:0] &= \text{DEST}[95:80] \times \text{SRC}[95:80]; \\
\text{TEMP6}[31:0] &= \text{DEST}[111:96] \times \text{SRC}[111:96]; \\
\text{TEMP7}[31:0] &= \text{DEST}[127:112] \times \text{SRC}[127:112]; \\
\text{DEST}[15:0] &= \text{TEMP0}[15:0]; \\
\text{DEST}[31:16] &= \text{TEMP1}[15:0]; \\
\text{DEST}[47:32] &= \text{TEMP2}[15:0]; \\
\text{DEST}[63:48] &= \text{TEMP3}[15:0]; \\
\text{DEST}[79:64] &= \text{TEMP4}[15:0]; \\
\text{DEST}[95:80] &= \text{TEMP5}[15:0]; \\
\text{DEST}[111:96] &= \text{TEMP6}[15:0]; \\
\text{DEST}[127:112] &= \text{TEMP7}[15:0]; \\
\text{DEST}[\text{MAXVL}-1:256] &= 0
\end{align*}
\]
**VPMULLW (VEX.128 Encoded Version)**

Temp0[31:0] := SRC1[15:0] * SRC2[15:0]
Temp4[31:0] := SRC1[79:64] * SRC2[79:64]
Temp6[31:0] := SRC1[111:96] * SRC2[111:96]

DEST[15:0] := Temp0[15:0]
DEST[31:16] := Temp1[15:0]
DEST[47:32] := Temp2[15:0]
DEST[63:48] := Temp3[15:0]
DEST[79:64] := Temp4[15:0]
DEST[95:80] := Temp5[15:0]
DEST[111:96] := Temp6[15:0]
DEST[127:112] := Temp7[15:0]
DEST[MAXVL-1:128] := 0

**PMULLW (EVEX Encoded Versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN
      temp[31:0] := SRC1[i+15:i] * SRC2[i+15:i]
      DEST[i+15:i] := temp[15:0]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16( __mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mullo_epi16(_ __m256i a, _ __m256i b);
VPMULLW __m256i _mm256_mask_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16( __mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm128_mullo_epi16(_ __m128i a, _ __m128i b);
VPMULLW __m128i _mm128_maskz_mullo_epi16( __mmask8 k, __m128i a, __m128i b);
PMULLW __m64 __mm_mullo_epi16(__m64 m1, __m64 m2)
(V)PMULLW __m128i _mm_mullo_epi16 ( __m128i a, __m128i b)
VPMULLW __m256i _mm256_mullo_epi16 ( __m256i a, __m256i b);

**Flags Affected**

None.
SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."
PMULUDQ—Multiply Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F F4 /r1 PMULUDQ mm1, mm2/m64</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.</td>
<td></td>
</tr>
<tr>
<td>66 0F F4 /r PMULUDQ xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 F4 /r VPMULUDQ xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 F4 /r VPMULUDQ ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 F4 /r VPMULUDQ zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned
quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth, and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth, and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth, and seventh doublewords of a YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

**Operation**

**PMULUDQ (With 64-Bit Operands)**

\[
\text{DEST}[63:0] := \text{DEST}[31:0] \times \text{SRC}[31:0];
\]

**PMULUDQ (With 128-Bit Operands)**

\[
\begin{align*}
\text{DEST}[63:0] & := \text{DEST}[31:0] \times \text{SRC}[31:0]; \\
\text{DEST}[127:64] & := \text{DEST}[95:64] \times \text{SRC}[95:64];
\end{align*}
\]

**VPMULUDQ (VEX.128 Encoded Version)**

\[
\begin{align*}
\text{DEST}[63:0] & := \text{SRC1}[31:0] \times \text{SRC2}[31:0] \\
\text{DEST}[127:64] & := \text{DEST}[95:64] \times \text{SRC2}[95:64] \\
\text{DEST}[\text{MAXVL}-1:128] & := 0
\end{align*}
\]

**VPMULUDQ (VEX.256 Encoded Version)**

\[
\begin{align*}
\text{DEST}[63:0] & := \text{SRC1}[31:0] \times \text{SRC2}[31:0] \\
\text{DEST}[127:64] & := \text{SRC1}[95:64] \times \text{SRC2}[95:64] \\
\text{DEST}[191:128] & := \text{SRC1}[159:128] \times \text{SRC2}[159:128] \\
\text{DEST}[255:192] & := \text{SRC1}[223:192] \times \text{SRC2}[223:192] \\
\text{DEST}[\text{MAXVL}-1:256] & := 0
\end{align*}
\]
VPMULUDQ (EVEX Encoded Versions)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[
\text{FOR } j := 0 \text{ TO KL-1}
\]
\[
i := j \times 64
\]

\[\text{IF } k1[j] \text{ OR *no writemask* THEN}
\]
\[\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)}
\]
\[\text{THEN DEST}[i+63:i] := \text{ZeroExtend64}(\text{SRC1}[i+31:i]) \times \text{ZeroExtend64}(\text{SRC2}[31:0])
\]
\[\text{ELSE DEST}[i+63:i] := \text{ZeroExtend64}(\text{SRC1}[i+31:i]) \times \text{ZeroExtend64}(\text{SRC2}[i+31:i])
\]
\[\text{FI};
\]
\[\text{ELSE}
\]
\[\text{IF *merging-masking* ; merging-masking}
\]
\[\text{THEN *DEST}[i+63:i] \text{ remains unchanged*}
\]
\[\text{ELSE *zeroing-masking* ; zeroing-masking}
\]
\[\text{DEST}[i+63:i] := 0
\]
\[\text{FI}
\]
\[\text{FI};
\]
\[\text{ENDFOR}
\]
\[\text{DEST[MAXVL-1:VL]} := 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VPMULUDQ } __m512i \_mm512\_mul\_epu32(__m512i a, __m512i b);
\]
\[
\text{VPMULUDQ } __m512i \_mm512\_mask\_mul\_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
\]
\[
\text{VPMULUDQ } __m512i \_mm512\_maskz\_mul\_epu32(__mmask8 k, __m512i a, __m512i b);
\]
\[
\text{VPMULUDQ } __m256i \_mm256\_mul\_epu32(__m256i a, __m256i b);
\]
\[
\text{VPMULUDQ } __m256i \_mm256\_mask\_mul\_epu32(__mmask8 k, __m256i a, __m256i b);
\]
\[
\text{VPMULUDQ } __m256i \_mm256\_maskz\_mul\_epu32(__mmask8 k, __m256i a, __m256i b);
\]
\[
\text{VPMULUDQ } __m128i \_mm128\_mul\_epu32(__m128i a, __m128i b);
\]
\[
\text{VPMULUDQ } __m128i \_mm128\_mask\_mul\_epu32(__mmask8 k, __m128i a, __m128i b);
\]
\[
\text{VPMULUDQ } __m128i \_mm128\_maskz\_mul\_epu32(__mmask8 k, __m128i a, __m128i b);
\]
\[
\text{PMULUDQ } __m64 \_mm\_mul\_su32 (__m64 a, __m64 b)
\]
\[
(V)\text{PMULUDQ } __m128i \_mm\_mul\_epu32 (__m128i a, __m128i b)
\]
\[
\text{VPMULUDQ } __m256i \_mm256\_mul\_epu32(__m256i a, __m256i b);
\]

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
POP—Pop a Value From the Stack

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8F /0</td>
<td>POP r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into m16; increment stack pointer.</td>
</tr>
<tr>
<td>8F /0</td>
<td>POP r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into m32; increment stack pointer.</td>
</tr>
<tr>
<td>8F /0</td>
<td>POP r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into m64; increment stack pointer. Cannot encode 32-bit operand size.</td>
</tr>
<tr>
<td>58+ rw</td>
<td>POP r16</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into r16; increment stack pointer.</td>
</tr>
<tr>
<td>58+ rd</td>
<td>POP r32</td>
<td>O</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into r32; increment stack pointer.</td>
</tr>
<tr>
<td>58+ rd</td>
<td>POP r64</td>
<td>O</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into r64; increment stack pointer. Cannot encode 32-bit operand size.</td>
</tr>
<tr>
<td>1F</td>
<td>POP DS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop top of stack into DS; increment stack pointer.</td>
</tr>
<tr>
<td>07</td>
<td>POP ES</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop top of stack into ES; increment stack pointer.</td>
</tr>
<tr>
<td>17</td>
<td>POP SS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop top of stack into SS; increment stack pointer.</td>
</tr>
<tr>
<td>0F A1</td>
<td>POP FS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into FS; increment stack pointer by 16 bits.</td>
</tr>
<tr>
<td>0F A1</td>
<td>POP FS</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into FS; increment stack pointer by 32 bits.</td>
</tr>
<tr>
<td>0F A1</td>
<td>POP FS</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into FS; increment stack pointer by 64 bits.</td>
</tr>
<tr>
<td>0F A9</td>
<td>POP GS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Pop top of stack into GS; increment stack pointer by 16 bits.</td>
</tr>
<tr>
<td>0F A9</td>
<td>POP GS</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Pop top of stack into GS; increment stack pointer by 32 bits.</td>
</tr>
<tr>
<td>0F A9</td>
<td>POP GS</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Pop top of stack into GS; increment stack pointer by 64 bits.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMr/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- **Address size.** The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
  
  The address size is used only when writing to a destination operand in memory.

- **Operand size.** The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).
  
  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- **Stack-address size.** Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.
  
  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automat-
ically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

Loading the SS register with a POP instruction suppresses or inhibits some debug exceptions and inhibits interrupts on the following instruction boundary. (The inhibition ends after delivery of an exception or the execution of the next instruction.) This behavior allows a stack pointer to be loaded into the ESP register with the next instruction (POP ESP) before an event can be delivered. See Section 6.8.3, "Masking Exceptions and Interrupts When Switching Stacks," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. Intel recommends that software use the LSS instruction to load the SS register and ESP together.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST := SS:ESP; (* Copy a doubleword *)
        ESP := ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST := SS:ESP; (* Copy a word *)
        ESP := ESP + 2;
      FI;
    ELSE IF StackAddrSize = 64
      THEN
        IF OperandSize = 64
          THEN
            DEST := SS:RSP; (* Copy a quadword *)
            RSP := RSP + 8;
          ELSE (* OperandSize = 16*)
            DEST := SS:RSP; (* Copy a word *)
            RSP := RSP + 2;
          FI;
        FI;
    ELSE StackAddrSize = 16
      THEN
        IF OperandSize = 16
          THEN
            DEST := SS:SP; (* Copy a word *)
            SP := SP + 2;
        FI;
      ELSE StackAddrSize = 16
        THEN
          IF OperandSize = 16
            THEN
              DEST := SS:SP; (* Copy a word *)
              SP := SP + 2;
          FI;
        FI;
      FI;
    FI;
  FI;
ELSE StackAddrSize = 16
  THEN
    IF OperandSize = 16
      THEN
        DEST := SS:SP; (* Copy a word *)
        SP := SP + 2;
      FI;
    FI;
  FI;
ELSE StackAddrSize = 16
  THEN
    IF OperandSize = 16
      THEN
        DEST := SS:SP; (* Copy a word *)
        SP := SP + 2;
      FI;
    FI;
  FI;
ELSE StackAddrSize = 16
  THEN
    IF OperandSize = 16
      THEN
        DEST := SS:SP; (* Copy a word *)
        SP := SP + 2;
      FI;
    FI;
  FI;
ELSE (* OperandSize = 32 *)
    DEST := SS:SP; (* Copy a doubleword *)
    SP := SP + 4;
    FI;
    FI;

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

64-BIT_MODE
IF FS, or GS is loaded with non-NULL selector;
    THEN
        IF segment selector index is outside descriptor table limits
            OR segment is not a data or readable code segment
            OR ((segment is a data or nonconforming code segment)
                AND ((RPL > DPL) or (CPL > DPL))
                THEN #GP(selector);
            IF segment not marked present
            THEN #NP(selector);
        ELSE
            SegmentRegister := segment selector;
            SegmentRegister := segment descriptor;
            FI;
        FI;
    IF FS, or GS is loaded with a NULL selector;
        THEN
            SegmentRegister := segment selector;
            SegmentRegister := segment descriptor;
        FI;

PREPROTECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded;
    THEN
        IF segment selector is NULL
            THEN #GP(0);
            FI;
        IF segment selector index is outside descriptor table limits
            or segment selector’s RPL ≠ CPL
            or segment is not a writable data segment
            or DPL ≠ CPL
            THEN #GP(selector);
            FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS := segment selector;
                SS := segment descriptor;
            FI;
            FI;
    IF DS, ES, FS, or GS is loaded with non-NULL selector;
        THEN
IF segment selector index is outside descriptor table limits
  or segment is not a data or readable code segment
  or ((segment is a data or nonconforming code segment)
      and ((RPL > DPL) or (CPL > DPL)))
    THEN #GP(selector);
  FI;
IF segment not marked present
  THEN #NP(selector);
ELSE
  SegmentRegister := segment selector;
  SegmentRegister := segment descriptor;
  FI;
FI;
IF DS, ES, FS, or GS is loaded with a NULL selector
  THEN
    SegmentRegister := segment selector;
    SegmentRegister := segment descriptor;
  FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If attempt is made to load SS register with NULL segment selector.
      If the destination operand is in a non-writable segment.
      If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
      If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment
      selector.
#GP(selector) If segment selector index is outside descriptor table limits.
      If the SS register is being loaded and the segment selector’s RPL and the segment descriptor’s
      DPL are not equal to the CPL.
      If the SS register is being loaded and the segment pointed to is a
      non-writable data segment.
      If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or
      readable code segment.
      If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or
      nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0) If the current top of stack is not within the stack segment.
      If a memory operand effective address is outside the SS segment limit.
#SS(selector) If the SS register is being loaded and the segment pointed to is marked not present.
#NP If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not
      present.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment
      checking is enabled.
#UD If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If the stack address is in a non-canonical form.
#GP(selector) If the descriptor is outside the descriptor table limit.
   If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
   If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code) If a page fault occurs.
#NP If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD If the LOCK prefix is used.
If the DS, ES, or SS register is being loaded.
POPA/POPAD—Pop All General-Purpose Registers

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>POPA</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop DI, SI, BP, BX, DX, CX, and AX.</td>
</tr>
<tr>
<td>61</td>
<td>POPAD</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

POPA/POPAD pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). These instructions reverse the operation of the PUSHA/PUSHAD instructions. The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment’s segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

Operation

IF 64-Bit Mode
THEN
  #UD;
ELSE
  IF OperandSize = 32 (* Instruction = POPAD *)
  THEN
    EDI := Pop();
    ESI := Pop();
    EBP := Pop();
    Increment ESP by 4; (* Skip next 4 bytes of stack *)
    EBX := Pop();
    EDX := Pop();
    ECX := Pop();
    EAX := Pop();
  ELSE (* OperandSize = 16, instruction = POPA *)
    DI := Pop();
    SI := Pop();
    BP := Pop();
    Increment ESP by 2; (* Skip next 2 bytes of stack *)
    BX := Pop();
    DX := Pop();
    CX := Pop();
    AX := Pop();
  FI;
FI;
Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the starting or ending stack address is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#SS If the starting or ending stack address is not within the stack segment.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#SS(0) If the starting or ending stack address is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
## POPCNT—Return the Count of Number of Bits Set to 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F B8 lr</td>
<td>POPCNT r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>POPCNT on r/m16</td>
</tr>
<tr>
<td>F3 0F B8 lr</td>
<td>POPCNT r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>POPCNT on r/m32</td>
</tr>
<tr>
<td>F3 REX.W 0F B8 lr</td>
<td>POPCNT r64, r/m64</td>
<td>RM</td>
<td>N.E.</td>
<td>POPCNT on r/m64</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

### Operation

1. `Count = 0;`
2. For \(i = 0; i < \text{OperandSize}; i++\)
   - IF \(\text{SRC}[i] = 1\) // i'th bit
     - THEN \(\text{Count}++; \text{F1;}\)
3. \(\text{DEST} := \text{Count};\)

### Flags Affected

- OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if \(\text{SRC} = 0\), otherwise ZF is cleared.

### Intel C/C++ Compiler Intrinsic Equivalent

- `POPCNT int _mm_popcnt_u32(unsigned int a);`
- `POPCNT int64_t _mm_popcnt_u64(unsigned __int64 a);`

### Protected Mode Exceptions

- [#AC(0)](https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-32-bit-architecture-manual.pdf) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
- [#UD] If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
  - If LOCK prefix is used.

### Real-Address Mode Exceptions

- [#GP(0)](https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-32-bit-architecture-manual.pdf) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
- [#UD] If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
  - If LOCK prefix is used.
Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
If LOCK prefix is used.
POPF/POPFD/POPFQ—Pop Stack Into EFLAGS Register

**Opcode** | **Instruction** | **Op/En** | **64-Bit Mode** | **Compat/Leg Mode** | **Description**
--- | --- | --- | --- | --- | ---
9D | POPF | ZO | Valid | Valid | Pop top of stack into lower 16 bits of EFLAGS.
9D | POPFD | ZO | N.E. | Valid | Pop top of stack into EFLAGS.
9D | POPFQ | ZO | Valid | N.E. | Pop top of stack and zero-extend into RFLAGS.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHFQ instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-16 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF1, VIP, VIF, and VM may be modified. VIP, VIF, and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

See Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the EFLAGS registers.

---

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.
Table 4-16. Effect of POPF/POPFQ on the EFLAGS Register

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operand Size</th>
<th>CPL</th>
<th>IOPL</th>
<th>Flags</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-Address Mode (CR0.PE = 0)</td>
<td>16</td>
<td>0</td>
<td>0-3</td>
<td>N  N  N  N  N  N  0  S  S  S  S  S  S  S  S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0</td>
<td>0-3</td>
<td>S  N  N  S  N  N  0  S  S  S  S  S  S  S  S</td>
<td></td>
</tr>
<tr>
<td>Protected, Compatibility, and 64-Bit Modes (CR0.PE = 1 EFLAGS.VM = 0)</td>
<td>16</td>
<td>0</td>
<td>0-3</td>
<td>N  N  N  N  N  N  0  S  S  S  S  S  S  S  S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1-3</td>
<td>&lt;CPL</td>
<td>N  N  N  N  N  N  0  S  N  S  S  N  S  S  S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1-3</td>
<td>≥CPL</td>
<td>N  N  N  N  N  N  0  S  N  S  S  N  S  S  S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32, 64</td>
<td>0</td>
<td>0-3</td>
<td>S  N  N  S  N  N  0  S  S  S  S  S  S  S  S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32, 64</td>
<td>1-3</td>
<td>≤CPL</td>
<td>S  N  N  S  N  N  0  S  N  S  S  N  S  S  S</td>
<td></td>
</tr>
<tr>
<td>Virtual-8086 (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 0)</td>
<td>16</td>
<td>3</td>
<td>0-2</td>
<td>X  X  X  X  X  X  X  X  X  X  X  X  X  X  X</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>3</td>
<td>0-2</td>
<td>X  X  X  X  X  X  X  X  X  X  X  X  X  X  X</td>
<td>1</td>
</tr>
<tr>
<td>VME (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 1)</td>
<td>16</td>
<td>3</td>
<td>0-2</td>
<td>N/ X  X  X  X  X  X  X  X  X  X  X  X  X  X</td>
<td>2, 3</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>N  N  N  N  N  N  0  S  N  S  S  S  S  S  S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>3</td>
<td>3</td>
<td>S  N  N  S  N  N  0  S  N  S  S  S  S  S  S</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Updated from stack</td>
</tr>
<tr>
<td>SV</td>
<td>Updated from IF (bit 9) in FLAGS value on stack</td>
</tr>
<tr>
<td>N</td>
<td>No change in value</td>
</tr>
<tr>
<td>X</td>
<td>No EFLAGS update</td>
</tr>
<tr>
<td>0</td>
<td>Value is cleared</td>
</tr>
</tbody>
</table>

Operation

IF EFLAGS.VM = 0 (* Not in Virtual-8086 Mode *)
THEN IF CPL = 0 OR CR0.PE = 0
THEN
IF OperandSize = 32;
THEN
EFLAGS := Pop(); (* 32-bit pop *)
(* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
ELSE IF (Operandsize = 64)
RFLAGS = Pop(); (* 64-bit pop *)
(* All non-reserved flags except RF, VIP, VIF, and VM can be modified; VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
ELSE (* Operandsize = 16 *)
  EFLAGS[15:0] := Pop(); (* 16-bit pop *)
  (* All non-reserved flags can be modified. *)
FI;
ELSE (* CPL > 0 *)
  IF Operandsize = 32
    THEN
      IF CPL > IOPL
        THEN
          EFLAGS := Pop(); (* 32-bit pop *)
          (* All non-reserved bits except IF, IOPL, VIP, VIF, VM, and RF can be modified;
           IF, IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
          ELSE
            EFLAGS := Pop(); (* 32-bit pop *)
            (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
             IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
        FI;
      ELSE IF (Operandsize = 64)
        IF CPL > IOPL
          THEN
            RFLAGS := Pop(); (* 64-bit pop *)
            (* All non-reserved bits except IF, IOPL, VIP, VIF, VM, and RF can be modified;
             IF, IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
          ELSE
            RFLAGS := Pop(); (* 64-bit pop *)
            (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
             IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
        FI;
      ELSE (* Operandsize = 16 *)
        EFLAGS[15:0] := Pop(); (* 16-bit pop *)
        (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
    FI;
  ELSE IF (Operandsize = 64)
    IF CPL > IOPL
      THEN
        RFLAGS := Pop(); (* 64-bit pop *)
        (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
         IF, IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
      ELSE
        RFLAGS := Pop(); (* 64-bit pop *)
        (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
         IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
    FI;
  ELSE (* Operandsize = 16 *)
    EFLAGS[15:0] := Pop(); (* 16-bit pop *)
    (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
  FI;
ELSE (* In virtual-8086 mode *)
  IF IOPL = 3
    THEN
      IF Operandsize = 32
        THEN
          EFLAGS := Pop(); (* 32-bit pop *)
          (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
           IF, IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
        ELSE
          EFLAGS := Pop(); (* 32-bit pop *)
          (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
           IOPL, VIP, VIF, VM, and all reserved bits are unaffected; RF is cleared. *)
      FI;
    ELSE (* IOPL < 3 *)
      IF Operandsize = 32 OR (CR4.VME = 0)
        THEN #GP(0); (* Trap to virtual-8086 monitor. *)
      ELSE (* Operandsize = 16 and CR4.VME = 1 *)
        tempFLAGS := Pop();
        IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
          THEN #GP(0);
        ELSE
POPF/POPFD/POPFQ—Pop Stack Into EFLAGS Register

EFLAGS.VIF := tempFLAGS[9];
EFLAGS[15:0] := tempFLAGS;
(* All non-reserved bits except IOPL and IF can be modified;
IOPL, IF, and all reserved bits are unaffected. *)

Flags Affected
All flags may be affected; see the Operation section for details.

Protected Mode Exceptions
#SS(0) If the top of stack is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#SS If the top of stack is not within the stack segment.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If IOPL < 3 and VME is not enabled.
If IOPL < 3 and the 32-bit operand size is used.
If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack.
If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack.
If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0) If the top of stack is not within the stack segment.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same as for protected mode exceptions.

64-Bit Mode Exceptions
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
## POR—Bitwise Logical OR

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EB /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Bitwise OR of mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F EB /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Bitwise OR of xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG EB /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Bitwise OR of xmm2/m128 and xmm3.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG EB /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Bitwise OR of ymm2/m256 and ymm3.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 EB /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 EB /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 EB /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 EB /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 EB /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 EB /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

### NOTES:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).
Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

**Operation**

**POR (64-bit Operand)**

\[
\text{DEST} := \text{DEST OR SRC}
\]

**POR (128-bit Legacy SSE Version)**

\[
\text{DEST} := \text{DEST OR SRC}
\]

\[
\text{DEST}[\text{MAXVL-1:128}] := \text{Unmodified}
\]

**VPOR (VEX.128 Encoded Version)**

\[
\text{DEST} := \text{SRC1 OR SRC2}
\]

\[
\text{DEST}[\text{MAXVL-1:128}] := 0
\]

**VPOR (VEX.256 Encoded Version)**

\[
\text{DEST} := \text{SRC1 OR SRC2}
\]

\[
\text{DEST}[\text{MAXVL-1:256}] := 0
\]

**VPORD (EVEX Encoded Versions)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0 \text{ TO } KL-1\)

\[i := j \times 32\]

IF \(k1[j] \text{ OR *no writemask* THEN}\)

IF \((\text{EVEX.b} = 1) \text{ AND (SRC2 *is memory*)}\)

THEN\(\text{DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE OR SRC2}[31:0]\)

ELSE\(\text{DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE OR SRC2}[i+31:i]\)

FI;

ELSE

IF *merging-masking* ; merging-masking

*\text{DEST}[i+31:i] \text{ remains unchanged*}

ELSE ; zeroing-masking

\(\text{DEST}[i+31:i] := 0\)

FI;

ENDIF

ENDFOR;

\[\text{DEST}[\text{MAXVL-1:VL}] := 0\]
**Intel C/C++ Compiler Intrinsic Equivalent**

```c
VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32(__mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORD __m256i _mm256_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b);
VPORD __m64 _mm_or_si64(__m64 m1, __m64 m2);
```

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."
PREFETCHh—Prefetch Data Into Caches

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 18 /1</td>
<td>PREFETCHT0 m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T0 hint.</td>
</tr>
<tr>
<td>0F 18 /2</td>
<td>PREFETCHT1 m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T1 hint.</td>
</tr>
<tr>
<td>0F 18 /3</td>
<td>PREFETCHT2 m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T2 hint.</td>
</tr>
<tr>
<td>0F 18 /0</td>
<td>PREFETCHNTA m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using NTA hint.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- **T0** (temporal data)—prefetch data into all levels of the cache hierarchy.
- **T1** (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- **T2** (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- **NTA** (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHh instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

FETCH (m8):
Intel C/C++ Compiler Intrinsic Equivalent

```c
void _mm_prefetch(char *p, int i)
```

The argument "p" gives the address of the byte (and corresponding cache line) to be prefetched. The value "i" gives a constant (_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, or _MM_HINT_NTA) that specifies the type of prefetch operation to be performed.

**Numeric Exceptions**

None.

**Exceptions (All Operating Modes)**

#UD       If the LOCK prefix is used.
PREFETCHW—Prefetch Data Into Caches in Anticipation of a Write

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 0D /1 PREFETCHW m8</td>
<td>M</td>
<td>V/V</td>
<td>PREFETCHW</td>
<td>Move data from m8 closer to the processor in anticipation of a write.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates other cached copies in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHW instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);

**Flags Affected**

None.

**C/C++ Compiler Intrinsic Equivalent**

void _m_prefetchw ( void *);

**Protected Mode Exceptions**

- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
### PSADBW—Compute Sum of Absolute Differences

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 66 /r&lt;sup&gt;1&lt;/sup&gt; PSADBW mm1, mm2/m64</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the absolute differences of the packed unsigned byte integers from mm2 to mm1; differences are then summed to produce an unsigned word integer result.</td>
</tr>
<tr>
<td>66 0F 66 /r PSADBW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Computes the absolute differences of the packed unsigned byte integers from xmm2 to xmm1; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the absolute differences of the packed unsigned byte integers from xmm3 to xmm2; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Computes the absolute differences of the packed unsigned byte integers from ymm3 to ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Computes the absolute differences of the packed unsigned byte integers from xmm3 to xmm2; then each consecutive 8 differences are summed separately to produce two unsigned word integer results.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Computes the absolute differences of the packed unsigned byte integers from ymm3 to ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Computes the absolute differences of the packed unsigned byte integers from zmm3 to zmm2; then each consecutive 8 differences are summed separately to produce eight unsigned word integer results.</td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
**Description**

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

---

**Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands**

```
<table>
<thead>
<tr>
<th>SRC</th>
<th>X7</th>
<th>X6</th>
<th>X5</th>
<th>X4</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEST</td>
<td>Y7</td>
<td>Y6</td>
<td>Y5</td>
<td>Y4</td>
<td>Y3</td>
<td>Y2</td>
<td>Y1</td>
<td>Y0</td>
</tr>
<tr>
<td>TEMP</td>
<td>ABS(X7:Y7)</td>
<td>ABS(X6:Y6)</td>
<td>ABS(X5:Y5)</td>
<td>ABS(X4:Y4)</td>
<td>ABS(X3:Y3)</td>
<td>ABS(X2:Y2)</td>
<td>ABS(X1:Y1)</td>
<td>ABS(X0:Y0)</td>
</tr>
<tr>
<td>DEST</td>
<td>00H</td>
<td>00H</td>
<td>00H</td>
<td>00H</td>
<td>00H</td>
<td>00H</td>
<td>SUM(TEMP?...TEMP0)</td>
<td></td>
</tr>
</tbody>
</table>
```
Operation

**VPSADBW (EVEX Encoded Versions)**

VL = 128, 256, 512
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 000000000000H

IF VL >= 256
(* Repeat operation for bytes 16 through 31*)
DEST[143:128] := SUM(TEMP16:TEMP23)
DEST[191:144] := 000000000000H
DEST[207:192] := SUM(TEMP24:TEMP31)
DEST[223:208] := 000000000000H
Fi;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
TEMP63 := ABS(SRC1[511:504] - SRC2[511:504])
DEST[271:256] := SUM(TEMP0:TEMP7)
DEST[319:272] := 000000000000H
DEST[335:320] := SUM(TEMP8:TEMP15)
DEST[447:400] := 000000000000H
DEST[463:448] := SUM(TEMP24:TEMP31)
DEST[511:464] := 000000000000H
Fi;
DEST[MAXVL-1:VL] := 0

**VPSADBW (VEX.256 Encoded Version)**

TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 000000000000H
DEST[143:128] := SUM(TEMP16:TEMP23)
DEST[191:144] := 000000000000H
DEST[207:192] := SUM(TEMP24:TEMP31)
DEST[223:208] := 000000000000H
DEST[MAXVL-1:256] := 0
VPSADBW (VEX.128 Encoded Version)
TEMP0 := ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 000000000000H
DEST[MAXVL-1:128] := 0

PSADBW (128-bit Legacy SSE Version)
TEMP0 := ABS(DST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 := ABS(DST[127:120] - SRC[127:120])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H
DEST[79:64] := SUM(TEMP8:TEMP15)
DEST[127:80] := 00000000000
DEST[MAXVL-1:128] (Unmodified)

PSADBW (64-bit Operand)
TEMP0 := ABS(DST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 := ABS(DST[63:56] - SRC[63:56])
DEST[15:0] := SUM(TEMP0:TEMP7)
DEST[63:16] := 000000000000H

Intel C/C++ Compiler Intrinsic Equivalent
VPSADBW __m512i _mm512_sad_epu8( __m512i a, __m512i b)
PSADBW __m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW __m256i _mm256_sad_epu8( __m256i a, __m256i b)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
PSHUFB—Packed Shuffle Bytes

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Description</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>ModRM:reg</th>
<th>VEX.vvvv</th>
<th>ModRM:r/m</th>
<th>VEX.vvvv</th>
<th>ModRM:r/m</th>
<th>N/A</th>
<th>N/A</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 00 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>mm1, mm2/m64</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>N/A</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>66 0F 38 00 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSSE3</td>
<td>xmm1, xmm2/m128</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 00 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>xmm1, xmm2, xmm3/m128</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>N/A</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 00 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>xmm1, xmm2, xmm3/m256</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.WIG 00 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>N/A</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.WIG 00 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>xmm1 [k1][z], ymm2, ymm3/m256</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.WIG 00 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>xmm1 [k1][z], zmm2, zmm3/m512</td>
<td>N/A</td>
<td>ModRM:reg</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask.

NOTES:

The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

**Operation**

**PSHUFB (With 64-bit Operands)**

```plaintext
TEMP := DEST
for i = 0 to 7 {
   if (SRC[(i * 8)+7] = 1 ) then
      DEST[(i*8)+7..(i*8)+0] := 0;
   else
      index[2..0] := SRC[(i*8)+2 .. (i*8)+0];
      DEST[(i*8)+7..(i*8)+0] := TEMP[(index*8+7)..(index*8+0)];
   endif;
}
```

**PSHUFB (with 128 bit operands)**

```plaintext
TEMP := DEST
for i = 0 to 15 {
   if (SRC[(i * 8)+7] = 1 ) then
      DEST[(i*8)+7..(i*8)+0] := 0;
   else
      index[3..0] := SRC[(i*8)+3 .. (i*8)+0];
      DEST[(i*8)+7..(i*8)+0] := TEMP[(index*8+7)..(index*8+0)];
   endif;
}
```

**VPSHUFB (VEX.128 Encoded Version)**

```plaintext
for i = 0 to 15 {
   if (SRC2[(i * 8)+7] = 1) then
      DEST[(i*8)+7..(i*8)+0] := 0;
   else
      index[3..0] := SRC2[(i*8)+3 .. (i*8)+0];
      DEST[(i*8)+7..(i*8)+0] := SRC1[(index*8+7)..(index*8+0)];
   endif;
}
```

**VPSHUFB (VEX.256 Encoded Version)**

```plaintext
for i = 0 to 15 {
   if (SRC2[(i * 8)+7] == 1) then
      DEST[(i*8)+7..(i*8)+0] := 0;
   else
      index[3..0] := SRC2[(i*8)+3 .. (i*8)+0];
      DEST[(i*8)+7..(i*8)+0] := SRC1[(index*8+7)..(index*8+0)];
   endif;
   if (SRC2[128 + (i * 8)+7] == 1 ) then
      DEST[128 + (i*8)+7..(i*8)+0] := 0;
   else
      index[3..0] := SRC2[128 + (i*8)+3 .. (i*8)+0];
      DEST[128 + (i*8)+7..(i*8)+0] := SRC1[128 + (index*8+7)..(index*8+0)];
   endif;
}
```

PSHUFB—Packed Shuffle Bytes
VPSHUFB (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

\[ j\text{mask} := (KL-1) & \sim 0xF \]

\[ \text{FOR } j = 0 \text{ TO } KL-1 \]

IF k[i] or no_masking

index := src.byte[ j ];

IF index & 0x80

Dest.byte[ j ] := 0;
ELSE

index := (index & 0xF) + (j & jmask); // 16-element in-lane lookup

Dest.byte[ j ] := src.byte[ index ];
ELSE if zeroing

Dest.byte[ j ] := 0;

DEST[MAXVL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VPSHUFB } & \_m512i \_mm512\_shuffle\_epi8(\_m512i \ a, \_m512i \ b); \\
\text{VPSHUFB } & \_m512i \_mm512\_mask\_shuffle\_epi8(\_m512i \ s, \_mmask64 \ k, \_m512i \ a, \_m512i \ b); \\
\text{VPSHUFB } & \_m512i \_mm512\_maskz\_shuffle\_epi8(\_mmask64 \ k, \_m512i \ a, \_m512i \ b); \\
\text{VPSHUFB } & \_m256i \_mm256\_mask\_shuffle\_epi8(\_mmask32 \ k, \_m256i \ a, \_m256i \ b); \\
\text{VPSHUFB } & \_m256i \_mm256\_maskz\_shuffle\_epi8(\_mmask32 \ k, \_m256i \ a, \_m256i \ b); \\
\text{VPSHUFB } & \_m128i \_mm128\_mask\_shuffle\_epi8(\_mmask16 \ k, \_m128i \ a, \_m128i \ b); \\
\text{VPSHUFB } & \_m128i \_mm128\_maskz\_shuffle\_epi8(\_mmask16 \ k, \_m128i \ a, \_m128i \ b); \\
\text{VPSHUFB } & \_mm64\_shuffle\_epi8(\_m64 \ a, \_m64 \ b) \\
\text{VPSHUFB } & \_m128i\_mm128\_shuffle\_epi8(\_m128i \ a, \_m128i \ b) \\
\text{VPSHUFB } & \_m256i\_mm256\_shuffle\_epi8(\_m256i \ a, \_m256i \ b)
\end{align*}
\]

SIMD Floating-Point Exceptions

None.
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
**PSHUFD—Shuffle Packed Doublewords**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 70 /r ib PSHUF D xmm1, xmm2/m128, imm8</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 70 /r ib VPShuf D xmm1, xmm2/m128, imm8</td>
<td>A V/V</td>
<td>AVX</td>
<td>Shuffle the doublewords in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 70 /r ib VPShuf D ymm1, ymm2/m256, imm8</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shuffle the doublewords in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 70 /r ib VPShufD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 70 /r ib VPShufD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 70 /r ib VPShufD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.
The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**PSHUFD (128-bit Legacy SSE Version)**

```
PSHUFD (128-bit Legacy SSE Version)
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] := 0
```

**VPSHUFD (VEX.128 Encoded Version)**

```
VPSHUFD (VEX.128 Encoded Version)
DEST[31:0] := (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[95:64] := (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] := (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] := 0
```
VPShufD (VEX.256 Encoded Version)
DEST[31:0] := (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[95:64] := (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[MAXVL-1:256] := 0

VPShufD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC is memory*)
    THEN TMP_SRC[i+31:i] := SRC[31:0]
    ELSE TMP_SRC[i+31:i] := SRC[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] := (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
  FI;
IF VL >= 256
  FI;
IF VL >= 512
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[i] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSHUFD __m512i __mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i __mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i __mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i __mm256_shuffle_epi32(__m256i a, int n);
VPSHUFD __m256i __mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i __mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
(V)PSHUFD __m128i __mm_shuffle_epi32(__m128i a, int n)
VPSHUFD __m256i __mm256_shuffle_epi32(__m256i a, const int n)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Table 2-50, "Type E4NF Class Exception Conditions."
Additionally:

#UD If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.
PSHUFHW—Shuffle Packed High Words

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 70 /r ib PSHUFHW xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.F3.0F:WIG 70 /r ib VPUSHUFHW xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.F3.0F:WIG 70 /r ib VPUSHUFHW ymm1, ymm2/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F:WIG 70 /r ib VPUSHUFHW xmm1 [k1][z]. xmm2/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F:WIG 70 /r ib VPUSHUFHW ymm1 [k1][z]. ymm2/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F:WIG 70 /r ib VPUSHUFHW zmm1 [k1][z]. zmm2/m512, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-line operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.
EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**PSHUFHW (128-bit Legacy SSE Version)**

DEST[63:0] := SRC[63:0]

DEST[79:64] := (SRC >> (imm[1:0] * 16))[79:64]


DEST[MAXVL-1:128] := 0

**VPSHUFHW (VEX.128 Encoded Version)**

DEST[63:0] := SRC1[63:0]

DEST[79:64] := (SRC1 >> (imm[1:0] * 16))[79:64]


DEST[MAXVL-1:128] := 0

**VPSHUFHW (VEX.256 Encoded Version)**

DEST[63:0] := SRC1[63:0]

DEST[79:64] := (SRC1 >> (imm[1:0] * 16))[79:64]


DEST[MAXVL-1:256] := 0

**VPSHUFHW (EVEX Encoded Versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

TMP_DEST[63:0] := SRC1[63:0]

TMP_DEST[79:64] := (SRC1 >> (imm[1:0] * 16))[79:64]


FI;

IF VL >= 256


FI;

IF VL >= 512


TMP_DEST[463:448] := (SRC1 >> (imm[1:0] * 16))[463:448]

FI;

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*
                    DEST[i+15:i] := 0
            FI
    FI
ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16(__mmask16 k, __m512i a, int n);
VPSHUFHW __m256i _mm256_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__mmask8 k, __m256i a, int n);
VPSHUFHW __m256i _mm256_maskz_shufflehi_epi16(__mmask8 k, __m256i a, int n);
(V)PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW __m256i _mm256_shufflehi_epi16(__m256i a, const int n)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
Additionally:

#UD If VEX.vvvv ! = 1111B, or EVEX.vvvv ! = 1111B.
PSHUFLW—Shuffle Packed Low Words

**Description**

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the write-mask.
Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**PSHUFLW (128-bit Legacy SSE Version)**

DEST[15:0] := (SRC >> (imm[1:0] * 16))[15:0]
DEST[127:64] := SRC[127:64]
DEST[MAXVL-1:128] := 0

**VPUSHUFLW (VEX.128 Encoded Version)**

DEST[15:0] := (SRC1 >> (imm[1:0] * 16))[15:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**VPUSHUFLW (VEX.256 Encoded Version)**

DEST[15:0] := (SRC1 >> (imm[1:0] * 16))[15:0]
DEST[127:64] := SRC1[127:64]
DEST[143:128] := SRC1[127:64]
DEST[191:176] := SRC1[127:64]
DEST[255:192] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**VPUSHUFLW (EVEX.U1.512 Encoded Version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

    TMP_DEST[15:0] := (SRC1 >> (imm[1:0] * 16))[15:0]
    TMP_DEST[127:64] := SRC1[127:64]

FI;

IF VL >= 256


FI;

IF VL >= 512

TMP_DEST[511:448] := SRC1[511:448]
FI;

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TMP_DEST[i+15:i];
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n);
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
Additionally:
#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.
PSHUFW—Shuffle Packed Words

**Description**
Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Operation**

\[
\begin{align*}
\text{DEST}[15:0] & := (\text{SRC} \gg (\text{ORDER}[1:0] \times 16))[15:0]; \\
\text{DEST}[31:16] & := (\text{SRC} \gg (\text{ORDER}[3:2] \times 16))[15:0]; \\
\text{DEST}[47:32] & := (\text{SRC} \gg (\text{ORDER}[5:4] \times 16))[15:0]; \\
\text{DEST}[63:48] & := (\text{SRC} \gg (\text{ORDER}[7:6] \times 16))[15:0];
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

PSHUFW __m64 _mm_shuffle_pi16(__m64 a, int n)

**Flags Affected**
None.

**Numeric Exceptions**
None.

**Other Exceptions**
### PSIGNB/PSIGNW/PSIGND—Packed SIGN

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 08 /r&lt;sup&gt;1&lt;/sup&gt; PSIGNB mm1, mm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed byte integers in mm1 depending on the corresponding sign in mm2/m64.</td>
</tr>
<tr>
<td>66 0F 38 08 /r PSIGNB xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed byte integers in xmm1 depending on the corresponding sign in xmm2/m128.</td>
</tr>
<tr>
<td>NP 0F 38 09 /r&lt;sup&gt;1&lt;/sup&gt; PSIGNW mm1, mm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed word integers in mm1 depending on the corresponding sign in mm2/m64.</td>
</tr>
<tr>
<td>66 0F 38 09 /r PSIGNW xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed word integers in xmm1 depending on the corresponding sign in xmm2/m128.</td>
</tr>
<tr>
<td>NP 0F 38 0A /r&lt;sup&gt;1&lt;/sup&gt; PSIGND mm1, mm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed doubleword integers in mm1 depending on the corresponding sign in mm2/m64.</td>
</tr>
<tr>
<td>66 0F 38 0A /r PSIGND xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSSE3</td>
<td>Negate/zero/preserve packed doubleword integers in xmm1 depending on the corresponding sign in xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 08 /r VPSIGNB xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Negate/zero/preserve packed byte integers in xmm2 depending on the corresponding sign in xmm3/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 09 /r VPSIGNW xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Negate/zero/preserve packed word integers in xmm2 depending on the corresponding sign in xmm3/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.WIG 0A /r VPSIGND xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Negate/zero/preserve packed doubleword integers in xmm2 depending on the corresponding sign in xmm3/m128.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 08 /r VPSIGNB ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Negate packed byte integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 09 /r VPSIGNW ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Negate packed 16-bit integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.WIG 0A /r VPSIGND ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Negate packed doubleword integers in ymm2 if the corresponding sign in ymm3/m256 is less than zero.</td>
</tr>
</tbody>
</table>

**NOTES:**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Operation

```python
def byte_sign(control, input_val):
    if control<0:
        return negate(input_val)
    elif control==0:
        return 0
    return input_val

def word_sign(control, input_val):
    if control<0:
        return negate(input_val)
    elif control==0:
        return 0
    return input_val

def dword_sign(control, input_val):
    if control<0:
        return negate(input_val)
    elif control==0:
        return 0
    return input_val

PSIGNB srcdest, src    // MMX 64-bit Operands
VL=64
KL := VL/8
for i in 0...KL-1:
    srcdest.byte[i] := byte_sign(src.byte[i], srcdest.byte[i])

PSIGNW srcdest, src    // MMX 64-bit Operands
VL=64
KL := VL/16
FOR i in 0...KL-1:
    srcdest.word[i] := word_sign(src.word[i], srcdest.word[i])
```

4-432 Vol. 2B
PSIGND srcdest, src // MMX 64-bit Operands
VL=64
KL := VL/32
FOR i in 0...KL-1:
    srcdest.dword[i] := dword_sign(src.dword[i], srcdest.dword[i])

PSIGNB srcdest, src // SSE 128-bit Operands
VL=128
KL := VL/8
FOR i in 0...KL-1:
    srcdest.byte[i] := byte_sign(src.byte[i], srcdest.byte[i])

PSIGNW srcdest, src // SSE 128-bit Operands
VL=128
KL := VL/16
FOR i in 0...KL-1:
    srcdest.word[i] := word_sign(src.word[i], srcdest.word[i])

PSIGND srcdest, src // SSE 128-bit Operands
VL=128
KL := VL/32
FOR i in 0...KL-1:
    srcdest.dword[i] := dword_sign(src.dword[i], srcdest.dword[i])

VPSIGNB dest, src1, src2 // AVX 128-bit or 256-bit Operands
VL=(128,256)
KL := VL/8
FOR i in 0...KL-1:
    dest.byte[i] := byte_sign(src2.byte[i], src1.byte[i])
DEST[MAXVL-1:VL] := 0

VPSIGNW dest, src1, src2 // AVX 128-bit or 256-bit Operands
VL=(128,256)
KL := VL/16
FOR i in 0...KL-1:
    dest.word[i] := word_sign(src2.word[i], src1.word[i])
DEST[MAXVL-1:VL] := 0

VPSIGND dest, src1, src2 // AVX 128-bit or 256-bit Operands
VL=(128,256)
KL := VL/32
FOR i in 0...KL-1:
    dest.dword[i] := dword_sign(src2.dword[i], src1.dword[i])
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB __m64 _mm_sign_pi8 (__m64 a, __m64 b)
(V)PSIGNB __m128i _mm_sign_epi8 (__m128i a, __m128i b)
VPSIGNB __m256i _mm256_sign_epi8 (__m256i a, __m256i b)
PSIGNW __m64 _mm_sign_pi16 (__m64 a, __m64 b)
(V)PSIGNW __m128i _mm_sign_epi16 (__m128i a, __m128i b)
VPSIGNW __m256i _mm256_sign_epi16 (__m256i a, __m256i b)
PSIGND __m64 _mm_sign_pi32 (__m64 a, __m64 b)
(V)PSIGND __m128i _mm_sign_epi32 (__m128i a, __m128i b)
VPSIGND __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-21, "Type 4 Class Exception Conditions," additionally:

#UD If VEX.L = 1.
PSLLDQ—Shift Double Quadword Left Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 73 /7 ib PSLLDQ xmm1, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift xmm1 left by imm8 bytes while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift xmm2 left by imm8 bytes while shifting in 0s and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift ymm2 left by imm8 bytes while shifting in 0s and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1, xmm2/ m128, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Operation

VPSLLDQ (EVEX.U1.512 Encoded Version)

```
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[MAXVL-1:512] := 0
```
VPSLLDQ (VEX.256 and EVEX.256 Encoded Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] << (TEMP * 8)
DEST[MAXVL-1:256] := 0

VPSLLDQ (VEX.128 and EVEX.128 Encoded Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC << (TEMP * 8)
DEST[MAXVL-1:128] := 0

PSLLDQ (128-bit Legacy SSE Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
(V)PSLLDQ __m128i _mm_slli_si128 ( __m128i a, int imm)
VPSLLDQ __m256i _mm256_slli_si256 ( __m256i a, const int imm)
VPSLLDQ __m512i _mm512_bslli_epi128 ( __m512i a, const int imm)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
### PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| NP 0F F1 /r<sup>1</sup>  
PSLLW mm, mm/m64 | A      | V/V                   | MMX               | Shift words in mm left mm/m64 while shifting in 0s. |
| 66 0F F1 /r  
PSLLW xmm1, xmm2/m128 | A      | V/V                   | SSE2              | Shift words in xmm1 left by xmm2/m128 while shifting in 0s. |
| NP 0F 71 /6 ib  
PSLLW mm1, imm8 | B      | V/V                   | MMX               | Shift words in mm left by imm8 while shifting in 0s. |
| 66 0F 71 /6 ib  
PSLLW xmm1, imm8 | B      | V/V                   | SSE2              | Shift words in xmm1 left by imm8 while shifting in 0s. |
| NP 0F F2 /r<sup>1</sup>  
PSLLD mm, mm/m64 | A      | V/V                   | MMX               | Shift doublewords in mm left by mm/m64 while shifting in 0s. |
| 66 0F F2 /r  
PSLLD xmm1, xmm2/m128 | A      | V/V                   | SSE2              | Shift doublewords in xmm1 left by xmm2/m128 while shifting in 0s. |
| NP 0F 72 /6 ib<sup>1</sup>  
PSLLD mm, imm8 | B      | V/V                   | MMX               | Shift doublewords in mm left by imm8 while shifting in 0s. |
| 66 0F 72 /6 ib  
PSLLD xmm1, imm8 | B      | V/V                   | SSE2              | Shift doublewords in xmm1 left by imm8 while shifting in 0s. |
| NP 0F F3 /r<sup>1</sup>  
PSLLQ mm, mm/m64 | A      | V/V                   | MMX               | Shift quadwords in mm left by mm/m64 while shifting in 0s. |
| 66 0F F3 /r  
PSLLQ xmm1, xmm2/m128 | A      | V/V                   | SSE2              | Shift quadwords in xmm1 left by xmm2/m128 while shifting in 0s. |
| NP 0F 73 /6 ib<sup>1</sup>  
PSLLQ mm, imm8 | B      | V/V                   | MMX               | Shift quadword in mm left by imm8 while shifting in 0s. |
| 66 0F 73 /6 ib  
PSLLQ xmm1, imm8 | B      | V/V                   | SSE2              | Shift quadword in xmm1 left by imm8 while shifting in 0s. |
| VEX.128.66.0F.WIG F1 /r  
VPSLLW xmm1, xmm2, xmm3/m128 | C      | V/V                   | AVX               | Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.128.66.0F.WIG F1 /r  
VPSLLW xmm1, xmm2, mm/m64 | C      | V/V                   | AVX               | Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.128.66.0F.WIG F2 /r  
VPSLLD xmm1, xmm2, xmm3/m128 | C      | V/V                   | AVX               | Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.128.66.0F.WIG F2 /r  
VPSLLD xmm1, xmm2, xmm/m64 | C      | V/V                   | AVX               | Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.128.66.0F.WIG F3 /r  
VPSLLQ xmm1, xmm2, xmm3/m128 | C      | V/V                   | AVX               | Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.128.66.0F.WIG F3 /r  
VPSLLQ xmm1, xmm2, xmm/m64 | C      | V/V                   | AVX               | Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.256.66.0F.WIG F1 /r  
VPSLLW ymm1, ymm2, xmm3/m128 | C      | V/V                   | AVX2              | Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.256.66.0F.WIG F1 /r  
VPSLLW ymm1, ymm2, xmm/m64 | C      | V/V                   | AVX2              | Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s. |
| VEX.256.66.0F.WIG F1 /r  
VPSLLW ymm1, ymm2, xmm/m64 | C      | V/V                   | AVX2              | Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s. |
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F.WIG F2 /r VPSLLD ymm1, ymm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 72 /6 ib VPSLLD ymm1, ymm2, xmm3/m128</td>
<td>D V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 left by imm8 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG F3 /r VPSLLQ ymm1, ymm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 73 /6 ib VPSLLQ ymm1, ymm2, xmm3/m128</td>
<td>D V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 left by imm8 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G V/V</td>
<td>AVX512BW</td>
<td>Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8</td>
<td>E V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, xmm2, xmm3/m128, imm8</td>
<td>E V/V</td>
<td>AVX512BW</td>
<td>Shift words in xmm2/m512 left by imm8 while shifting in 0 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>G V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, xmm2/m256/m32bcst, imm8</td>
<td>G V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, xmm2/m512/m32bcst, imm8</td>
<td>G V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>F V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, xmm2/m256/m32bcst, imm8</td>
<td>F V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, xmm2/m512/m32bcst, imm8</td>
<td>F V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

**NOTES:**

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

**Operation**

**PSLLW (with 64-bit Operand)**

```c
IF (COUNT > 15)
THEN
    DEST[64:0] := 0000000000000000H;
ELSE
    DEST[15:0] := ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
FI;
```

**PSLLD (with 64-bit operand)**

```c
IF (COUNT > 31)
THEN
    DEST[64:0] := 0000000000000000H;
ELSE
    DEST[31:0] := ZeroExtend(DEST[31:0] << COUNT);
FI;
```
PSLLQ (With 64-bit Operand)

IF (COUNT > 63)
    THEN
        DEST[64:0] := 00000000000000000000000000000000H;
    ELSE
        DEST := ZeroExtend(DEST << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN
        DEST[127:0] := 00000000000000000000000000000000H
    ELSE
        DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
        (* Repeat shift operation for 2nd through 7th words *)
        DEST[127:112] := ZeroExtend(SRC[127:112] << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN
        DEST[31:0] := 0
    ELSE
        DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN
        DEST[127:0] := 00000000000000000000000000000000H
    ELSE
        DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
        (* Repeat shift operation for 2nd through 3rd words *)
        DEST[127:96] := ZeroExtend(SRC[127:96] << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN
        DEST[63:0] := 0
    ELSE
        DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
    FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN
        DEST[127:0] := 00000000000000000000000000000000H
    ELSE

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical
DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
DEST[127:64] := ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
  DEST[127:0] := 00000000000000000000000000000000H
  DEST[255:128] := 00000000000000000000000000000000H
ELSE
  DEST[15:0] := ZeroExtend(SRC[15:0] << COUNT);
  (* Repeat shift operation for 2nd through 15th words *)
  DEST[255:240] := ZeroExtend(SRC[255:240] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
  DEST[127:0] := 00000000000000000000000000000000H
  DEST[255:128] := 00000000000000000000000000000000H
ELSE
  DEST[31:0] := ZeroExtend(SRC[31:0] << COUNT);
  (* Repeat shift operation for 2nd through 7th words *)
  DEST[255:224] := ZeroExtend(SRC[255:224] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
  DEST[127:0] := 00000000000000000000000000000000H
  DEST[255:128] := 00000000000000000000000000000000H
  DEST[255:192] := ZeroExtend(SRC[255:192] << COUNT);
ELSE
  DEST[63:0] := ZeroExtend(SRC[63:0] << COUNT);
  DEST[255:192] := ZeroExtend(SRC[255:192] << COUNT);
FI;

VPSLLw (EVEX Versions, xmm/m128)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
  TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
  TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
  TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
  TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j := 0 TO KL-1
\[
i := j \times 16 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \text{THEN } \text{DEST}[i+15:i] := \text{TMPDEST}[i+15:i] \\
\quad \text{ELSE} \\
\quad \quad \text{IF *merging-masking* ; merging-masking} \\
\quad \quad \quad \text{THEN *DEST}[i+15:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE *zeroing-masking* ; zeroing-masking} \\
\quad \quad \quad \quad \text{DEST}[i+15:i] = 0 \\
\quad \text{FI} \\
\FI \\
\text{ENDIF}
\]

\[
\text{DEST}[\text{MAXVL}-1:\text{VL}] := 0
\]

**VPSLLW (EVEX Versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
\[
\text{TMPDEST}[127:0] := \text{LOGICALLEFTSHIFTWORDS}_{128b}(\text{SRC1}[127:0], \text{imm8})
\]
FI;

IF VL = 256
\[
\text{TMPDEST}[255:0] := \text{LOGICALRIGHTSHIFTWORDS}_{256b}(\text{SRC1}[255:0], \text{imm8})
\]
FI;

IF VL = 512
\[
\text{TMPDEST}[255:0] := \text{LOGICALLEFTSHIFTWORDS}_{256b}(\text{SRC1}[255:0], \text{imm8}) \\
\text{TMPDEST}[511:256] := \text{LOGICALLEFTSHIFTWORDS}_{256b}(\text{SRC1}[511:256], \text{imm8})
\]
FI;

FOR \( j := 0 \) TO KL-1
\[
i := j \times 16 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \text{THEN } \text{DEST}[i+15:i] := \text{TMPDEST}[i+15:i] \\
\quad \text{ELSE} \\
\quad \quad \text{IF *merging-masking* ; merging-masking} \\
\quad \quad \quad \text{THEN *DEST}[i+15:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE *zeroing-masking* ; zeroing-masking} \\
\quad \quad \quad \quad \text{DEST}[i+15:i] = 0 \\
\quad \text{FI} \\
\FI \\
\text{ENDIF}
\]

\[
\text{DEST}[\text{MAXVL}-1:\text{VL}] := 0
\]

**VPSLLW (ymm, ymm, xmm/m128) - VEX.256 Encoding**

\[
\text{DEST}[255:0] := \text{LOGICALLEFTSHIFTWORDS}_{256b}(\text{SRC1}, \text{SRC2})
\]
\[
\text{DEST}[\text{MAXVL}-1:256] := 0;
\]

**VPSLLW (ymm, imm8) - VEX.256 Encoding**

\[
\text{DEST}[255:0] := \text{LOGICALLEFTSHIFTWORD}_{256b}(\text{SRC1}, \text{imm8})
\]
\[
\text{DEST}[\text{MAXVL}-1:256] := 0;
\]

**VPSLLW (xmm, xmm, xmm/m128) - VEX.128 Encoding**

\[
\text{DEST}[127:0] := \text{LOGICALLEFTSHIFTWORDS}(\text{SRC1}, \text{SRC2})
\]
\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
VPSLLw (xmm, imm8) - VEX.128 Encoding
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0

PSLLw (xmm, xmm, xmm/m128)
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSLLw (xmm, imm8)
DEST[127:0] := LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSLLD (EVEX versions, imm8)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] := LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPSLLD (EVEX Versions, xmm/m128)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
  TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
  TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
  TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
  TMP_DEST[511:256] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    DEST[i+31:i] := TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*
      DEST[i+31:i] := 0
    FI
  FI
VPSLLD (ymm, ymm, xmm/m128) - VEX.256 Encoding
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;

VPSLLD (ymm, imm8) - VEX.256 Encoding
DEST[255:0] := LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 Encoding
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPSLLD (xmm, imm8) - VEX.128 Encoding
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0

PSLLD (xmm, xmm, xmm/m128)
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSLLD (xmm, imm8)
DEST[127:0] := LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSLLQ (EVEX Versions, imm8)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] := LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
    FL;
    ELSE
      IF *merging-mask* ; merging-mask
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-mask* ; zeroing-mask
        DEST[i+63:i] := 0
      FL
  FI;
ENDFOR

VPSLLQ (EVEX Versions, xmm/m128)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
  TMP_DEST[127:0] := LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
  Fl;
IF VL = 256
  TMP_DEST[255:0] := LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
  FL;

PSLLD/PSLLD/PSLLQ—Shift Packed Data Left Logical
IF \( VL = 512 \)
\[
\begin{align*}
\text{TMP\_DEST}[255:0] & := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b}(\text{SRC1}[255:0], \text{SRC2}) \\
\text{TMP\_DEST}[511:256] & := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b}(\text{SRC1}[511:256], \text{SRC2}) \\
\end{align*}
\]
FI;

FOR \( j := 0 \) TO \( KL-1 \)
\[
\begin{align*}
i & := j \times 64 \\
\text{IF } k1[j] \text{ OR } \text{no writemask} & \text{ THEN } \text{DEST}[i+63:i] := \text{TMP\_DEST}[i+63:i] \\
\text{ELSE } & \\
\text{IF } \text{merging-masking} & \text{ THEN } \text{DEST}[i+63:i] \text{ remains unchanged} \\
\text{ELSE } \text{zeroing-masking} & \text{ THEN } \text{DEST}[i+63:i] := 0 \\
\end{align*}
\]
FI;
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)

\textbf{VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding}
\( \text{DEST}[255:0] := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b}(\text{SRC1}, \text{SRC2}) \\
\text{DEST}[\text{MAXVL}-1:256] := 0; \)

\textbf{VPSLLQ (ymm, imm8) - VEX.256 Encoding}
\( \text{DEST}[255:0] := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b}(\text{SRC1}, \text{imm8}) \\
\text{DEST}[\text{MAXVL}-1:256] := 0; \)

\textbf{VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding}
\( \text{DEST}[127:0] := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{SRC1}, \text{SRC2}) \\
\text{DEST}[\text{MAXVL}-1:128] := 0; \)

\textbf{VPSLLQ (xmm, imm8) - VEX.128 Encoding}
\( \text{DEST}[127:0] := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{SRC1}, \text{imm8}) \\
\text{DEST}[\text{MAXVL}-1:128] := 0; \)

\textbf{PSLLQ (xmm, xmm, xmm/m128)}
\( \text{DEST}[127:0] := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{DEST}, \text{SRC}) \\
\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \)

\textbf{PSLLQ (xmm, imm8)}
\( \text{DEST}[127:0] := \text{LOGICAL\_LEFT\_SHIFT\_QWORDS}(\text{DEST}, \text{imm8}) \\
\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)} \)

\textbf{Intel C/C++ Compiler Intrinsic Equivalents}

\texttt{VPSLLD __m512i \_mm512\_slli\_epi32(__m512i a, unsigned int imm);}  
\texttt{VPSLLD __m512i \_mm512\_mask\_slli\_epi32(__m512i a, unsigned int imm);}  
\texttt{VPSLLD \_m512i \_mm512\_maskz\_slli\_epi32(__mmask16 k, \_m512i a, unsigned int imm);}  
\texttt{VPSLLD \_m256i \_mm256\_mask\_slli\_epi32(__mmask8 k, \_m256i a, unsigned int imm);}  
\texttt{VPSLLD \_m256i \_mm256\_maskz\_slli\_epi32(__mmask8 k, \_m256i a, unsigned int imm);}  
\texttt{VPSLLD \_m128i \_mm128\_mask\_slli\_epi32(__m128i s, \_mmask8 k, \_m128i a, unsigned int imm);}  
\texttt{VPSLLD \_m128i \_mm128\_maskz\_slli\_epi32(__m128i s, \_mmask8 k, \_m128i a, unsigned int imm);}  
\texttt{VPSLLD \_m512i \_mm512\_slli\_epi32(__m512i a, \_m128i cnt);}  
\texttt{VPSLLD \_m512i \_mm512\_mask\_slli\_epi32(__m512i a, \_m128i cnt);}  
\texttt{VPSLLD \_m512i \_mm512\_maskz\_slli\_epi32(__m512i a, \_m128i cnt);}  
VPSLLD __m512i_mm512_maskz_slli_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i_mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i_mm128_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m512i_mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i_mm512_maskz_slli_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i_mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i_mm256_maskz_slli_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i_mm128_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i_mm128_maskz_slli_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m512i_mm512_slli_epi16(__m512i a, __m128i cnt);
VPSLLD __m512i_mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLD __m256i_mm256_mask_slli_epi16(__m256i s, __mmask32 k, __m256i a, __m128i cnt);
VPSLLD __m256i_mm256_maskz_slli_epi16(__mmask32 k, __m256i a, __m128i cnt);
VPSLLW __m512i_mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i_mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i_mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i_mm256_slli_epi16(__m256i s, __mmask32 k, __m256i a, unsigned int imm);
VPSLLW __m256i_mm256_mask_slli_epi16(__m256i s, __mmask32 k, __m256i a, __m128i cnt);
VPSLLW __m256i_mm256_maskz_slli_epi16(__mmask32 k, __m256i a, __m128i cnt);
VPSLLW __m128i_mm128_slli_epi16(__m128i s, __mmask32 k, __m128i a, __m128i cnt);
VPSLLW __m128i_mm128_mask_slli_epi16(__m128i s, __mmask32 k, __m128i a, __m128i cnt);
VPSLLW __m128i_mm128_maskz_slli_epi16(__mmask32 k, __m128i a, __m128i cnt);
VPSLLW __m512i_mm512_slli_epi64(__m512i a, __m128i cnt);
VPSLLW __m512i_mm512_mask_slli_epi64(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i_mm512_maskz_slli_epi64(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i_mm256_slli_epi64(__m256i s, __mmask32 k, __m256i a, __m128i cnt);
VPSLLW __m256i_mm256_mask_slli_epi64(__m256i s, __mmask32 k, __m256i a, __m128i cnt);
VPSLLW __m256i_mm256_maskz_slli_epi64(__mmask32 k, __m256i a, __m128i cnt);
PSLLW __m64_mm_slli_epi16(__m64 m, int count)
PSLLW __m64_mm_slli_epi16(__m64 m, __m64 m)
Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
• VEX-encoded instructions:
  — Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  — Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."
• EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
• EVEX-encoded VPSLLD/Q:
  — Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
  — Syntax with Full tuple type (F in the operand encoding table), see Table 2-49, "Type E4 Class Exception Conditions."
## PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E1 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td></td>
<td>Shift words in mm right by mm/m64 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E1 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td></td>
<td>Shift words in xmm1 right by xmm2/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 71 /4 lb¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td></td>
<td>Shift words in mm right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 71 /4 lb</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td></td>
<td>Shift words in xmm1 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAW xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F E2 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td></td>
<td>Shift doublewords in mm right by mm/m64 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E2 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td></td>
<td>Shift doubleword in xmm1 right by xmm2/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 72 /4 lb¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td></td>
<td>Shift doublewords in mm right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 72 /4 lb</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td></td>
<td>Shift doublewords in xmm1 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>PSRAD xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G E1 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td></td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G 71 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td></td>
<td>Shift words in xmm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G E2 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td></td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G 72 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td></td>
<td>Shift doublewords in xmm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAD xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G E1 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G 71 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift words in ymm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G E2 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift doublewords in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAW ymm1, ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G 72 /4 lb</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift doublewords in ymm2 right by imm8 while shifting in sign bits.</td>
</tr>
<tr>
<td>VPSRAD ymm1, ymm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1G E1 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAW xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1G E1 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAW ymm1 [k1][z], ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1G E1 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VPSRAW zmm1 [k1][z], zmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode/ Instruction</td>
<td>Op/ En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>-----------------------</td>
<td>--------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1G 71 /4 lb</td>
<td>E</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1G 71 /4 lb</td>
<td>E</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1G 71 /4 lb</td>
<td>E</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 E2 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 E2 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 E2 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 72 /4 lb</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 72 /4 lb</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 72 /4 lb</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 E2 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 E2 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 E2 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 72 /4 lb</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 72 /4 lb</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 72 /4 lb</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
**Description**

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

![Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand](image)

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.
EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

**Operation**

**PSRAW (With 64-bit Operand)**

IF (COUNT > 15)
   THEN COUNT := 16;
   Fiol
DEST[15:0] := SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] := SignExtend(DEST[63:48] >> COUNT);

PSRAD (with 64-bit operand)

IF (COUNT > 31)
   THEN COUNT := 32;
   Fiol
DEST[31:0] := SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] := SignExtend(DEST[63:32] >> COUNT);

**ARITHMETIC_RIGHT_SHIFT_DWORDS1**(SRC, COUNT_SRC)

COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
   THEN
      DEST[31:0] := SignBit
   ELSE
      DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
   Fiol

**ARITHMETIC_RIGHT_SHIFT_QWORDS1**(SRC, COUNT_SRC)

COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
   THEN
      DEST[63:0] := SignBit
   ELSE
      DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
   Fiol

**ARITHMETIC_RIGHT_SHIFT_WORDS_256b**(SRC, COUNT_SRC)

COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
   THEN COUNT := 16;
   Fiol
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] := SignExtend(SRC[255:240] >> COUNT);
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN COUNT := 32;
Ft;
DEST[31:0] := SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] := SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL) ; VL: 128b, 256b or 512b
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN COUNT := 64;
Ft;
DEST[63:0] := SignExtend(SRC[63:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] := SignExtend(SRC[VL-1:VL-64] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN COUNT := 16;
Ft;
DEST[15:0] := SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] := SignExtend(SRC[127:112] >> COUNT);

VPSRAW (EVEX versions, xmm/m128)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
Ft;
IF VL = 256
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
Ft;
IF VL = 512
    TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
Ft;
FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRAW (EVEX Versions, imm8)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
  TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
  TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
  TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
  TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TMP_DEST[i+15:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] = 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRAW (ymm, ymm, xmm/m128) - VEX
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

VPSRAW (ymm, imm8) - VEX
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0

VPSRAW (xmm, xmm, xmm/m128) - VEX
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPSRAW (xmm, imm8) - VEX
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0
PSRAW (xmm, xmm, xmm/m128)
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRAW (xmm, imm8)
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSRAD (EVEX Versions, imm8)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] := ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] := 0
        FI
    FI
  ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRAD (EVEX Versions, xmm/m128)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
  TMP_DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
  TMP_DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
  TMP_DEST[511:256] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
ENDFOR
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] := 0
        FI
    FI
  ENDFOR
DEST[MAXVL-1:VL] := 0
VPSRAD (ymm, ymm, xmm/m128) - VEX
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

VPSRAD (ymm, imm8) - VEX
DEST[255:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0

VPSRAD (xmm, xmm, xmm/m128) - VEX
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPSRAD (xmm, imm8) - VEX
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0

PSRAD (xmm, xmm, xmm/m128)
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRAD (xmm, imm8)
DEST[127:0] := ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

VPSRAQ (EVEX Versions, imm8)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] := ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI
  ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRAQ (EVEX Versions, xmm/m128)
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] := ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)
FOR j := 0 TO 7
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE
      THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+63:i] := 0
FI
FI;
ENDIFOR
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32(__mmask k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32(__mmask k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm128_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm128_maskz_srai_epi32(__mmask k, __m128i a, unsigned int imm);

VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32(__mmask k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32(__mmask k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm128_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm128_maskz_sra_epi32(__mmask k, __m128i a, __m128i cnt);

VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_maskz_srai_epi64(__mmask k, __m512i a, unsigned int imm);
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64(__mmask k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm128_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm128_maskz_srai_epi64(__mmask k, __m128i a, unsigned int imm);

VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m512i _mm512_maskz_srai_epi16(__mmask k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16(__mmask k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm128_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i _mm128_maskz_srai_epi16(__mmask k, __m128i a, unsigned int imm);

PSRAW __m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW __m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW __m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW __m128i _mm_sra_epi16(__m128i m, __m128i count)
VPSRAW __m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW __m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD __m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD __m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD __m128i _mm_sra_epi32 (__m128i m, int count)
(V)PSRAD __m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD __m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD __m256i _mm256_sra_epi32 (__m256i m, __m128i count)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, “Type 4 Class Exception Conditions.”
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, “Type 7 Class Exception Conditions.”
- EVEX-encoded VPSRAW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
- EVEX-encoded VPSRAD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-49, “Type E4 Class Exception Conditions.”
### PSRLDQ—Shift Double Quadword Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 73 /3 ib PSRLDQ xmm1, imm8</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Shift xmm1 right by imm8 while shifting in Os.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2, imm8</td>
<td>B V/V</td>
<td>AVX</td>
<td>Shift xmm2 right by imm8 bytes while shifting in Os.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2, imm8</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Shift ymm1 right by imm8 bytes while shifting in Os.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift xmm2/m128 right by imm8 bytes while shifting in Os and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift ymm2/m256 right by imm8 bytes while shifting in Os and store result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Shift zmm2/m512 right by imm8 bytes while shifting in Os and store result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>EVEX.vvvv (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.
Operation

VPSRLDQ (EVEX.512 Encoded Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[MAXVL-1:512] := 0;

VPSRLDQ (VEX.256 and EVEX.256 Encoded Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST[127:0] := SRC[127:0] >> (TEMP * 8)
DEST[MAXVL-1:256] := 0;

VPSRLDQ (VEX.128 and EVEX.128 Encoded Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := SRC >> (TEMP * 8)
DEST[MAXVL-1:128] := 0;

PSRLDQ (128-bit Legacy SSE Version)
TEMP := COUNT
IF (TEMP > 15) THEN TEMP := 16; FI
DEST := DEST >> (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents
(V)PSRLDQ __m128i __mm_srli_si128 ( __m128i a, int imm)
VPSRLDQ __m256i __mm256_bsrli_epi128 ( __m256i, const int)
VPSRLDQ __m512i __mm512_bsrli_epi128 ( __m512i, int)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-24, “Type 7 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D1 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in mm right by amount specified in mm/m64 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D1 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 71 /2 ib¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift words in mm right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLW mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 71 /2 ib</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift words in xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLW xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F D2 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in mm right by amount specified in mm/m64 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D2 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 72 /2 ib¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift doublewords in mm right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLD mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 72 /2 ib</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift doublewords in xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLD xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F D3 /r¹</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift mm right by amount specified in mm/m64 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLQ mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D3 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift quadwords in xmm1 right by amount specified in xmm2/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 73 /2 ib¹</td>
<td>B</td>
<td>V/V</td>
<td>MMX</td>
<td>Shift mm right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLQ mm, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 73 /2 ib</td>
<td>B</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shift quadwords in xmm1 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>PSRLQ xmm1, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.Wig D1 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.Wig 71 /2 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift words in xmm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLW xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.Wig D2 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.Wig 72 /2 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift doublewords in xmm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLD xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.Wig D3 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.Wig 73 /2 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX</td>
<td>Shift quadwords in xmm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLQ xmm1, xmm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.Wig D1 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in ymm2 right by amount specified in ymm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLW ymm1, ymm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.Wig 71 /2 ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift words in ymm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLW ymm1, ymm2, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode/ Instruction</td>
<td>Op/ En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------</td>
<td>------------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D2 /r VPSRLD ymm1, ymm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 72 /2 ib VPSRLD ymm1, ymm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D3 /r VPSRLQ ymm1, ymm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 73 /2 ib VPSRLQ ymm1, ymm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 right by imm8 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8</td>
<td>E</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8</td>
<td>E</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8</td>
<td>E</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/8/m32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128</td>
<td>G</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
</tbody>
</table>
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

![Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand](image-url)

NOTES:


### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>N/A</td>
<td>VEX.vvvv (w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Full Mem</td>
<td>EVEX.vvvv (w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>Full</td>
<td>EVEX.vvvv (w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>G</td>
<td>Mem128</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F.W1 D3 /r</td>
<td>G</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLQ zmm1 [k1][z], zmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 73 /2 ib</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLQ xmm1 [k1][z], xmm2/m128/m64bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 73 /2 ib</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLQ ymm1 [k1][z], ymm2/m256/m64bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 73 /2 ib</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLQ zmm1 [k1][z], zmm2/m512/m64bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

**Operation**

**PSRLW (with 64-bit Operand)**

IF (COUNT > 15)
THEN
DEST[64:0] := 0000000000000000H
ELSE
DEST[15:0] := ZeroExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
FI;

**PSRLD (with 64-bit Operand)**

IF (COUNT > 31)
THEN
DEST[64:0] := 0000000000000000H
ELSE
DEST[31:0] := ZeroExtend(DEST[31:0] >> COUNT);
FI;

**PSRLQ (with 64-bit Operand)**

IF (COUNT > 63)
THEN
DEST[64:0] := 0000000000000000H
ELSE
DEST := ZeroExtend(Dest >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] := 0
ELSE
    DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] := 0
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] := ZeroExtend(SRC[255:240] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] := 00000000000000000000000000000000H
ELSE
    DEST[15:0] := ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] := ZeroExtend(SRC[127:112] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] := 0
ELSE
    DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] := ZeroExtend(SRC[255:224] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
   DEST[127:0] := 00000000000000000000000000000000H
ELSE
   DEST[31:0] := ZeroExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 3rd words *)
   DEST[127:96] := ZeroExtend(SRC[127:96] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
   DEST[255:0] := 0
ELSE
   DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
   DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
   DEST[255:192] := ZeroExtend(SRC[255:192] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
   DEST[127:0] := 00000000000000000000000000000000H
ELSE
   DEST[63:0] := ZeroExtend(SRC[63:0] >> COUNT);
   DEST[127:64] := ZeroExtend(SRC[127:64] >> COUNT);
FI;

VPSRLW (EVEX Versions, xmm/m128)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
   TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
   TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
   TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
   TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] := TMP_DEST[i+15:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+15:i] = 0
   FI
FI;
VPSRLW (EVEX Versions, imm8)  
(KL, VL) = (8, 128), (16, 256), (32, 512)  
\[ \text{IF VL = 128} \] 
\[ \text{TMP_DEST}[127:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}_128b(\text{SRC1}[127:0], \text{imm8}) \]  
\[ \text{Fi;} \]  
\[ \text{IF VL = 256} \] 
\[ \text{TMP_DEST}[255:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}_256b(\text{SRC1}[255:0], \text{imm8}) \]  
\[ \text{Fi;} \]  
\[ \text{IF VL = 512} \] 
\[ \text{TMP_DEST}[255:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}_256b(\text{SRC1}[255:0], \text{imm8}) \]  
\[ \text{TMP_DEST}[511:256] := \text{LOGICAL_RIGHT_SHIFT_WORDS}_256b(\text{SRC1}[511:256], \text{imm8}) \]  
\[ \text{Fi;} \]  
\[ \text{FOR j := 0 TO KL-1} \]  
\[ i := j \times 16 \]  
\[ \text{IF k1[j] OR *no writemask*} \] 
\[ \text{THEN DEST}[i+15:i] := \text{TMP_DEST}[i+15:i] \]  
\[ \text{ELSE} \]  
\[ \text{IF *merging-masking*} \] 
\[ \text{THEN *DEST*[i+15:i] remains unchanged*} \]  
\[ \text{ELSE *zeroing-masking*} \] 
\[ \text{DEST}[i+15:i] = 0 \]  
\[ \text{Fi} \]  
\[ \text{ Fi;} \]  
\[ \text{ENDFOR} \]  
\[ \text{DEST[MAXVL-1:VL] := 0} \]  
VPSRLW (ymm, ymm, xmm/m128) - VEX.256 Encoding  
\[ \text{DEST}[255:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}_256b(\text{SRC1}, \text{SRC2}) \]  
\[ \text{DEST}[\text{MAXVL-1:256}] := 0; \]  
VPSRLW (ymm, imm8) - VEX.256 Encoding  
\[ \text{DEST}[255:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}_256b(\text{SRC1}, \text{imm8}) \]  
\[ \text{DEST}[\text{MAXVL-1:256}] := 0; \]  
VPSRLW (xmm, xmm, xmm/m128) - VEX.128 Encoding  
\[ \text{DEST}[127:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}(\text{SRC1}, \text{SRC2}) \]  
\[ \text{DEST}[\text{MAXVL-1:128}] := 0 \]  
VPSRLW (xmm, imm8) - VEX.128 Encoding  
\[ \text{DEST}[127:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}(\text{SRC1}, \text{imm8}) \]  
\[ \text{DEST}[\text{MAXVL-1:128}] := 0 \]  
PSRLW (xmm, xmm, xmm/m128)  
\[ \text{DEST}[127:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}(\text{DEST}, \text{SRC}) \]  
\[ \text{DEST}[\text{MAXVL-1:128}] \text{(Unmodified)} \]  
PSRLW (xmm, imm8)  
\[ \text{DEST}[127:0] := \text{LOGICAL_RIGHT_SHIFT_WORDS}(\text{DEST}, \text{imm8}) \]  
\[ \text{DEST}[\text{MAXVL-1:128}] \text{(Unmodified)} \]
VPSRLD (EVEX Versions, xmm/m128)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRLD (EVEX Versions, imm8)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
        ELSE DEST[i+31:i] := LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
    FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 Encoding
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;

VPSRLD (ymm, imm8) - VEX.256 Encoding
DEST[255:0] := LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;
### VPSRLD (xmm, xmm, xmm/m128) - VEX.128 Encoding

DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

### VPSRLD (xmm, imm8) - VEX.128 Encoding

DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0

### PSRLD (xmm, xmm, xmm/m128)

DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

### PSRLD (xmm, imm8)

DEST[127:0] := LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

### VPSRLQ (EVEX Versions, xmm/m128)

(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
   TMP_DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
   FI;
IF VL = 256
   TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
   FI;
IF VL = 512
   TMP_DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
   TMP_DEST[511:256] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
   FI;
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[i] OR *no writemask*
      THEN DEST[i+63:i] := TMP_DEST[i+63:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
               DEST[i+63:i] := 0
        FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPSRLQ (EVEX Versions, imm8)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] := LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 Encoding
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0;

VPSRLQ (ymm, imm8) - VEX.256 Encoding
DEST[255:0] := LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] := 0;

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 Encoding
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPSRLQ (xmm, imm8) - VEX.128 Encoding
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] := 0

PSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

PSRLQ (xmm, imm8)
DEST[127:0] := LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents
VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm128_maskr_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm128_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_maskr_srli_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m256i _mm256_maskz_srl_epi32(__m128i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_maskz_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_maskz_srl_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, unsigned int imm);

VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi16(__m512i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi16(__mmask8 k, __m256i a, __m128i cnt);
PSRLQ __m256i _mm256_srl_epi16(__m256i a, __m128i cnt);
PSRLQ __m256i _mm256_mask_srl_epi16(__m256i a, __m128i cnt);
PSRLQ __m256i _mm256_maskz_srl_epi16(__m256i a, __m128i cnt);

VPSRLD __m64 _mm_srl_pi32 (__m64 m, __m64 count)
VPSRLQ __m64 _mm_srl_si64 (__m64 m, __m64 count)

Flags Affected
None.
Numeric Exceptions

None.

Other Exceptions

- VEX-encoded instructions:
  - Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Table 2-21, "Type 4 Class Exception Conditions."
  - Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Table 2-24, "Type 7 Class Exception Conditions."

- EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."

- EVEX-encoded VPSRLD/Q:
  - Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
  - Syntax with Full tuple type (F in the operand encoding table), see Table 2-49, "Type E4 Class Exception Conditions."
## PSUBB/PSUBW/PSUBD—Subtract Packed Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F F8 /r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract packed byte integers in mm/m64 from packed byte integers in mm.</td>
</tr>
<tr>
<td>PSUBB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F F8 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed byte integers in xmm2/m128 from packed byte integers in xmm1.</td>
</tr>
<tr>
<td>PSUBB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F F9 /r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract packed word integers in mm/m64 from packed word integers in mm.</td>
</tr>
<tr>
<td>PSUBW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F F9 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed word integers in xmm2/m128 from packed word integers in xmm1.</td>
</tr>
<tr>
<td>PSUBW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F FA /r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract packed doubleword integers in mm/m64 from packed doubleword integers in mm.</td>
</tr>
<tr>
<td>PSUBD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F FA /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed doubleword integers in xmm2/m128 from packed doubleword integers in xmm1.</td>
</tr>
<tr>
<td>PSUBD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG F8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed byte integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG F9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed word integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG FA /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed doubleword integers in xmm3/m128 from xmm2.</td>
</tr>
<tr>
<td>VPSUBD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG F8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed byte integers in ymm3/m256 from ymm2.</td>
</tr>
<tr>
<td>VPSUBB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG F9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed word integers in ymm3/m256 from ymm2.</td>
</tr>
<tr>
<td>VPSUBW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG FA /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed doubleword integers in ymm3/m256 from ymm2.</td>
</tr>
<tr>
<td>VPSUBD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG F8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBB xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG F8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBB ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG F8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBB zmm1 [k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG F9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBW xmm1 [k1][z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG F9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBW ymm1 [k1][z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG F9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBW zmm1 [k1][z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Description
Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBBW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

**PSUBB (With 64-bit Operands)**

\[
\text{DEST}[7:0] := \text{DEST}[7:0] - \text{SRC}[7:0];
\]

(* Repeat subtract operation for 2nd through 7th byte *)

\[
\text{DEST}[63:56] := \text{DEST}[63:56] - \text{SRC}[63:56];
\]

**PSUBW (With 64-bit Operands)**

\[
\text{DEST}[15:0] := \text{DEST}[15:0] - \text{SRC}[15:0];
\]

(* Repeat subtract operation for 2nd and 3rd word *)

\[
\]

**PSUBD (With 64-bit Operands)**

\[
\text{DEST}[31:0] := \text{DEST}[31:0] - \text{SRC}[31:0];
\]

\[
\]

**PSUBD (With 128-bit Operands)**

\[
\text{DEST}[31:0] := \text{DEST}[31:0] - \text{SRC}[31:0];
\]

(* Repeat subtract operation for 2nd and 3rd doubleword *)

\[
\text{DEST}[127:96] := \text{DEST}[127:96] - \text{SRC}[127:96];
\]

**VPSUBB (EVEX Encoded Versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR \( j := 0 \) TO KL-1

\[
i := j \times 8
\]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+7:i] := \text{SRC1}[i+7:i] - \text{SRC2}[i+7:i] \)
ELSE

IF *merging-mask* ; merging-mask

THEN *DEST[i+7:i] remains unchanged* ; zeroing-mask

ELSE *zeroing-mask*

\[
\text{DEST}[i+7:i] = 0
\]

\fi

ENDFOR;

\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]
VPSUB (EVEX Encoded Versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC1[i+15:i] - SRC2[i+15:i]
  ELSE
    IF *merging-masking*
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking*
      DEST[i+15:i] := 0
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

VPSUBD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := SRC1[i+31:i] - SRC2[31:0]
    ELSE DEST[i+31:i] := SRC1[i+31:i] - SRC2[i+31:i]
    FI;
  ELSE
    IF *merging-masking*
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

VPSUBB (VEX.256 Encoded Version)
DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
VPSUBB (VEX.128 Encoded Version)

DEST[7:0] := SRC1[7:0]-SRC2[7:0]
DEST[47:40] := SRC1[47:40]-SRC2[47:40]
DEST[63:56] := SRC1[63:56]-SRC2[63:56]
DEST[71:64] := SRC1[71:64]-SRC2[71:64]
DEST[87:80] := SRC1[87:80]-SRC2[87:80]
DEST[103:96] := SRC1[103:96]-SRC2[103:96]
DEST[MAXVL-1:128] := 0

PSUBB (128-bit Legacy SSE Version)

DEST[7:0] := DEST[7:0]-SRC[7:0]
DEST[47:40] := DEST[47:40]-SRC[47:40]
DEST[71:64] := DEST[71:64]-SRC[71:64]
DEST[87:80] := DEST[87:80]-SRC[87:80]
DEST[103:96] := DEST[103:96]-SRC[103:96]
DEST[MAXVL-1:128] := 0
VPSUBW (VEX.256 Encoded Version)
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[159:144] := SRC1[159:144]-SRC2[159:144]
DEST[MAXVL-1:256] := 0

VPSUBW (VEX.128 Encoded Version)
DEST[15:0] := SRC1[15:0]-SRC2[15:0]
DEST[79:64] := SRC1[79:64]-SRC2[79:64]
DEST[111:96] := SRC1[111:96]-SRC2[111:96]
DEST[MAXVL-1:128] := 0

PSUBW (128-bit Legacy SSE Version)
DEST[15:0] := DEST[15:0]-SRC[15:0]
DEST[79:64] := DEST[79:64]-SRC[79:64]
DEST[111:96] := DEST[111:96]-SRC[111:96]
DEST[MAXVL-1:128] (Unmodified)

VPSUBD (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0]-SRC2[31:0]
DEST[95:64] := SRC1[95:64]-SRC2[95:64]
DEST[MAXVL-1:256] := 0
VPSUBD (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0]−SRC2[31:0]
DEST[95:64] := SRC1[95:64]−SRC2[95:64]
DEST[MAXVL-1:128] := 0

PSUBD (128-bit Legacy SSE Version)
DEST[31:0] := DEST[31:0]−SRC[31:0]
DEST[95:64] := DEST[95:64]−SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents
VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__mmask16 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi16(__mmask8 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi16(__mmask8 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi32(__m128i s, __mmask1 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi32(__mmask1 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi32(__mmask1 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi32(__mmask8 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi32(__mmask8 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi32(__m128i s, __mmask4 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi32(__mmask4 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi32(__mmask4 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi32(__m128i s, __mmask2 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi32(__mmask2 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi32(__mmask2 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi32(__m128i s, __mmask1 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi32(__mmask1 k, __m128i a, __m128i b);
VPSUBB __m256i _mm256_mask_sub_epi32(__mmask1 k, __m256i a, __m256i b);

Flags Affected
None.

Numeric Exceptions
None.
Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPSUBD, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
PSUBQ—Subtract Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F FB /r1 PSUBQ mm1, mm2/m64</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Subtract quadword integer in mm1 from mm2 /m64.</td>
<td></td>
</tr>
<tr>
<td>66 0F FB /r PSUBQ xmm1, xmm2/m128</td>
<td>A V/V</td>
<td>SSE2</td>
<td>Subtract packed quadword integers in xmm1 from xmm2 /m128.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG FB/r PSUBQ xmm1, xmm2, xmm3/m128</td>
<td>B V/V</td>
<td>AVX</td>
<td>Subtract packed quadword integers in xmm3/m128 from xmm2.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG FB/r PSUBQ ymm1, ymm2, ymm3/m256</td>
<td>B V/V</td>
<td>AVX2</td>
<td>Subtract packed quadword integers in ymm3/m256 from ymm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 FB /r PSUBQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 FB /r PSUBQ ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 FB /r PSUBQ zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location. 128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.
VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation**

**PSUBQ (with 64-Bit Operands)**
\[
\text{DEST}[63:0] := \text{DEST}[63:0] - \text{SRC}[63:0];
\]

**PSUBQ (with 128-Bit Operands)**
\[
\text{DEST}[63:0] := \text{DEST}[63:0] - \text{SRC}[63:0];
\]
\[
\text{DEST}[127:64] := \text{DEST}[127:64] - \text{SRC}[127:64];
\]

**VPSUBQ (VEX.128 Encoded Version)**
\[
\text{DEST}[63:0] := \text{SRC1}[63:0] - \text{SRC2}[63:0]
\]
\[
\text{DEST}[127:64] := \text{SRC1}[127:64] - \text{SRC2}[127:64]
\]
\[
\text{DEST}[\text{MAXVL-1:128}] := 0
\]

**VPSUBQ (VEX.256 Encoded Version)**
\[
\text{DEST}[63:0] := \text{SRC1}[63:0] - \text{SRC2}[63:0]
\]
\[
\text{DEST}[127:64] := \text{SRC1}[127:64] - \text{SRC2}[127:64]
\]
\[
\]
\[
\]
\[
\text{DEST}[\text{MAXVL-1:256}] := 0
\]

**VPSUBQ (EVEX Encoded Versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j = 0\) TO \(KL-1\)
\[i := j * 64\]
IF \(k1[j]\) OR *no writemask* THEN
\[
\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)}
\]
\[
\text{THEN DEST}[i+63:i] := SRC1[i+63:i] - SRC2[63:0]
\]
\[
\text{ELSE DEST}[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i]
\]
\[
\FI;
\]
ELSE
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{THEN *DEST}[i+63:i] remains unchanged*}
\]
\[
\text{ELSE *zeroing-masking* ; zeroing-masking}
\]
\[
\text{DEST}[i+63:i] := 0
\]
\[
\FI
\]
ENDFOR;
\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]
Intel C/C++ Compiler Intrinsic Equivalents

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ __m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPSUBQ, see Table 2-49, “Type E4 Class Exception Conditions.”
### PSUBSB/PSUBSW—Subtract Packed Signed Integers With Signed Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F E8 /r&lt;sup&gt;1&lt;/sup&gt;</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract signed packed bytes in mm/m64 from signed packed bytes in mm and saturate results.</td>
</tr>
<tr>
<td>PSUBSB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E8 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed signed byte integers in xmm2/m128 from packed signed byte integers in xmm1 and saturate results.</td>
</tr>
<tr>
<td>PSUBSB xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F E9 /r&lt;sup&gt;1&lt;/sup&gt;</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Subtract signed packed words in mm/m64 from signed packed words in mm and saturate results.</td>
</tr>
<tr>
<td>PSUBSW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F E9 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed signed word integers in xmm2/m128 from packed signed word integers in xmm1 and saturate results.</td>
</tr>
<tr>
<td>PSUBSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG E9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E8 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG E9 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results.</td>
</tr>
<tr>
<td>VPSUBSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG E8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG E8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG E8 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG E9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG E9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG E9 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NOTES:

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

### Operation

**PSUBSB (With 64-bit Operands)**

DEST[7:0] := SaturateToSignedByte (DEST[7:0] – SRC[7:0]);

("Repeat subtract operation for 2nd through 7th bytes *)

PSUBSW (With 64-bit Operands)

\[ \text{DEST}[15:0] := \text{SaturateToSignedWord}(\text{DEST}[15:0] - \text{SRC}[15:0]); \]
\((^* \text{Repeat subtract operation for 2nd and 7th words} ^*)\)


VPSUBSB (EVEX Encoded Versions)

\((KL, VL) = (16, 128), (32, 256), (64, 512)\)

FOR \( j := 0 \) TO \( KL - 1 \)

\[ i := j \times 8; \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+7:i] := \text{SaturateToSignedByte}(\text{SRC1}[i+7:i] - \text{SRC2}[i+7:i]) \)
ELSE

IF *merging-masking* ; merging-masking

THEN *\text{DEST}[i+7:i] remains unchanged*\n
ELSE *zeroing-masking* ; zeroing-masking

\( \text{DEST}[i+7:i] := 0; \)
FI
\FI
ENDFOR;

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)

VPSUBSB (EVEX Encoded Versions)

\((KL, VL) = (8, 128), (16, 256), (32, 512)\)

FOR \( j := 0 \) TO \( KL - 1 \)

\[ i := j \times 16 \]

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+15:i] := \text{SaturateToSignedWord}(\text{SRC1}[i+15:i] - \text{SRC2}[i+15:i]) \)
ELSE

IF *merging-masking* ; merging-masking

THEN *\text{DEST}[i+15:i] remains unchanged*\n
ELSE *zeroing-masking* ; zeroing-masking

\( \text{DEST}[i+15:i] := 0; \)
FI
\FI
ENDFOR;

\( \text{DEST}[\text{MAXVL}-1:VL] := 0; \)

VPSUBSB (VEX.256 Encoded Version)

\( \text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{SRC1}[7:0] - \text{SRC2}[7:0]); \)
\((^* \text{Repeat subtract operation for 2nd through 31th bytes} ^*)\)

\( \text{DEST}[\text{MAXVL}-1:256] := 0; \)

VPSUBSB (VEX.128 Encoded Version)

\( \text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{SRC1}[7:0] - \text{SRC2}[7:0]); \)
\((^* \text{Repeat subtract operation for 2nd through 14th bytes} ^*)\)

\( \text{DEST}[127:120] := \text{SaturateToSignedByte}(\text{SRC1}[127:120] - \text{SRC2}[127:120]); \)
\( \text{DEST}[\text{MAXVL}-1:128] := 0; \)

PSUBSB (128-bit Legacy SSE Version)

\( \text{DEST}[7:0] := \text{SaturateToSignedByte}(\text{DEST}[7:0] - \text{SRC}[7:0]); \)
\((^* \text{Repeat subtract operation for 2nd through 14th bytes} ^*)\)

\( \text{DEST}[127:120] := \text{SaturateToSignedByte}(\text{DEST}[127:120] - \text{SRC}[127:120]); \)
\( \text{DEST}[\text{MAXVL}-1:128] \) (Unmodified);
VPSUBSW (VEX.256 Encoded Version)
(* Repeat subtract operation for 2nd through 15th words *)
DEST[MAXVL-1:256] := 0;

VPSUBSW (VEX.128 Encoded Version)
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] := 0;

PSUBSW (128-bit Legacy SSE Version)
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] (Unmodified);

Intel C/C++ Compiler Intrinsic Equivalents
VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_subs_epi8(__m256i a, __m256i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm128_subs_epi8(__m128i a, __m128i b);
VPSUBSB __m128i _mm128_mask_subs_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm128_maskz_subs_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm128_maskz_subs_epi16(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_subs_epi16(__m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm128_subs_epi16(__m128i a, __m128i b);
VPSUBSW __m128i _mm128_maskz_subs_epi16(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm128_maskz_subs_epi16(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_subs_epi16(__m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm128_subs_epi16(__m128i a, __m128i b);
VPSUBSW __m128i _mm128_maskz_subs_epi16(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm128_maskz_subs_epi16(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F D8 /r¹</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Subtract unsigned packed bytes in mm/m64 from unsigned packed bytes in mm and saturate result.</td>
</tr>
<tr>
<td>PSUBUSB mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D8 /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Subtract packed unsigned byte integers in xmm2/m128 from packed unsigned byte integers in xmm1 and saturate result.</td>
</tr>
<tr>
<td>PSUBUSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F D9 /r¹</td>
<td>A V/V</td>
<td>MMX</td>
<td></td>
<td>Subtract unsigned packed words in mm/m64 from unsigned packed words in mm and saturate result.</td>
</tr>
<tr>
<td>PSUBUSW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F D9 /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Subtract packed unsigned word integers in xmm2/m128 from packed unsigned word integers in xmm1 and saturate result.</td>
</tr>
<tr>
<td>PSUBUSW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D8 /r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2 and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSB xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG D9 /r</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D8 /r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2 and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSB ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG D9 /r</td>
<td>B V/V</td>
<td>AVX2</td>
<td></td>
<td>Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2 and saturate result.</td>
</tr>
<tr>
<td>VPSUBUSW ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG D8 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td></td>
<td>Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG D8 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td></td>
<td>Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG D8 /r</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td></td>
<td>Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.WIG D9 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td></td>
<td>Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.WIG D9 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td></td>
<td>Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.WIG D9 /r</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td></td>
<td>Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Description**

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation**

**PSUBUSB (With 64-bit Operands)**

\[ \text{DEST}[7:0] := \text{SaturateToUnsignedByte}(\text{DEST}[7:0] – \text{SRC}[7:0]); \]
\[ (* \text{Repeat add operation for 2nd through 7th bytes} *) \]
\[ \text{DEST}[63:56] := \text{SaturateToUnsignedByte}(\text{DEST}[63:56] – \text{SRC}[63:56]); \]

**PSUBUSW (With 64-bit Operands)**

\[ \text{DEST}[15:0] := \text{SaturateToUnsignedWord}(\text{DEST}[15:0] – \text{SRC}[15:0]); \]
\[ (* \text{Repeat add operation for 2nd and 3rd words} *) \]

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**NOTES:**

VPSUBUSB (EVEX Encoded Versions)

\[(KL, VL) = (16, 128), (32, 256), (64, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times 8;\]

IF \(k1[j] OR *no writemask*\)

THEN \(DEST[i+7:i] := \text{SaturateToUnsignedByte}(SRC1[i+7:i] - SRC2[i+7:i])\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+7:i] remains unchanged* ; zeroing-masking

ELSE *zeroing-masking*

\[DEST[i+7:i] := 0;\]

\FI

\FI

ENDFOR;

\(DEST[MAXVL-1:VL] := 0;\)

VPSUBUSW (EVEX Encoded Versions)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times 16;\]

IF \(k1[j] OR *no writemask*\)

THEN \(DEST[i+15:i] := \text{SaturateToUnsignedWord}(SRC1[i+15:i] - SRC2[i+15:i])\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+15:i] remains unchanged* ; zeroing-masking

ELSE *zeroing-masking*

\[DEST[i+15:i] := 0;\]

\FI

\FI

ENDFOR;

\(DEST[MAXVL-1:VL] := 0;\)

VPSUBUSB (VEX.256 Encoded Version)

\(DEST[7:0] := \text{SaturateToUnsignedByte}(SRC1[7:0] - SRC2[7:0]);\)

(* Repeat subtract operation for 2nd through 31st bytes *)


\(DEST[MAXVL-1:256] := 0;\)

VPSUBUSB (VEX.128 Encoded Version)

\(DEST[7:0] := \text{SaturateToUnsignedByte}(SRC1[7:0] - SRC2[7:0]);\)

(* Repeat subtract operation for 2nd through 14th bytes *)

\(DEST[127:120] := \text{SaturateToUnsignedByte}(SRC1[127:120] - SRC2[127:120]);\)

\(DEST[MAXVL-1:128] := 0;\)

PSUBUSB (128-bit Legacy SSE Version)

\(DEST[7:0] := \text{SaturateToUnsignedByte}(DEST[7:0] - SRC[7:0]);\)

(* Repeat subtract operation for 2nd through 14th bytes *)

\(DEST[127:120] := \text{SaturateToUnsignedByte}(DEST[127:120] - SRC[127:120]);\)

\(DEST[MAXVL-1:128] \text{ (Unmodified)};\)
PSUBSUB (VEX.256 Encoded Version)
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[MAXVL-1:256] := 0;

PSUBSUB (VEX.128 Encoded Version)
DEST[15:0] := SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] := 0;

PSUBSW (128-bit Legacy SSE Version)
DEST[15:0] := SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents
VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu16(__mmask32 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_subs_epu16(__m256i a, __m256i b);
VPSUBUSB __m256i _mm256_mask_subs_epu16(__mmask16 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu16(__mmask16 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu16(__mmask8 k, __m128i a, __m128i b);
PSUBUSB _m64 _mm_subs_pu8(__m64 m1, __m64 m2);
(V)PSUBUSB _m128 _mm_subs_epu8(__m128 m1, __m128 m2)
VPSUBUSB __m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSB _m64 _mm_subs_pu16(__m64 m1, __m64 m2);
(V)PSUBUSB _m128 _mm_subs_epu16(__m128 m1, __m128 m2)
VPSUBUSB __m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
PTEST—Logical Compare

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 17 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Set ZF if xmm2/m128 AND xmm1 result is all 0s. Set CF if xmm2/m128 AND NOT xmm1 result is all 0s.</td>
</tr>
<tr>
<td>PTEST xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1G 17 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on bitwise AND and ANDN of sources.</td>
</tr>
<tr>
<td>VPTEST xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1G 17 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on bitwise AND and ANDN of sources.</td>
</tr>
<tr>
<td>VPTEST ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M reg field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)PTEST (128-bit Version)

IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)
    THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;

VPTEST (VEX.256 Encoded Version)

IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF := 1;
    ELSE ZF := 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF := 1;
    ELSE CF := 0;
DEST (unmodified)
AF := OF := PF := SF := 0;
Intel C/C++ Compiler Intrinsic Equivalent

PTEST int _mm_testz_si128 (__m128i s1, __m128i s2);
PTEST int _mm_testc_si128 (__m128i s1, __m128i s2);
PTEST int _mm_testnzc_si128 (__m128i s1, __m128i s2);
VPTEST int _mm256_testz_si256 (__m256i s1, __m256i s2);
VPTEST int _mm256_testc_si256 (__m256i s1, __m256i s2);
VPTEST int _mm256_testnzc_si256 (__m256i s1, __m256i s2);
VPTEST int _mm_testz_si128 (__m128i s1, __m128i s2);
VPTEST int _mm_testc_si128 (__m128i s1, __m128i s2);
VPTEST int _mm_testnzc_si128 (__m128i s1, __m128i s2);

Flags Affected
The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions,” additionally:
#UD If VEX.vvvv ≠ 1111B.
PTWRITE—Write Data to a Processor Trace Packet

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 REX.W OF AE /4</td>
<td>RM</td>
<td>V/N/E</td>
<td>PTWRITE</td>
<td>Reads the data from r64/m64 to encode into a PTWrite packet if dependencies are met (see details below).</td>
</tr>
<tr>
<td>PTWRITE r64/m64</td>
<td>RM</td>
<td>V/N/E</td>
<td>PTWRITE</td>
<td>Reads the data from r32/m32 to encode into a PTWrite packet if dependencies are met (see details below).</td>
</tr>
<tr>
<td>F3 0F AE /4</td>
<td>RM</td>
<td>V/V</td>
<td>PTWRITE</td>
<td></td>
</tr>
<tr>
<td>PTWRITE r32/m32</td>
<td>RM</td>
<td>V/V</td>
<td>PTWRITE</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:rm (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTWrite packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, Section 33.2.2, “Software Trace Instrumentation with PTWRITE.” The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

*Note:* The instruction will #UD if prefix 66H is used.

**Operation**

IF (IA32_RTIT_STATUS.TriggerEn & IA32_RTIT_STATUS.ContextEn & IA32_RTIT_STATUS.FilterEn & IA32_RTIT_CTL.PTWEn) = 1

PTWritePayloadBytes := Encoded payload size;
PTWriteIP := IA32_RTIT_CTL.FUPonPTWrite
IF IA32_RTIT_CTL.FUPonPTWrite = 1
Insert FUP packet with IP of PTWRITE;
FI;
FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF (fault-code) For a page fault.
- #AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
- #UD If CPUID.(EAX=14H, ECX=0H):EBX.PTWRITE [Bit 4] = 0.
  If LOCK prefix is used.
  If 66H prefix is used.
Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If CPUID.(EAX=14H, ECX=0H):EBX._PTWRITE [Bit 4] = 0.
   If LOCK prefix is used.
   If 66H prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF (fault-code) For a page fault.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If CPUID.(EAX=14H, ECX=0H):EBX._PTWRITE [Bit 4] = 0.
   If LOCK prefix is used.
   If 66H prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If CPUID.(EAX=14H, ECX=0H):EBX._PTWRITE [Bit 4] = 0.
   If LOCK prefix is used.
   If 66H prefix is used.
## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 68 /r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Unpack and interleave high-order bytes from mm and mm/m64 into mm.</td>
</tr>
<tr>
<td>PUNPCKHBW mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 68 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpack and interleave high-order bytes from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>PUNPCKHBW xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 69 /r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Unpack and interleave high-order words from mm and mm/m64 into mm.</td>
</tr>
<tr>
<td>PUNPCKHWD mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 69 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpack and interleave high-order words from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>PUNPCKHWD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP 0F 6A /r1</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Unpack and interleave high-order doublewords from mm and mm/m64 into mm.</td>
</tr>
<tr>
<td>PUNPCKHDQ mm, mm/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 6A /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpack and interleave high-order doublewords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>PUNPCKHDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 6D /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpack and interleave high-order quadwords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>PUNPCKHDQ xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W16 68 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>VPUNPCKHBW xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W16 69 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave high-order words from xmm2 and xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>VPUNPCKHWD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W16 6A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave high-order doublewords from xmm2 and xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>VPUNPCKHDQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F.W16 6D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave high-order quadwords from xmm2 and xmm3/m128 into xmm1 register.</td>
</tr>
<tr>
<td>VPUNPCKHDQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W16 68 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VPUNPCKHBW ymin1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W16 69 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VPUNPCKHWD ymin1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W16 6A /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave high-order doublewords from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VPUNPCKHDQ ymin1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F.W16 6D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave high-order quadword from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VPUNPCKHDQ ymin1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W8 68 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL  AVX512BW</td>
<td>Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHBW xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W8 69 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL  AVX512BW</td>
<td>Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHWD xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W8 6A /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL  AVX512F</td>
<td>Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHDQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F.W8 6D /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL  AVX512F</td>
<td>Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.</td>
</tr>
<tr>
<td>VPUNPCKHDQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| EVEX.256.66.0F.W1 6B /r
VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256 | C V/V | AVX512VL AVX512BW | Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask. |
| EVEX.256.66.0F.W1 6D /r
VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst | D V/V | AVX512VL AVX512F | Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask. |
| EVEX.512.66.0F.W1 6B /r
VPUNPCKBHWD zmm1 {k1}{z}, zmm2, zmm3/m512 | C V/V | AVX512Bl | Interleave high-order doublewords from zmm2 and zmm3/m512 into zmm1 register. |
| EVEX.512.66.0F.W1 6D /r
VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst | D V/V | AVX512F | Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask. |

### Instruction Opcode Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.
When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.
EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation**

**PUNPCKHBW Instruction With 64-bit Operands:**

\[
\begin{align*}
\text{DEST}[7:0] &= \text{DEST}[39:32]; \\
\text{DEST}[15:8] &= \text{SRC}[39:32]; \\
\text{DEST}[23:16] &= \text{DEST}[47:40]; \\
\text{DEST}[31:24] &= \text{SRC}[47:40]; \\
\text{DEST}[39:32] &= \text{DEST}[55:48]; \\
\text{DEST}[47:40] &= \text{SRC}[55:48]; \\
\text{DEST}[55:48] &= \text{DEST}[63:56]; \\
\text{DEST}[63:56] &= \text{SRC}[63:56];
\end{align*}
\]

**PUNPCKHW Instruction With 64-bit Operands:**

\[
\begin{align*}
\text{DEST}[15:0] &= \text{DEST}[47:32]; \\
\text{DEST}[31:16] &= \text{SRC}[47:32]; \\
\text{DEST}[47:32] &= \text{DEST}[63:48]; \\
\text{DEST}[63:48] &= \text{SRC}[63:48];
\end{align*}
\]

**PUNPCKHDQ Instruction With 64-bit Operands:**

\[
\begin{align*}
\text{DEST}[31:0] &= \text{DEST}[63:32]; \\
\text{DEST}[63:32] &= \text{SRC}[63:32];
\end{align*}
\]

**INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)**

\[
\begin{align*}
\text{TMP_DEST}[255:0] &= \text{INTERLEAVE_HIGH_BYTES_256b}(\text{SRC1}[255:0], \text{SRC}[255:0]) \\
\text{TMP_DEST}[511:256] &= \text{INTERLEAVE_HIGH_BYTES_256b}(\text{SRC1}[511:256], \text{SRC}[511:256])
\end{align*}
\]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] := SRC[71:64]
DEST[15:8] := SRC[71:64]
DEST[47:40] := SRC[87:80]
DEST[63:56] := SRC[95:88]
DEST[71:64] := SRC[103:96]
DEST[79:72] := SRC[103:96]
DEST[87:80] := SRC[111:104]
DEST[103:96] := SRC[119:112]
DEST[127:120] := SRC[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b (SRC1, SRC2)
DEST[15:0] := SRC[79:64]
DEST[31:16] := SRC[79:64]
DEST[79:64] := SRC[111:96]
DEST[87:80] := SRC[111:96]
DEST[95:80] := SRC[111:96]
DEST[111:96] := SRC[127:112]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]

INTERLEAVE_HIGH_DWORDS_256(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]

PUNPCKHBW (128-bit Legacy SSE Version)
DEST[127:0] := INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] := 0

VPUNPCKHBW (VEX.128 Encoded Version)
DEST[127:0] := INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0

VPUNPCKHBW (VEX.256 Encoded Version)
DEST[255:0] := INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
VPUNPCKHBW (EVEX Encoded Versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128
   TMP_DEST[VL-1:0] := INTERLEAVE_HIGHBYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

IF VL = 256
   TMP_DEST[VL-1:0] := INTERLEAVE_HIGHBYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

IF VL = 512
   TMP_DEST[VL-1:0] := INTERLEAVE_HIGHBYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
   i := j * 8
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] := TMP_DEST[i+7:i]
   ELSE
      IF *merging-masking*; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking*; zeroing-masking
         DEST[i+7:i] := 0
   FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

PUNPCKHWD (128-bit Legacy SSE Version)
DEST[127:0] := INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHWD (VEX.128 Encoded Version)
DEST[127:0] := INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0

VPUNPCKHWD (VEX.256 Encoded Version)
DEST[255:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

VPUNPCKHWD (EVEX Encoded Versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
   TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

IF VL = 256
   TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

IF VL = 512
   TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
THEN DEST[i+15:i] := TMP_DEST[i+15:i]
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] := 0
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

PUNPCKHDQ (128-bit Legacy SSE Version)
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHDQ (VEX.128 Encoded Version)
DEST[127:0] := INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0

VPUNPCKHDQ (VEX.256 Encoded Version)
DEST[255:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

VPUNPCKHDQ (EVEX.512 Encoded Version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fi;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fi;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fi;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**PUNPCKHQDQ (128-bit Legacy SSE Version)**
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] := 0

**VPUNPCKHQDQ (VEX.128 Encoded Version)**
DEST[127:0] := INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

**VPUNPCKHQDQ (VEX.256 Encoded Version)**
DEST[255:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

**VPUNPCKHQDQ (EVEX Encoded Versions)**
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN TMP_SRC2[i+63:i] := SRC2[63:0]
ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
FI;
ENDFOR;
IF VL = 128
TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fi;
IF VL = 256
TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fi;
IF VL = 512
TMP_DEST[VL-1:0] := INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
Fi;
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:i] := TMP_DEST[i+63:i]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+63:i] := 0
FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalents**
VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m128i __mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKBHWD __m512i __mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHDQ __m512i __mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i __mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHDQ __m128i __mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i __mm512_maskz_unpackhi_epi8( __mmask16 k, __m128i a, __m128i b);
VPUNPCKHDQ __m256i __mm512_maskz_unpackhi_epi16(__m512i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHDQ __m512i __mm512_maskz_unpackhi_epi32( __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i __mm512_maskz_unpackhi_epi64( __mmask16 k, __m256i a, __m256i b);
VPUNPCKHDQ __m128i __mm_maskz_unpackhi_epi16(__m512i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i __mm512_maskz_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i __mm512_maskz_unpackhi_epi64(__m512i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKHDQ __m128i __mm_maskz_unpackhi_epi32(__m512i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i __mm512_maskz_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i __mm512_maskz_unpackhi_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKHDQ __m128i __mm_maskz_unpackhi_epi64(__m256i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i __mm512_maskz_unpackhi_epi64(__m128i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i __mm512_maskz_unpackhi_epi64(__m128i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKHDQ __m128i __mm_maskz_unpackhi_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);

PUNPCKHBW __m64 __mm64_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW __m128i __mm128i_unpackhi_pi8(__m128i m1, __m128i m2)
PUNPCKHBW __m256i __mm256i_unpackhi_pi8(__m256i m1, __m256i m2)
(V)PUNPCKHBW __m128i __mm128i_unpackhi_pi16(__m64 m1, __m64 m2)
(V)PUNPCKHBW __m128i __mm128i_unpackhi_pi16(__m128i m1, __m128i m2)
PUNPCKHDQ __m64 __mm64_unpackhi_pi16(__m64 m1, __m64 m2)
(V)PUNPCKHDQ __m128i __mm128i_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ __m128i __mm128i_unpackhi_pi32(__m128i m1, __m128i m2)
PUNPCKHDQ __m256i __mm256i_unpackhi_pi32(__m256i m1, __m256i m2)
(V)PUNPCKHDQ __m128i __mm128i_unpackhi_pi64 (__m64 m1, __m64 m2)
PUNPCKHDQ __m256i __mm256i_unpackhi_pi64 (__m256i a, __m256i b)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, ”Type 4 Class Exception Conditions.”
EVEX-encoded VPUNPCKHDQ/QDQ, see Table 2-50, ”Type E4NF Class Exception Conditions.”
EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb in Table 2-50, ”Type E4NF Class Exception Conditions.”
## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 60 /r1 PUNPCKLBW mm, mm/m32</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Interleave low-order bytes from mm and mm/m32 into mm.</td>
</tr>
<tr>
<td>66 0F 60 /r PUNPCKLBW xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Interleave low-order bytes from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>NP 0F 61 /r1 PUNPCKLWD mm, mm/m32</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Interleave low-order words from mm and mm/m32 into mm.</td>
</tr>
<tr>
<td>66 0F 61 /r PUNPCKLWD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Interleave low-order words from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>NP 0F 62 /r1 PUNPCKLDQ mm, mm/m32</td>
<td>A</td>
<td>V/V</td>
<td>MMX</td>
<td>Interleave low-order doublewords from mm and mm/m32 into mm.</td>
</tr>
<tr>
<td>66 0F 62 /r PUNPCKLDQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Interleave low-order doublewords from xmm1 and xmm2/m128 into xmm1.</td>
</tr>
<tr>
<td>66 0F 6C /r PUNPCKLQDQ xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Interleave low-order quadword from xmm1 and xmm2/m128 into xmm1 register.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WiG 60 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WiG 61 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave low-order words from xmm2 and xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WiG 62 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave low-order doublewords from xmm2 and xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WiG 6C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Interleave low-order quadword from xmm2 and xmm3/m128 into xmm1 register.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WiG 60 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WiG 61 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WiG 62 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave low-order doublewords from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WiG 6C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX2</td>
<td>Interleave low-order quadword from ymm2 and ymm3/m256 into ymm1 register.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WiG 60 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.WiG 61 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.Wo 60 /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.Wi 6C /r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.</td>
</tr>
</tbody>
</table>
**Description**

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.
operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation**

**PUNPCKLBw Instruction With 64-bit Operands:**
- DEST[63:56] := SRC[31:24];
- DEST[47:40] := SRC[23:16];
- DEST[31:24] := SRC[15:8];
- DEST[23:16] := DEST[15:8];
- DEST[15:8] := SRC[7:0];
- DEST[7:0] := DEST[7:0];

**PUNPCKLWD Instruction With 64-bit Operands:**
- DEST[63:48] := SRC[31:16];
- DEST[31:16] := SRC[15:0];
- DEST[15:0] := DEST[15:0];

**PUNPCKLDQ Instruction With 64-bit Operands:**
- DEST[63:32] := SRC[31:0];
- DEST[31:0] := DEST[31:0];

`INTERLEAVE_BYTES_512b (SRC1, SRC2)`

TMP_DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])

TMP_DEST[511:256] := INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

`INTERLEAVE_BYTES_256b (SRC1, SRC2)`

DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[47:40] := SRC2[23:16]
DEST[63:56] := SRC2[31:24]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]
DEST[151:144] := SRC1[143:136]
DEST[159:152] := SRC2[143:136]
DEST[183:176] := SRC1[159:152]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] := SRC1[7:0]
DEST[15:8] := SRC2[7:0]
DEST[47:40] := SRC2[23:16]
DEST[63:56] := SRC2[31:24]
DEST[87:80] := SRC1[47:40]
DEST[95:88] := SRC2[47:40]
DEST[119:112] := SRC1[63:56]
DEST[127:120] := SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] := INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
DEST[79:64] := SRC1[47:32]
DEST[111:96] := SRC1[63:48]
DEST[159:144] := SRC2[143:128]
DEST[175:160] := SRC1[159:144]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] := SRC1[15:0]
DEST[31:16] := SRC2[15:0]
DEST[79:64] := SRC1[47:32]
DEST[111:96] := SRC1[63:48]

`INTERLEAVE_DWORDS_512b (SRC1, SRC2)`
TMP_DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

`INTERLEAVE_DWORDS_256b(SRC1, SRC2)`
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]

`INTERLEAVE_DWORDS(SRC1, SRC2)`
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]

`INTERLEAVE_QWORDS_512b (SRC1, SRC2)`
TMP_DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] := INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

`INTERLEAVE_QWORDS_256b(SRC1, SRC2)`
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]

`INTERLEAVE_QWORDS(SRC1, SRC2)`
DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]

`PUNPCKLBW`  
DEST[127:0] := INTERLEAVE_BYTES(DEST, SRC)  
DEST[255:127] := (Unmodified)

`VPUNPCKLBW (VEX.128 Encoded Instruction)`
DEST[127:0] := INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] := 0

`VPUNPCKLBW (VEX.256 Encoded Instruction)`
DEST[255:0] := INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0
VPUNPCKLBW (EVEX.512 Encoded Instruction)
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
   TMP_DEST[VL-1:0] := INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
   TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
   TMP_DEST[VL-1:0] := INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
FOR j := 0 TO KL-1
   i := j * 8
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] := TMP_DEST[i+7:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+7:i] := 0
   FI
ENDFOR
DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_BYTES_512b(SRC1, SRC2)

PUNPCKLWD
DEST[127:0] := INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLWD (VEX.128 Encoded Instruction)
DEST[127:0] := INTERLEAVE_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] := 0

VPUNPCKLWD (VEX.256 Encoded Instruction)
DEST[255:0] := INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

VPUNPCKLWD (EVEX.512 Encoded Instruction)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
   TMP_DEST[VL-1:0] := INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
   TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
   TMP_DEST[VL-1:0] := INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
THEN Dest[i+15:i] := TMP_DEST[i+15:i]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *Dest[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] := 0
  FI
ENDFOR

DEST[MAXVL-1:VL] := 0
DEST[511:0] := INTERLEAVE_WORDS_512b(SRC1, SRC2)

PUNPCKLDQ
DEST[127:0] := INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] := 0

VPUNPCKLDQ (VEX.128 Encoded Instruction)
DEST[127:0] := INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

VPUNPCKLDQ (VEX.256 Encoded Instruction)
DEST[255:0] := INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

VPUNPCKLDQ (EVEX Encoded Instructions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;

IF VL = 128
  THEN Dest[VL-1:0] := INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
  FI;
IF VL = 256
  THEN Dest[VL-1:0] := INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
  FI;
IF VL = 512
  THEN Dest[VL-1:0] := INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
  FI;

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN Dest[i+31:i] := TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *Dest[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          Dest[i+31:i] := 0
      FI
    FI
ENDFOR
DEST[511:0] := INTERLEAVE_QWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] := 0

**PUNPCKLQDQ**
DEST[127:0] := INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] := 0

**VPUNPCKLQDQ (VEX.128 Encoded Instruction)**
DEST[127:0] := INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] := 0

**VPUNPCKLQDQ (VEX.256 Encoded Instruction)**
DEST[255:0] := INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] := 0

**VPUNPCKLQDQ (EVEX Encoded Instructions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] := INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i __m256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m256i __m256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i __mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i __mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLDW __m512i __m512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLDW __m512i __m512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLDW __m256i __m256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLDW __m256i __m256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLDW __m128i __mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDW __m128i __mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i __m512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i __m512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i __m256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i __m256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i __mm_mask_unpacklo_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i __mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i __m512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i __m512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i __m256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i __m256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i __mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i __mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b);
PUNPCKLBW __m64 __mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW __m128i __mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW __m256i __m256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLDW __m64 __mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLDW __m128i __mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLDW __m256i __m256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ __m64 __mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ __m128i __mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ __m256i __m256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLDQ __m128i __mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLDQ __m256i __m256_unpacklo_epi64 (__m256i m1, __m256i m2)

Flags Affected
None.

Numeric Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPUNPCKLDQ/QDQ, see Table 2-50, “Type E4NF Class Exception Conditions.”
EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF:nb in Table 2-50, “Type E4NF Class Exception Conditions.”
**PUSH—Push Word, Doubleword, or Quadword Onto the Stack**

<table>
<thead>
<tr>
<th>Opcode 1</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF /6</td>
<td>PUSH r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m32</td>
<td>M</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r/m64.</td>
</tr>
<tr>
<td>50+rw</td>
<td>PUSH r16</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r16.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r32</td>
<td>O</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r32.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r64</td>
<td>O</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r64.</td>
</tr>
<tr>
<td>6B lw</td>
<td>PUSH imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Push imm16.</td>
</tr>
<tr>
<td>6B id</td>
<td>PUSH imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Push imm32.</td>
</tr>
<tr>
<td>0E</td>
<td>PUSH CS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push CS.</td>
</tr>
<tr>
<td>16</td>
<td>PUSH SS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push SS.</td>
</tr>
<tr>
<td>1E</td>
<td>PUSH DS</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push DS.</td>
</tr>
<tr>
<td>06</td>
<td>PUSH ES</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push ES.</td>
</tr>
<tr>
<td>0F A0</td>
<td>PUSH FS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push FS.</td>
</tr>
<tr>
<td>0F A8</td>
<td>PUSH GS</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push GS.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. See the IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>I</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- **Address size.** The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).
  
The address size is used only when referencing a source operand in memory.

- **Operand size.** The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).
  
The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Intel Core and Intel Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.
• Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

(* See Description section for possible sign-extension or zero-extension of source operand and for *)
(* a case in which the size of the memory store may be smaller than the instruction's operand size *)

IF StackAddrSize = 64
THEN
  IF OperandSize = 64
  THEN
    RSP := RSP - 8;
    Memory[SS:RSP] := SRC; (* push quadword *)
  ELSE IF OperandSize = 32
  THEN
    RSP := RSP - 4;
    Memory[SS:RSP] := SRC; (* push dword *)
  ELSE (* OperandSize = 16 *)
    RSP := RSP - 2;
    Memory[SS:RSP] := SRC; (* push word *)
  FI;
ELSE IF StackAddrSize = 32
THEN
  IF OperandSize = 64
  THEN
    ESP := ESP - 8;
    Memory[SS:ESP] := SRC; (* push quadword *)
  ELSE IF OperandSize = 32
  THEN
    ESP := ESP - 4;
    Memory[SS:ESP] := SRC; (* push dword *)
  ELSE (* OperandSize = 16 *)
    ESP := ESP - 2;
    Memory[SS:ESP] := SRC; (* push word *)
  FI;
PUSH—Push Word, Doubleword, or Quadword Onto the Stack

```
FI;
ELSE (* StackAddrSize = 16 *)
IF OperandSize = 32
    THEN
        SP := SP – 4;
        Memory[SS:SP] := SRC; (* push dword *)
    ELSE (* OperandSize = 16 *)
        SP := SP – 2;
        Memory[SS:SP] := SRC; (* push word *)
    FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0)     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)     If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0)     If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD        If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP        If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS        If a memory operand effective address is outside the SS segment limit.
If the new value of the SP or ESP register is outside the stack segment limit.
#UD        If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0)     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)     If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0)     If alignment checking is enabled and an unaligned memory reference is made.
#UD        If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0)     If the memory address is in a non-canonical form.
#SS(0)     If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0)     If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD        If the LOCK prefix is used.
If the PUSH is of CS, SS, DS, or ES.
PUSHA/PUSHAD—Push All General-Purpose Registers

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>PUSHA</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push AX, CX, DX, BX, original SP, BP, SI, and DI.</td>
</tr>
<tr>
<td>60</td>
<td>PUSHAD</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.</td>
</tr>
</tbody>
</table>

Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the “Operation” section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

IF 64-bit Mode 
    THEN #UD
FI;
IF OperandSize = 32 (* PUSHAD instruction *) 
    THEN
        Temp := (ESP);
        Push(EAX);
        Push(ECX);
        Push(EDX);
        Push(EBX);
        Push(Temp);
        Push(EBP);
        Push(ESI);
        Push(EDI);
ELSE (* OperandSize = 16, PUSHA instruction *) 
    Temp := (SP);
    Push(AX);
    Push(CX);
    Push(DX);
    Push(BX);
    Push(Temp);
Push(BP);
Push(SI);
Push(DI);
Fl;

Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If in 64-bit mode.
PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register Onto the Stack

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9C</td>
<td>PUSHF</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Push lower 16 bits of EFLAGS.</td>
</tr>
<tr>
<td>9C</td>
<td>PUSHFD</td>
<td>ZO</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push EFLAGS.</td>
</tr>
<tr>
<td>9C</td>
<td>PUSHFQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push RFLAGS.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Decrement the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFD instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction’s default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Operation

IF (PE = 0) or ((VM = 0) or (VM = 1 and IOPL = 3))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
THEN
   IF OperandSize = 32
       THEN
           push (EFLAGS AND 00FCFFFFH);
           (* VM and RF bits are cleared in image stored on the stack *)
       ELSE
           push (EFLAGS); (* Lower 16 bits only *)

PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register Onto the Stack

Vol. 2B 4-521
PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register Onto the Stack

FI;
ELSE IF 64-bit MODE (* In 64-bit Mode *)
IF OperandSize = 64
THEN
    push (RFLAGS AND 00000000_00FCFFFFH);
    (* VM and RF bits are cleared in image stored on the stack; *)
ELSE
    push (EFLAGS); (* Lower 16 bits only *)
FI;
ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
IF (CR4.VME = 0) OR (OperandSize = 32)
THEN #GP(0); (* Trap to virtual-8086 monitor *)
ELSE
    tempFLAGS = EFLAGS[15:0];
    tempFLAGS[9] = tempFLAGS[19]; (* VIF replaces IF *)
    tempFlags[13:12] = 3; (* IOPL is set to 3 in image stored on the stack *)
    push (tempFLAGS);
FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#SS(0) If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the I/O privilege level is less than 3.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while alignment checking is enabled.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD If the LOCK prefix is used.
PXOR—Logical Exclusive OR

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F EF /r</td>
<td>V/V</td>
<td>MMX</td>
<td>Bitwise XOR of mm/m64 and mm.</td>
</tr>
<tr>
<td>66 0F EF /r</td>
<td>V/V</td>
<td>SSE2</td>
<td>Bitwise XOR of xmm2/m128 and xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG /r</td>
<td>V/V</td>
<td>AVX</td>
<td>Bitwise XOR of xmm3/m128 and xmm2.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG /r</td>
<td>V/V</td>
<td>AVX2</td>
<td>Bitwise XOR of ymm3/m256 and ymm2.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W0 /r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 /r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 /r</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 /r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 /r</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 /r</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.
128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PXOR (64-bit Operand)
DEST := DEST XOR SRC

PXOR (128-bit Legacy SSE Version)
DEST := DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)

VPXOR (VEX.128 Encoded Version)
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:128] := 0

VPXOR (VEX.256 Encoded Version)
DEST := SRC1 XOR SRC2
DEST[MAXVL-1:256] := 0

VPXORD (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0]
            ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
VPXORQ (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[63:0]
            ELSE DEST[i+63:i] := SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            DEST[63:0] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32(__mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32(__mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32(__mmask8 m, __m128i a, __m128i b)
VPXORD __m512i _mm512_xor_epi64(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi64(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b)
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 (__m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, ”Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, ”Type E4 Class Exception Conditions.”
## RCL/RCR/ROL/ROR—Rotate

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /2</td>
<td>RCL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) left once.</td>
</tr>
<tr>
<td>REX + D0 /2</td>
<td>RCL r/m8b, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) left once.</td>
</tr>
<tr>
<td>D2 /2</td>
<td>RCL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) left CL times.</td>
</tr>
<tr>
<td>REX + D2 /2</td>
<td>RCL r/m8b, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) left CL times.</td>
</tr>
<tr>
<td>C0 /2 lb</td>
<td>RCL r/m8, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) left imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /2 lb</td>
<td>RCL r/m8b, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) left imm8 times.</td>
</tr>
<tr>
<td>D1 /2</td>
<td>RCL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) left once.</td>
</tr>
<tr>
<td>D3 /2</td>
<td>RCL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) left CL times.</td>
</tr>
<tr>
<td>C1 /2 lb</td>
<td>RCL r/m16, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) left imm8 times.</td>
</tr>
<tr>
<td>D1 /2</td>
<td>RCL r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) left once.</td>
</tr>
<tr>
<td>REX.W + D1 /2</td>
<td>RCL r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) left once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /2</td>
<td>RCL r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) left CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /2</td>
<td>RCL r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) left CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /2 lb</td>
<td>RCL r/m32, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) left imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /2 lb</td>
<td>RCL r/m64, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) left imm8 times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D0 /3</td>
<td>RCR r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) right once.</td>
</tr>
<tr>
<td>REX + D0 /3</td>
<td>RCR r/m8b, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) right once.</td>
</tr>
<tr>
<td>D2 /3</td>
<td>RCR r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) right CL times.</td>
</tr>
<tr>
<td>REX + D2 /3</td>
<td>RCR r/m8b, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) right CL times.</td>
</tr>
<tr>
<td>C0 /3 lb</td>
<td>RCR r/m8, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 9 bits (CF, r/m8) right imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /3 lb</td>
<td>RCR r/m8b, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 9 bits (CF, r/m8) right imm8 times.</td>
</tr>
<tr>
<td>D1 /3</td>
<td>RCR r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) right once.</td>
</tr>
<tr>
<td>D3 /3</td>
<td>RCR r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) right CL times.</td>
</tr>
<tr>
<td>C1 /3 lb</td>
<td>RCR r/m16, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 17 bits (CF, r/m16) right imm8 times.</td>
</tr>
<tr>
<td>D1 /3</td>
<td>RCR r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) right once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>REX.W + D1 /3</td>
<td>RCR r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) right once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /3</td>
<td>RCR r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) right CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /3</td>
<td>RCR r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) right CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /3 lb</td>
<td>RCR r/m32, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 33 bits (CF, r/m32) right imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /3 lb</td>
<td>RCR r/m64, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 65 bits (CF, r/m64) right imm8 times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D0 /0</td>
<td>ROL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 left once.</td>
</tr>
<tr>
<td>REX + D0 /0</td>
<td>ROL r/m8b, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 left once</td>
</tr>
<tr>
<td>D2 /0</td>
<td>ROL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 left CL times.</td>
</tr>
<tr>
<td>REX + D2 /0</td>
<td>ROL r/m8b, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 left CL times.</td>
</tr>
<tr>
<td>C0 /0 lb</td>
<td>ROL r/m8, imm8</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 left imm8 times.</td>
</tr>
</tbody>
</table>
## Instruction Set Reference, M-U

### RCL/RCR/ROL/ROR—Rotate

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REX + C0 /0 ib</td>
<td>ROL r/m8&lt;sup&gt;2&lt;/sup&gt;, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 left imm8 times.</td>
</tr>
<tr>
<td>D1 /0</td>
<td>ROL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 left once.</td>
</tr>
<tr>
<td>D3 /0</td>
<td>ROL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 left CL times.</td>
</tr>
<tr>
<td>C1 /0 ib</td>
<td>ROL r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 left imm8 times.</td>
</tr>
<tr>
<td>D1 /0</td>
<td>ROL r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 left once.</td>
</tr>
<tr>
<td>REX.W + D1 /0</td>
<td>ROL r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 left once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /0</td>
<td>ROL r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 left CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /0</td>
<td>ROL r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 left CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /0 ib</td>
<td>ROL r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 left imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /0 ib</td>
<td>ROL r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 left imm8 times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D0 /1</td>
<td>ROR r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 right once.</td>
</tr>
<tr>
<td>REX + D0 /1</td>
<td>ROR r/m8&lt;sup&gt;2&lt;/sup&gt;, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 right once.</td>
</tr>
<tr>
<td>D2 /1</td>
<td>ROR r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m8 right CL times.</td>
</tr>
<tr>
<td>REX + D2 /1</td>
<td>ROR r/m8&lt;sup&gt;2&lt;/sup&gt;, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m8 right CL times.</td>
</tr>
<tr>
<td>C0 /1 ib</td>
<td>ROR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 8 bits r/m16 right imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /1 ib</td>
<td>ROR r/m8&lt;sup&gt;2&lt;/sup&gt;, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 8 bits r/m16 right imm8 times.</td>
</tr>
<tr>
<td>D1 /1</td>
<td>ROR r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 right once.</td>
</tr>
<tr>
<td>D3 /1</td>
<td>ROR r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 right CL times.</td>
</tr>
<tr>
<td>C1 /1 ib</td>
<td>ROR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 16 bits r/m16 right imm8 times.</td>
</tr>
<tr>
<td>D1 /1</td>
<td>ROR r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 right once.</td>
</tr>
<tr>
<td>REX.W + D1 /1</td>
<td>ROR r/m64, 1</td>
<td>M1</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 right once. Uses a 6 bit count.</td>
</tr>
<tr>
<td>D3 /1</td>
<td>ROR r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 right CL times.</td>
</tr>
<tr>
<td>REX.W + D3 /1</td>
<td>ROR r/m64, CL</td>
<td>MC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 right CL times. Uses a 6 bit count.</td>
</tr>
<tr>
<td>C1 /1 ib</td>
<td>ROR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Rotate 32 bits r/m32 right imm8 times.</td>
</tr>
<tr>
<td>REX.W + C1 /1 ib</td>
<td>ROR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Rotate 64 bits r/m64 right imm8 times. Uses a 6 bit count.</td>
</tr>
</tbody>
</table>

### Instruction Operands Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>ModRM&lt;sup&gt;r/m (w)&lt;/sup&gt;</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MC</td>
<td>ModRM&lt;sup&gt;r/m (w)&lt;/sup&gt;</td>
<td>CL</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM&lt;sup&gt;r/m (w)&lt;/sup&gt;</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### NOTES:

1. See the IA-32Architecture Compatibility section below.
2. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
Description
Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility
The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation
(* RCL and RCR Instructions *)
SIZE := OperandSize;
CASE (determine count) OF
  SIZE := 8: tempCOUNT := (COUNT AND 1FH) MOD 9;
  SIZE := 16: tempCOUNT := (COUNT AND 1FH) MOD 17;
  SIZE := 32: tempCOUNT := COUNT AND 1FH;
  SIZE := 64: tempCOUNT := COUNT AND 3FH;
ESAC;
IF OperandSize = 64
  THEN COUNTMASK = 3FH;
  ELSE COUNTMASK = 1FH;
FI;

(* RCL Instruction Operation *)
WHILE (tempCOUNT ≠ 0)
  DO
    tempCF := MSB(DEST);
    DEST := (DEST * 2) + CF;
    CF := tempCF;
    tempCOUNT := tempCOUNT – 1;
  OD;
ELIH;
IF (COUNT & COUNTMASK) = 1
  THEN OF := MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
(* RCR Instruction Operation *)
IF (COUNT & COUNTMASK) = 1
   THEN OF := MSB(DEST) XOR CF;
   ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
   DO
      tempCF := LSB(SRC);
      DEST := (DEST / 2) + (CF * 2^SIZE);
      CF := tempCF;
      tempCOUNT := tempCOUNT – 1;
   OD;
(* ROL Instruction Operation *)
tempCOUNT := (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
   DO
      tempCF := MSB(DEST);
      DEST := (DEST * 2) + tempCF;
      tempCOUNT := tempCOUNT – 1;
   OD;
ELIHw;
IF (COUNT & COUNTMASK) ≠ 0
   THEN CF := LSB(DEST);
   FI;
IF (COUNT & COUNTMASK) = 1
   THEN OF := MSB(DEST) XOR CF;
   ELSE OF is undefined;
FI;
(* ROR Instruction Operation *)
tempCOUNT := (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
   DO
      tempCF := LSB(SRC);
      DEST := (DEST / 2) + (tempCF * 2^SIZE);
      tempCOUNT := tempCOUNT – 1;
   OD;
ELIHw;
IF (COUNT & COUNTMASK) ≠ 0
   THEN CF := MSB(DEST);
   FI;
IF (COUNT & COUNTMASK) = 1
   THEN OF := MSB(DEST) XOR MSB − 1(DEST);
   ELSE OF is undefined;
FI;

Flags Affected
For RCL and RCR instructions, a zero-bit rotate does nothing, i.e., affects no flags. For ROL and ROR instructions, if the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined.
For all instructions, the CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

**Protected Mode Exceptions**

- **#GP(0)** If the source operand is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the source operand is located in a non-writable segment.
- If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
### RCPPS—Compute Reciprocals of Packed Single Precision Floating-Point Values

#### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 53 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocals of the packed single precision floating-point values in xmm2/m128 and stores the results in xmm1.</td>
</tr>
<tr>
<td>RCPPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocals of the packed single precision floating-point values in xmm2/m128 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 53 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of packed single precision values in xmm2/mem and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEXCPPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of packed single precision values in xmm2/mem and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 53 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of packed single precision values in ymm2/mem and stores the results in ymm1.</td>
</tr>
<tr>
<td>VRCPPS ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of packed single precision values in ymm2/mem and stores the results in ymm1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Performs a SIMD computation of the approximate reciprocals of the four packed single precision floating-point values in the source operand (second operand) stores the packed single precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD single precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 \times 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, “Numeric Underflow Exception (#U)” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $[1.11111111101100000000000000B \times 2^{125}]$ are guaranteed to not produce tiny results; input values less than or equal to $[1.00000000000110000000001B \times 2^{126}]$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
**Operation**

**RCPPS (128-bit Legacy SSE Version)**
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[95:64] := APPROXIMATE(1/SRC[95:64])
DEST[127:96] := APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

**VRCPSS (VEX.128 Encoded Version)**
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[95:64] := APPROXIMATE(1/SRC[95:64])
DEST[127:96] := APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] := 0

**VRCPSS (VEX.256 Encoded Version)**
DEST[31:0] := APPROXIMATE(1/SRC[31:0])
DEST[95:64] := APPROXIMATE(1/SRC[95:64])
DEST[127:96] := APPROXIMATE(1/SRC[127:96])
DEST[159:128] := APPROXIMATE(1/SRC[159:128])
DEST[255:224] := APPROXIMATE(1/SRC[255:224])

**Intel C/C++ Compiler Intrinsic Equivalent**
RCCPS __m128 _mm_rcp_ps(__m128 a)
RCPPS __m256 _mm256_rcp_ps (__m256 a);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-21, "Type 4 Class Exception Conditions," additionally:
#UD If VEX.vvvv ≠ 1111B.
RCPSS—Compute Reciprocal of Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 53 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocal of the scalar single precision floating-point value in xmm2/m32 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 53 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocal of the scalar single precision floating-point value in xmm3/m32 and stores the result in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/ En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes of an approximate reciprocal of the low single precision floating-point value in the source operand (second operand) and stores the single precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a scalar single precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 \times 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, “Numeric Underflow Exception (#U)” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $[1.1111111110100000000000B \times 2^{125}]$ are guaranteed to not produce tiny results; input values less than or equal to $[0.0000000000110000000001B \times 2^{126}]$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation**

**RCPSS (128-bit Legacy SSE Version)**

DEST[31:0] := APPROXIMATE(1/SRC[31:0])

DEST[MAXVL-1:32] (Unmodified)
VRCPSS (VEX.128 Encoded Version)
DEST[31:0] := APPROXIMATE(1/SRC2[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
RCPSS __m128 _mm_rcp_ss(__m128 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-22, “Type 5 Class Exception Conditions.”
RDFSBASE/RDGSBASE—Read FS/GS Segment Base

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F AE /0 RDFSBASE r32</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 32-bit destination register with the FS base address.</td>
</tr>
<tr>
<td>F3 REX.W OF AE /0 RDGSBASE r64</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 64-bit destination register with the FS base address.</td>
</tr>
<tr>
<td>F3 0F AE /1 RDGSBASE r32</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 32-bit destination register with the GS base address.</td>
</tr>
<tr>
<td>F3 REX.W OF AE /1 RDGSBASE r64</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the 64-bit destination register with the GS base address.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Loads the general-purpose register indicated by the ModR/M:r/m field with the FS or GS segment base address. The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

Operation

DEST := FS/GS segment base address;

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDFSBASE unsigned int _readfsbase_u32(void);
RDFSBASE unsigned __int64 _readfsbase_u64(void);
RDGSBASE unsigned int _readgsbase_u32(void);
RDGSBASE unsigned __int64 _readgsbase_u64(void);

Protected Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.
64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
   If CR4.FSGSBASE[bit 16] = 0.
   If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0.
RDMSR—Read From Model Specific Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF 32</td>
<td>RDMSR</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Read MSR specified by ECX into EDX:EAX.</td>
</tr>
</tbody>
</table>

NOTES:
1. See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 2, “Model-Specific Registers (MSRs)” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility
The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

Operation
EDX:EAX := MSR[ECX];

Flags Affected
None.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- If the value in ECX specifies a reserved or unimplemented MSR address.
- #UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#GP     If the value in ECX specifies a reserved or unimplemented MSR address.
#UD     If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0)   The RDMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
RDPI D—Read Processor ID

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F C7 /7 RDPID r32</td>
<td>R</td>
<td>N.E./V</td>
<td>RDPID</td>
<td>Read IA32_TSC_AUX into r32.</td>
</tr>
<tr>
<td>F3 0F C7 /7 RDPID r64</td>
<td>R</td>
<td>V/N.E.</td>
<td>RDPID</td>
<td>Read IA32_TSC_AUX into r64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

Operation

DEST := IA32_TSC_AUX

Flags Affected

None.

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.7H:0:ECX.RDPID[bit 22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

1. ModRM.MOD = 011B required
RDPKRU—Read Protection Key Rights for User Pages

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 EE</td>
<td>RDPKRU</td>
<td>ZO</td>
<td>V/V</td>
<td>OSPKE</td>
<td>Reads PKRU into EAX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

**Operation**

IF (ECX = 0)
THEN
EAX := PKRU;
EDX := 0;
ELSE #GP(0);
FI;

**Flags Affected**

None.

**C/C++ Compiler Intrinsic Equivalent**

RDPKRU uint32_t _rdpkru_u32(void);

**Protected Mode Exceptions**

- #GP(0) If ECX ≠ 0.
- #UD If the LOCK prefix is used.
  - If CR4.PKE = 0.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
RDPMC—Read Performance-Monitoring Counters

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 33</td>
<td>RDPMC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Read performance-monitoring counter specified by ECX into EDX:EAX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Reads the contents of the performance monitoring counter (PMC) specified in ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the PMC and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the PMC being read, unimplemented bits returned to EDX:EAX will have value zero.

The width of PMCs on processors supporting architectural performance monitoring (CPUID.0AH:EAX[7:0] ≠ 0) are reported by CPUID.0AH:EAX[23:16]. On processors that do not support architectural performance monitoring (CPUID.0AH:EAX[7:0]=0), the width of general-purpose performance PMCs is 40 bits, while the widths of special-purpose PMCs are implementation specific.

Use of ECX to specify a PMC depends on whether the processor supports architectural performance monitoring:

- If the processor does not support architectural performance monitoring (CPUID.0AH:EAX[7:0]=0), ECX[30:0] specifies the index of the PMC to be read. Setting ECX[31] selects "fast" read mode if supported. In this mode, RDPMC returns bits 31:0 of the PMC in EAX while clearing EDX to zero.
- If the processor does support architectural performance monitoring (CPUID.0AH:EAX[7:0] ≠ 0), ECX[31:16] specifies type of PMC while ECX[15:0] specifies the index of the PMC to be read within that type. The following PMC types are currently defined:
  - General-purpose counters use type 0. The index x (to read IA32_PMCx) must be less than the value enumerated by CPUID.0AH:EAX[15:8] (thus ECX[15:8] must be zero).
  - Fixed-function counters use type 4000H. The index x (to read IA32_FIXED_CTRx) can be used if either CPUID.0AH.EDX[4:0] > x or CPUID.0AH.ECX[x] = 1 (thus ECX[15:5] must be 0).
  - Performance metrics use type 2000H. This type can be used only if IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE[bit 15]=1. For this type, the index in ECX[15:0] is implementation specific.

Specifying an unsupported PMC encoding will cause a general protection exception #GP(0). For PMC details see Chapter 20, "Performance Monitoring," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

When in protected or virtual 8086 mode, the Performance-monitoring Counters Enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.) The PMCs can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the PMC, and the event count is stored in the full EAX and EDX registers. The
RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have PMCs, but they must be read with the RDMSR instruction.

**Operation**

MSCB = Most Significant Counter Bit (* Model-specific *)

\[
\text{IF } (((\text{CR4.PCE} = 1) \text{ or } (\text{CPL} = 0) \text{ or } (\text{CR0.PE} = 0)) \text{ and } (\text{ECX indicates a supported counter})) \\
\quad \text{THEN} \\
\quad \quad \text{EAX} := \text{counter}[31:0]; \\
\quad \quad \text{EDX} := \text{ZeroExtend}(\text{counter}[\text{MSCB}:32]); \\
\quad \quad \text{ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)} \\
\quad \quad \quad \#GP(0); \\
\quad \text{FI;}
\]

**Flags Affected**

None.

**Protected Mode Exceptions**

- \#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
- #GP If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0) If the PCE flag in the CR4 register is clear.
- #GP If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
- #GP If an invalid performance counter index is specified.
- #UD If the LOCK prefix is used.
RDRAND—Read Random Number

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF x 0F C7 /6 RDRAND r16</td>
<td>M V/V</td>
<td>RDRAND</td>
<td>Read a 16-bit random number and store in the destination register.</td>
<td></td>
</tr>
<tr>
<td>NF x 0F C7 /6 RDRAND r32</td>
<td>M V/V</td>
<td>RDRAND</td>
<td>Read a 32-bit random number and store in the destination register.</td>
<td></td>
</tr>
<tr>
<td>NF x REX.W + 0F C7 /6 RDRAND r64</td>
<td>M V/I</td>
<td>RDRAND</td>
<td>Read a 64-bit random number and store in the destination register.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Section 7.3.17, “Random Number Generator Instructions”).

This instruction is available at all privilege levels.

In 64-bit mode, the instruction’s default operand size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF HW_RND_GEN.ready = 1
THEN
    CASE of
        operand size is 64: DEST[63:0] := HW_RND_GEN.data;
        operand size is 32: DEST[31:0] := HW_RND_GEN.data;
        operand size is 16: DEST[15:0] := HW_RND_GEN.data;
    ESAC
    CF := 1;
ELSE
    CASE of
        operand size is 64: DEST[63:0] := 0;
        operand size is 32: DEST[31:0] := 0;
        operand size is 16: DEST[15:0] := 0;
    ESAC
    CF := 0;
FI
OF, SF, ZF, AF, PF := 0;

Flags Affected

The CF flag is set according to the result (see the “Operation” section above). The OF, SF, ZF, AF, and PF flags are set to 0.
Intel C/C++ Compiler Intrinsic Equivalent

RDRAND int __rdrand16_step( unsigned short * );
RDRAND int __rdrand32_step( unsigned int * );
RDRAND int __rdrand64_step( unsigned __int64 * );

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.01H:ECX.RDRAND[bit 30] = 0.

Real-Address Mode Exceptions
Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
RDSEED—Read Random SEED

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFx 0F C7 /7 RDSEED r16</td>
<td>M</td>
<td>V/V</td>
<td>RDSEED</td>
<td>Read a 16-bit NIST SP800-90B &amp; C compliant random value and store in the destination register.</td>
</tr>
<tr>
<td>NFx 0F C7 /7 RDSEED r32</td>
<td>M</td>
<td>V/V</td>
<td>RDSEED</td>
<td>Read a 32-bit NIST SP800-90B &amp; C compliant random value and store in the destination register.</td>
</tr>
<tr>
<td>NFx REX.W + 0F C7 /7 RDSEED r64</td>
<td>M</td>
<td>V/I</td>
<td>RDSEED</td>
<td>Read a 64-bit NIST SP800-90B &amp; C compliant random value and store in the destination register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM.r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operand size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[
\text{IF} \text{Hw\_NRND\_GEN}\text{.ready} = 1 \\
\text{THEN} \\
\text{CASE of} \\
\text{operand size is 64: DEST}[63:0] := \text{Hw\_NRND\_GEN}\text{.data}; \\
\text{operand size is 32: DEST}[31:0] := \text{Hw\_NRND\_GEN}\text{.data}; \\
\text{operand size is 16: DEST}[15:0] := \text{Hw\_NRND\_GEN}\text{.data}; \\
\text{ESAC;} \\
\text{CF} := 1; \\
\text{ELSE} \\
\text{CASE of} \\
\text{operand size is 64: DEST}[63:0] := 0; \\
\text{operand size is 32: DEST}[31:0] := 0; \\
\text{operand size is 16: DEST}[15:0] := 0; \\
\text{ESAC;} \\
\text{CF} := 0; \\
\text{FI;} \\
\text{OF, SF, ZF, AF, PF} := 0; \
\]

**Instruction Set Reference, M-U**
Flags Affected
The CF flag is set according to the result (see the “Operation” section above). The OF, SF, ZF, AF, and PF flags are set to 0.

C/C++ Compiler Intrinsic Equivalent
RDSEED int _rdseed16_step( unsigned short * );
RDSEED int _rdseed32_step( unsigned int * );
RDSEED int _rdseed64_step( unsigned __int64 * );

Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.
RDSSPD/RDSSPQ—Read Shadow Stack Pointer

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 1E /1 (mod=11) RDSSPD r32</td>
<td>R</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Copy low 32 bits of shadow stack pointer (SSP) to r32.</td>
</tr>
<tr>
<td>F3 REX.W 0F 1E /1 (mod=11) RDSSPQ r64</td>
<td>R</td>
<td>V/N.E.</td>
<td>CET_SS</td>
<td>Copies shadow stack pointer (SSP) to r64.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description
Copies the current shadow stack pointer (SSP) register to the register destination. This opcode is a NOP when CET shadow stacks are not enabled and on processors that do not support CET.

### Operation

IF CPL = 3
  IF CR4.CET & IA32_U.CET.SH_STK_EN
    IF (operand size is 64 bit)
      THEN
        Dest := SSP;
      ELSE
        Dest := SSP[31:0];
    FI;
  FI;
ELSE
  IF CR4.CET & IA32_S.CET.SH_STK_EN
    IF (operand size is 64 bit)
      THEN
        Dest := SSP;
      ELSE
        Dest := SSP[31:0];
    FI;
  FI;

### Flags Affected
None.

### C/C++ Compiler Intrinsic Equivalent

RDSSPD__int32 _rdsspd_i32(void);
RDSSPQ__int64 _rdsspq_i64(void);

### Protected Mode Exceptions
None.
Real-Address Mode Exceptions
None.

Virtual-8086 Mode Exceptions
None.

Compatibility Mode Exceptions
None.

64-Bit Mode Exceptions
None.
RDTSC—Read Time-Stamp Counter

### Description

Reads the current value of the processor’s time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See “Time Stamp Counter” in Chapter 18 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSC:

- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads are globally visible, it can execute LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed only after all previous instructions have executed and all previous loads and stores are globally visible, it can execute the sequence MFENCE;LFENCE immediately before RDTSC.
- If software requires RDTSC to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute the sequence LFENCE immediately after RDTSC.

This instruction was introduced by the Pentium processor.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```plaintext
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX := TimeStampCounter;
ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

### Flags Affected

None.

---

1. A load is considered to become globally visible when the value to be loaded is determined.
Protected Mode Exceptions
#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the TSD flag in register CR4 is set.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**RDTSCP—Read Time-Stamp Counter and Processor ID**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 F9</td>
<td>RDTSCP</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Reads the current value of the processor’s time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 18 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible. But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. The following items may guide software seeking to order executions of RDTSCP:

- If software requires RDTSCP to be executed only after all previous stores are globally visible, it can execute MFENCE immediately before RDTSCP.
- If software requires RDTSCP to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute LFENCE immediately after RDTSCP.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

**Operation**

IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
THEN
    EDX:EAX := TimeStampCounter;
    ECX := IA32_TSC_AUX[31:0];
ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;

**Flags Affected**

None.

---

1. A load is considered to become globally visible when the value to be loaded is determined.
**Protected Mode Exceptions**

- #GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.
- #UD If the LOCK prefix is used.
  
  If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

**Real-Address Mode Exceptions**

- #UD If the LOCK prefix is used.
  
  If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

**Virtual-8086 Mode Exceptions**

- #GP(0) If the TSD flag in register CR4 is set.
- #UD If the LOCK prefix is used.
  
  If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 6C</td>
<td>REP INS m8, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input (E)CX bytes from port DX into ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 6C</td>
<td>REP INS m8, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Input RCX bytes from port DX into [RDI].</td>
</tr>
<tr>
<td>F3 6D</td>
<td>REP INS m16, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input (E)CX words from port DX into ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 6D</td>
<td>REP INS m32, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Input (E)CX doublewords from port DX into ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 6D</td>
<td>REP INS r/m32, DX</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Input RCX default size from port DX into [RDI].</td>
</tr>
<tr>
<td>F3 A4</td>
<td>REP MOV m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 REX.W A4</td>
<td>REP MOV m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move RCX bytes from [RSI] to [RDI].</td>
</tr>
<tr>
<td>F3 A5</td>
<td>REP MOV m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 A5</td>
<td>REP MOV m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 REX.W A5</td>
<td>REP MOV m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move RCX quadwords from [RSI] to [RDI].</td>
</tr>
<tr>
<td>F3 6E</td>
<td>REP OUTS DX, r/m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output (E)CX bytes from DS:[(E)SI] to port DX.</td>
</tr>
<tr>
<td>F3 REX.W 6E</td>
<td>REP OUTS DX, r/m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Output RCX bytes from [RSI] to port DX.</td>
</tr>
<tr>
<td>F3 6F</td>
<td>REP OUTS DX, r/m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output (E)CX words from DS:[(E)SI] to port DX.</td>
</tr>
<tr>
<td>F3 6F</td>
<td>REP OUTS DX, r/m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Output (E)CX doublewords from DS:[(E)SI] to port DX.</td>
</tr>
<tr>
<td>F3 REX.W 6F</td>
<td>REP OUTS DX, r/m32</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Output RCX default size from [RSI] to port DX.</td>
</tr>
<tr>
<td>F3 AC</td>
<td>REP LODS AL</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Load (E)CX bytes from DS:[(E)SI] to AL.</td>
</tr>
<tr>
<td>F3 REX.W AC</td>
<td>REP LODS AL</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load RCX bytes from [RSI] to AL.</td>
</tr>
<tr>
<td>F3 AD</td>
<td>REP LODS AX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Load (E)CX words from DS:[(E)SI] to AX.</td>
</tr>
<tr>
<td>F3 AD</td>
<td>REP LODS EAX</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Load (E)CX doublewords from DS:[(E)SI] to EAX.</td>
</tr>
<tr>
<td>F3 REX.W AD</td>
<td>REP LODS RAX</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Load RCX quadwords from [RSI] to EAX.</td>
</tr>
<tr>
<td>F3 AA</td>
<td>REP STOS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fill (E)CX bytes at ES:[(E)DI] with AL.</td>
</tr>
<tr>
<td>F3 REX.W AA</td>
<td>REP STOS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Fill RCX bytes at [RDI] with AL.</td>
</tr>
<tr>
<td>F3 AB</td>
<td>REP STOS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fill (E)CX words at ES:[(E)DI] with AX.</td>
</tr>
<tr>
<td>F3 AB</td>
<td>REP STOS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fill (E)CX doublewords at ES:[(E)DI] with EAX.</td>
</tr>
<tr>
<td>F3 REX.W AB</td>
<td>REP STOS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Fill RCX quadwords at [RDI] with RAX.</td>
</tr>
<tr>
<td>F3 A6</td>
<td>REP CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F3 REX.W A6</td>
<td>REP CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-matching bytes in [RDI] and [RSI].</td>
</tr>
<tr>
<td>F3 A7</td>
<td>REP CMPS m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F3 A7</td>
<td>REP CMPS m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F3 REX.W A7</td>
<td>REP CMPS m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-matching quadwords in [RDI] and [RSI].</td>
</tr>
<tr>
<td>F3 AE</td>
<td>REP SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find non-AL byte starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 REX.W AE</td>
<td>REP SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-AL byte starting at [RDI].</td>
</tr>
<tr>
<td>F3 AF</td>
<td>REP SCAS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find non-AX word starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F3 AF</td>
<td>REP SCAS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find non-EAX doubleword starting at ES:[(E)DI].</td>
</tr>
</tbody>
</table>
Rep/Repe/Repz/Repne/Repnz—Repeat String Operation Prefix

Description

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The F3H prefix is defined for the following instructions and undefined for the rest:

- F3H as REP/REPE/REPZ for string and input/output instruction.
- F3H is a mandatory prefix for POPCNT, LZCNT, and ADOX.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-17.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 REX.W AF</td>
<td>REPE SCAS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find non-RAX quadword starting at [RDI].</td>
</tr>
<tr>
<td>F2 A6</td>
<td>REPNE CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find matching bytes in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F2 REX.W A6</td>
<td>REPNE CMPS m8, m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find matching bytes in [RDI] and [RSI].</td>
</tr>
<tr>
<td>F2 A7</td>
<td>REPNE CMPS m16, m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find matching words in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F2 A7</td>
<td>REPNE CMPS m32, m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].</td>
</tr>
<tr>
<td>F2 REX.W A7</td>
<td>REPNE CMPS m64, m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find matching doublewords in [RDI] and [RSI].</td>
</tr>
<tr>
<td>F2 A6</td>
<td>REPNE SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find AL, starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F2 REX.W AE</td>
<td>REPNE SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find AL, starting at [RDI].</td>
</tr>
<tr>
<td>F2 AF</td>
<td>REPNE SCAS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find AX, starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F2 AF</td>
<td>REPNE SCAS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Find EAX, starting at ES:[(E)DI].</td>
</tr>
<tr>
<td>F2 REX.W AF</td>
<td>REPNE SCAS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Find RAX, starting at [RDI].</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 4-17. Repeat Prefixes**

<table>
<thead>
<tr>
<th>Repeat Prefix</th>
<th>Termination Condition 1*</th>
<th>Termination Condition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>RCX or (E)CX = 0</td>
<td>None</td>
</tr>
<tr>
<td>REP/REPEZ</td>
<td>RCX or (E)CX = 0</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>RCX or (E)CX = 0</td>
<td>ZF = 1</td>
</tr>
</tbody>
</table>

**NOTES:**
* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.
The REP/REPE/REPZ, and REPNE/REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

REP INS may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g., #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

**Operation**

IF AddressSize = 16
THEN
   Use CX for CountReg;
   Implicit Source/Dest operand for memory use of SI/DI;
ELSE IF AddressSize = 64
THEN Use RCX for CountReg;
   Implicit Source/Dest operand for memory use of RSI/RDI;
ELSE
   Use ECX for CountReg;
   Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
   DO
      Service pending interrupts (if any);
      Execute associated string instruction;
      CountReg := (CountReg – 1);
      IF CountReg = 0
         THEN exit WHILE loop; Fl;
      IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
         or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
         THEN exit WHILE loop; Fl;
   OD;

**Flags Affected**

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.
Exceptions (All Operating Modes)
Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
**RET—Return From Procedure**

**Description**

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.

- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.

- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure’s stack and the calling procedure’s stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e., 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.
Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions," and Chapter 17, "Control-flow Enforcement Technology (CET),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for CET details.

**Instruction ordering.** Instructions following a far return may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far return have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

Unlike near indirect CALL and near indirect JMP, the processor will not speculatively execute the next sequential instruction after a near RET unless that instruction is also the target of a jump or is a target in a branch predictor.

**Operation**

(* Near return *)

IF instruction = near return

THEN;

IF OperandSize = 32

THEN

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP := Pop();

IF ShadowStackEnabled(CPL)

tempSSEIP = ShadowStackPop4B();

IF EIP != TempSSeIP

THEN #CP(NEAR_RET); FI;

FI;
ELSE

IF OperandSize = 64

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

RIP := Pop();

IF ShadowStackEnabled(CPL)

tempSSeIP = ShadowStackPop8B();

IF RIP != tempSSeIP

THEN #CP(NEAR_RET); FI;

FI;
ELSE (* OperandSize = 16 *)

IF top 2 bytes of stack not within stack limits

THEN #SS(0); FI;

tempEIP := Pop();

tempEIP := tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits

THEN #GP(0); FI;

EIP := tempEIP;

IF ShadowStackEnabled(CPL)

tempSSeIP = ShadowStackPop4B();

IF EIP != tempSSeIP

THEN #CP(NEAR_RET); FI;

FI;
ELSE (* OperandSize = 16 *)

IF top 2 bytes of stack not within stack limits

THEN #SS(0); FI;

tempEIP := Pop();

tempEIP := tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits

THEN #GP(0); FI;

EIP := tempEIP;

IF ShadowStackEnabled(CPL)

tempSSeIP = ShadowStackPop4B();

IF EIP != tempSSeIP

THEN #CP(NEAR_RET); FI;

FI;

FI;

IF instruction has immediate operand

THEN (* Release parameters from stack *)

IF StackAddressSize = 32
THEN
    ESP := ESP + SRC;
ELSE
    IF StackAddressSize = 64
        THEN
            RSP := RSP + SRC;
        ELSE (* StackAddressSize = 16 *)
            SP := SP + SRC;
        FI;
    FI;
    FI;
    FI;
    FI;
( * Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
THEN
    IF OperandSize = 32
        THEN
            IF top 8 bytes of stack not within stack limits
                THEN #SS(0); FI;
            EIP := Pop();
            CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE (* OperandSize = 16 *)
            IF top 4 bytes of stack not within stack limits
                THEN #SS(0); FI;
            tempEIP := Pop();
            tempEIP := tempEIP AND 0000FFFFH;
            IF tempEIP not within code segment limits
                THEN #GP(0); FI;
            EIP := tempEIP;
            CS := Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP := SP + (SRC AND FFFFH);
        FI;
    FI;
( * Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
THEN
    IF OperandSize = 32
        THEN
            IF second doubleword on stack is not within stack limits
                THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
            FI;
        IF return code segment selector is NULL
            THEN #GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN #GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
IF return code segment descriptor is not a code segment
    THEN #GP(selector); FI;
IF return code segment selector RPL < CPL
    THEN #GP(selector); FI;
IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is non-conforming and return code
    segment DPL ≠ return code segment selector RPL
    THEN #GP(selector); FI;
IF return code segment descriptor is not present
    THEN #NP(selector); FI;
IF return code segment selector RPL > CPL
    THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
FI;
FI;

RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE (* OperandSize = 16 *)
        EIP := Pop();
        EIP := EIP AND 0000FFFFH;
        CS := Pop(); (* 16-bit pop *)
    FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP := SP + SRC;
            FI;
    FI;
IF ShadowStackEnabled(CPL)
    (* SSP must be 8 byte aligned *)
    IF SSP AND 0x7 ! = 0
        THEN #CP(FAR-RET/IRET); FI;
    tempSsCS = shadow_stack_load 8 bytes from SSP+16;
    tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
    prevSSP = shadow_stack_load 8 bytes from SSP;
    SSP = SSP + 24;
    (* do a 64 bit-compare to check if any bits beyond bit 15 are set *)
    tempCS = CS; (* zero pad to 64 bit *)
    IF tempCS != tempSsCS
        THEN #CP(FAR-RET/IRET); FI;
    (* do a 64 bit-compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
    IF CSBASE + RIP != tempSsLIP
        THEN #CP(FAR-RET/IRET); FI;
RETURN FROM PROCEDURE:

IF prevSSP AND 0x3 != 0
  THEN #CP(FAR-RET/IRET); Fl;

IF prevSSP[63:32] != 0
  THEN #GP(0); Fl;

SSP := prevSSP
Fl;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:

IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
  THEN #SS(0); Fl;

Read return segment selector;
IF stack segment selector is NULL
  THEN #GP(0); Fl;
IF return stack segment selector index is not within its descriptor table limits
  THEN #GP(selector); Fl;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL != RPL of the return code segment selector
  or stack segment is not a writable data segment
or stack segment descriptor DPL != RPL of the return code segment selector
  THEN #GP(selector); Fl;
IF stack segment not present
  THEN #SS(StackSegmentSelector); Fl;
IF the return instruction pointer is not within the return code segment limit
  THEN #GP(0); Fl;

IF OperandSize = 32
  THEN
    EIP := Pop();
    CS := Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
    CS(RPL) := ReturnCodeSegmentSelector(RPL);
    IF instruction has immediate operand
      THEN (* Release parameters from called procedure’s stack *)
        IF StackAddressSize = 32
          THEN
            ESP := ESP + SRC;
          ELSE (* StackAddressSize = 16 *)
            SP := SP + SRC;
        FI;
    FI;
    tempESP := Pop();
    tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
ELSE (* OperandSize = 16 *)
  EIP := Pop();
  EIP := EIP AND 0000FFFFH;
  CS := Pop(); (* 16-bit pop; segment descriptor loaded *)
  CS(RPL) := ReturnCodeSegmentSelector(RPL);
  IF instruction has immediate operand
    THEN (* Release parameters from called procedure’s stack *)
      IF StackAddressSize = 32
        THEN
          ESP := ESP + SRC;
        ELSE (* StackAddressSize = 16 *)
          }
\[ SP := SP + SRC; \]
\[ FI; \]
\[ tempESP := \text{Pop}(); \]
\[ tempSS := \text{Pop}(); (* 16-bit pop; segment descriptor loaded *) \]
\[ FI; \]
\[ \text{IF ShadowStackEnabled}(CPL) \]
\[ (* check if 8 byte aligned *) \]
\[ \text{IF SSP AND 0x7} = 0 \]
\[ THEN \#CP(FAR-RET/IRET); FI; \]
\[ \text{IF ReturnCodeSegmentSelector}(RPL) = 3 \]
\[ THEN \]
\[ \quad \text{tempSSCS} = \text{shadow_stack_load} 8 \text{ bytes from SSP+16}; \]
\[ \quad \text{tempSSLIP} = \text{shadow_stack_load} 8 \text{ bytes from SSP+8}; \]
\[ \quad \text{tempSSP} = \text{shadow_stack_load} 8 \text{ bytes from SSP}; \]
\[ \quad \text{SSP} = \text{SSP} + 24; \]
\[ (* \text{Do 64 bit compare to detect bits beyond 15 being set} *) \]
\[ \quad \text{tempCS} = \text{CS}; (* \text{zero extended to 64 bit} *) \]
\[ \quad \text{IF tempCS} \neq \text{tempSSCS} \]
\[ \quad THEN \#CP(FAR-RET/IRET); FI; \]
\[ (* \text{Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LA} *) \]
\[ \quad \text{IF CSBASE + RIP} \neq \text{tempSSLIP} \]
\[ \quad THEN \#CP(FAR-RET/IRET); FI; \]
\[ (* \text{check if 4 byte aligned} *) \]
\[ \quad \text{IF tempSSP AND 0x3} = 0 \]
\[ \quad THEN \#CP(FAR-RET/IRET); FI; \]
\[ FI; \]
\[ FI; \]
\[ \quad \text{tempOldCPL} = \text{CPL}; \]
\[ \quad \text{CPL} := \text{ReturnCodeSegmentSelector}(RPL); \]
\[ \quad \text{ESP} := \text{tempESP}; \]
\[ \quad \text{SS} := \text{tempSS}; \]
\[ \quad \text{tempOldSSP} = \text{SSP}; \]
\[ \text{IF ShadowStackEnabled}(CPL) \]
\[ \quad \text{IF CPL} = 3 \]
\[ \quad \quad THEN tempSSP := IA32_PL3_SSP; FI; \]
\[ \quad \quad \text{IF tempSSP}[63:32] \neq 0 \]
\[ \quad \quad \quad THEN \#GP(0); FI; \]
\[ \quad \quad \text{SSP} := \text{tempSSP} \]
\[ FI; \]
\[ (* \text{Now past all faulting points; safe to free the token. The token free is done using the old SSP} \]
\[ (* \text{and using a supervisor override as old CPL was a supervisor privilege level} *) \]
\[ \text{IF ShadowStackEnabled}(\text{tempOldCPL}) \]
\[ \quad \text{expected_token_value} = \text{tempOldSSP} \mid \text{BUSY_BIT} (* \text{busy bit - bit position 0 - must be set} *) \]
\[ \quad \text{new_token_value} = \text{tempOldSSP} \]
\[ \quad \quad (* \text{clear the busy bit} *) \]
\[ \quad \quad \text{shadow_stack_lock_cmpxchg8b} \text{(tempOldSSP, new_token_value, expected_token_value)} \]
\[ FI; \]
\[ FI; \]
\[ \text{FOR each SegReg in (ES, FS, GS, and DS)} \]
\[ \quad \text{tempDesc} := \text{descriptor cache for SegReg} (* \text{hidden part of segment register} *) \]
\[ \quad \text{IF (SegmentSelector} == \text{NULL} \text{OR (tempDesc}(DPL) < \text{CPL AND tempDesc}(Type) \text{is} \text{(data or non-conforming code)*)} \]
THEN (* Segment register invalid *)
    SegmentSelector := 0; (*Segment selector becomes null*)
FI;
O;

IF instruction has immediate operand
THEN (* Release parameters from calling procedure's stack *)
    IF StackAddressSize = 32
        THEN
            ESP := ESP + SRC;
        ELSE (* StackAddressSize = 16 *)
            SP := SP + SRC;
        FI;
    FI;

(* IA-32e Mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
THEN
    IF OperandSize = 32
        THEN
            IF second doubleword on stack is not within stack limits
                THEN #SS(0); FI;
            IF first or second doubleword on stack is not in canonical space
                THEN #SS(0); FI;
        ELSE
            IF OperandSize = 16
                THEN
                    IF second word on stack is not within stack limits
                        THEN #SS(0); FI;
                    IF first or second word on stack is not in canonical space
                        THEN #SS(0); FI;
                ELSE (* OperandSize = 64 *)
                    IF first or second quadword on stack is not in canonical space
                        THEN #SS(0); FI;
                FI
            FI;
    FI;
    IF return code segment selector is NULL
        THEN GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN GP(selector); FI;
    IF return code segment selector addresses descriptor in non-canonical space
        THEN GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment descriptor has L-bit = 1 and D-bit = 1
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
        and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming
        and return code segment DPL ≠ return code segment selector RPL
THEN #GP(selector); Fi;
IF return code segment descriptor is not present
THEN #NP(selector); Fi;
IF return code segment selector RPL > CPL
THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
Fi;
FI;

IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
THEN #GP(0); Fi;
IF the return instruction pointer is not within canonical address space
THEN #GP(0); Fi;
IF OperandSize = 32
THEN
  EIP := Pop();
  CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
ELSE
  IF OperandSize = 16
  THEN
    EIP := Pop();
    EIP := EIP AND 0000FFFFH;
    CS := Pop(); (* 16-bit pop *)
  ELSE (* OperandSize = 64 *)
    RIP := Pop();
    CS := Pop(); (* 64-bit pop, high-order 48 bits discarded *)
  FI;
FI;
IF instruction has immediate operand
THEN (* Release parameters from stack *)
IF StackAddressSize = 32
THEN
  ESP := ESP + SRC;
ELSE
  IF StackAddressSize = 16
  THEN
    SP := SP + SRC;
  ELSE (* StackAddressSize = 64 *)
    RSP := RSP + SRC;
  FI;
FI;
IF ShadowStackEnabled(CPL)
IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
THEN #CP(FAR-RET/IRET); Fi;
tempSsCS = shadow_stack_load 8 bytes from SSP+16;
tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
tempSSP = shadow_stack_load 8 bytes from SSP;
SSP = SSP + 24;
tempCS = CS; (* zero padded to 64 bit *)
IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
THEN #CP(FAR-RET/IRET); Fi;
IF CSBASE + RIP != tempSsLIP (* 64 bit compare *)
THEN #CP(FAR-RET/IRET); FI;
IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
    THEN #CP(FAR-RET/IRET); FI;
IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
    (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
    THEN #GP(0); FI;
SSP := tempSSP
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
    FI;
IF return stack segment descriptor is not within descriptor table limits
    THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) := ReturnCodeSegmentSelector(RPL);
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP := ESP + SRC;
                    ELSE
                        IF StackAddressSize = 16
                            THEN
                                SP := SP + SRC;
ELSE (* StackAddressSize = 64 *)
RSP := RSP + SRC;
FI;
FI;
FI;
tempESP := Pop();
tempSS := Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
ELSE
IF OperandSize = 16
THEN
EIP := Pop();
EIP := EIP AND 0000FFFFH;
CS := Pop(); (* 16-bit pop; segment descriptor loaded *)
CS(RPL) := ReturnCodeSegmentSelector(RPL);
IF instruction has immediate operand
THEN (* Release parameters from called procedure's stack *)
IF StackAddressSize = 32
THEN
ESP := ESP + SRC;
ELSE
IF StackAddressSize = 16
THEN
SP := SP + SRC;
ELSE (* StackAddressSize = 64 *)
RSP := RSP + SRC;
FI;
FI;
tempESP := Pop();
tempSS := Pop(); (* 16-bit pop; segment descriptor loaded *)
ELSE (* OperandSize = 64 *)
RIP := Pop();
CS := Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
CS(RPL) := ReturnCodeSegmentSelector(RPL);
IF instruction has immediate operand
THEN (* Release parameters from called procedure's stack *)
RSP := RSP + SRC;
FI;
tempESP := Pop();
tempSS := Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
FI;
FI;
IF ShadowStackEnabled(CPL)
(* check if 8 byte aligned *)
IF SSP AND 0x7 != 0
THEN #CP(FAR-RET/IRET); FI;
IF ReturnCodeSegmentSelector(RPL) != 3
THEN
tempSsCS = shadow_stack_load 8 bytes from SSP+16;
tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
tempSSP = shadow_stack_load 8 bytes from SSP;
SSP = SSP + 24;
(* Do 64 bit compare to detect bits beyond 15 being set *)
tempCS = CS; (* zero padded to 64 bit *)
IF tempCS != tempSsCS
    THEN #CP(FAR-RET/IRET); Fl;
(* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
IF CSBASE + RIP != tempSsLIP
    THEN #CP(FAR-RET/IRET); Fl;
(* check if 4 byte aligned *)
IF tempSSP AND 0x3 != 0
    THEN #CP(FAR-RET/IRET); Fl;
FI;
FI;
tempOldCPL = CPL;
CPL := ReturnCodeSegmentSelector(RPL);
ESP := tempESP;
SS := tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSsP := IA32_PL3_SSP; Fl;
    IF (CS.L = 0 AND tempSSP[63:32] != 0) OR
       (CS.L = 1 AND tempSSP is not canonical relative to the current paging mode)
        THEN #GP(0); Fl;
    SSP := tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
* and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    expected_token_value = tempOldSSP | BUSY_BIT     (* busy bit - bit position 0 - must be set *)
    new_token_value = tempOldSSP                   (* clear the busy bit *)
    shadow_stack_lock_cmpxchg8b(tempOldSSP, new_token_value, expected_token_value)
FI;
FOR each of segment register (ES, FS, GS, and DS)
DO
    IF segment register points to data or non-conforming code segment
       and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
        THEN SegmentSelector := 0; (* SegmentSelector invalid *)
    FI;
OD;
IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP := ESP + SRC;
        ELSE
            IF StackAddressSize = 16
                THEN
                    SP := SP + SRC;
            ELSE (* StackAddressSize = 64 *)
                RSP := RSP + SRC;
            FI;
        FI;
    FI;
FI;

4-568 Vol. 2B
Flags Affected
None.

Protected Mode Exceptions

#GP(0) If the return code or stack segment selector is NULL.
If the return instruction pointer is not within the return code segment limit.
If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.

#GP(selector) If the RPL of the return code segment selector is less then the CPL.
If the return code or stack segment selector index is not within its descriptor table limits.
If the return code segment descriptor does not indicate a code segment.
If the return code segment is non-conforming and the segment selector’s DPL is not equal to the RPL of the code segment’s segment selector
If the return code segment is conforming and the segment selector’s DPL greater than the RPL of the code segment’s segment selector
If the stack segment is not a writable data segment.
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0) If the top bytes of stack are not within stack limits.
If the return stack segment is not present.

#NP(selector) If the return code segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

#CP(Far-RET/IRET) If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
If return instruction pointer from stack and shadow stack do not match.

Real-Address Mode Exceptions

#GP If the return instruction pointer is not within the return code segment limit
#SS If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0) If the return instruction pointer is not within the return code segment limit
#SS(0) If the top bytes of stack are not within stack limits.
#PF(fault-code) If a page fault occurs.
#AC(0) If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions
Same as 64-bit mode exceptions.
64-Bit Mode Exceptions

#GP(0) If the return instruction pointer is non-canonical.
If the return instruction pointer is not within the return code segment limit.
If the stack segment selector is NULL going back to compatibility mode.
If the stack segment selector is NULL going back to CPL3 64-bit mode.
If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
If the return code segment selector is NULL.
If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4GB.

#GP(selector) If the proposed segment descriptor for a code segment does not indicate it is a code segment.
If the proposed new code segment descriptor has both the D-bit and L-bit set.
If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
If CPL is greater than the RPL of the code segment selector.
If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
If a segment selector index is outside its descriptor table limits.
If a segment descriptor memory address is non-canonical.
If the stack segment is not a writable data segment.
If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
If the stack segment selector RPL is not equal to the RPL of the return code segment selector.

#SS(0) If an attempt to pop a value off the stack violates the SS limit.
If an attempt to pop a value off the stack causes a non-canonical address to be referenced.

#NP(selector) If the return code or stack segment is not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#CP(Far-RET/IRET) If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
If return instruction pointer from stack and shadow stack do not match.
RORX — Rotate Right Logical Without Affecting Flags

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.F2.66.0F3A.W0 ib RORX r32, r/m32, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>BMI2</td>
<td>Rotate 32-bit r/m32 right imm8 times without affecting arithmetic flags.</td>
</tr>
<tr>
<td>VEX.LZ.F2.66.0F3A.W1 ib RORX r64, r/m64, imm8</td>
<td>RMI</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Rotate 64-bit r/m64 right imm8 times without affecting arithmetic flags.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description
Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

IF (OperandSize = 32)
- \( y := \text{imm8 AND 1FH} \);  
- \( \text{DEST} := \text{(SRC >> y) | (SRC << (32-y))} \);
ELSEIF (OperandSize = 64)
- \( y := \text{imm8 AND 3FH} \);  
- \( \text{DEST} := \text{(SRC >> y) | (SRC << (64-y))} \);
FI;

### Flags Affected
None.

### Intel C/C++ Compiler Intrinsic Equivalent
Auto-generated from high-level language.

### SIMD Floating-Point Exceptions
None.

### Other Exceptions
See Table 2-29, “Type 13 Class Exception Conditions.”
**ROUNDPD—Round Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/* Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 09 /r ib ROUNDPD xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Round packed double precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A,WIG 09 /r ib VROUNDPD xmm1, xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round packed double precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A,WIG 09 /r ib VROUNDPD ymm1, ymm2/m256, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round packed double precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Round the 2 double precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a double precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Operation

IF (imm[2] = '1)
    THEN // rounding mode is determined by MXCSR.RC
        DEST[63:0] := ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] := ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[63:0] := ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] := ConvertDPFPToInteger_Imm(SRC[127:64]);
FI

ROUNDPD (128-bit Legacy SSE Version)
DEST[63:0] := RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] := RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)

VROUNDPD (VEX.128 Encoded Version)
DEST[63:0] := RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] := RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[MAXVL-1:128] := 0

VROUNDPD (VEX.256 Encoded Version)
DEST[63:0] := RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] := RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[191:128] := RoundToInteger(SRC[191:128], ROUND_CONTROL)
DEST[255:192] := RoundToInteger(SRC[255:192], ROUND_CONTROL)

Table 4-18. Rounding Modes and Encoding of Rounding Control (RC) Field

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>RC Field Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to nearest (even)</td>
<td>00B</td>
<td>Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).</td>
</tr>
<tr>
<td>Round down (toward −∞)</td>
<td>01B</td>
<td>Rounded result is closest to but no greater than the infinitely precise result.</td>
</tr>
<tr>
<td>Round up (toward +∞)</td>
<td>10B</td>
<td>Rounded result is closest to but no less than the infinitely precise result.</td>
</tr>
<tr>
<td>Round toward zero (Truncate)</td>
<td>11B</td>
<td>Rounded result is closest to but no greater in absolute value than the infinitely precise result.</td>
</tr>
</tbody>
</table>

Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction
Intel C/C++ Compiler Intrinsic Equivalent

__m128 __mm_round_pd(__m128d s1, int iRoundMode);
__m128 __mm_floor_pd(__m128d s1);
__m128 __mm_ceil_pd(__m128d s1)
__m256 __mm256_round_pd(__m256d s1, int iRoundMode);
__m256 __mm256_floor_pd(__m256d s1);
__m256 __mm256_ceil_pd(__m256d s1)

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN).
Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by ROUNDPD.

Other Exceptions
See Table 2-19, "Type 2 Class Exception Conditions," additionally:
#UD If VEX.vvvv ≠ 1111B.
ROUNDPS—Round Packed Single Precision Floating-Point Values

Opcode*/
Instruction| Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description
---|---|---|---|---
66 0F 3A 08 /r ib | RMI | V/V | SSE4_1 | Round packed single precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.

VEX.128.66.0F3A.WIG 08 /r ib | RMI | V/V | AVX | Round packed single precision floating-point values in xmm2/m128 and place the result in xmm1. The rounding mode is determined by imm8.

VEX.256.66.0F3A.WIG 08 /r ib | RMI | V/V | AVX | Round packed single precision floating-point values in ymm2/m256 and place the result in ymm1. The rounding mode is determined by imm8.

Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
---|---|---|---|---|
RMI | ModRM:reg (w) | ModRM:r/m (r) | imm8 | N/A |

Description

Round the 4 single precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the second operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

IF (imm[2] = ‘1’)

THEN // rounding mode is determined by MXCSR.RC

DEST[31:0] := ConvertSPPFPToInteger_M(SRC[31:0]);
DEST[63:32] := ConvertSPPFPToInteger_M(SRC[63:32]);
DEST[95:64] := ConvertSPPFPToInteger_M(SRC[95:64]);
DEST[127:96] := ConvertSPPFPToInteger_M(SRC[127:96]);
ELSE // rounding mode is determined by IMM8.RC

END.

ROUNDPS—Round Packed Single Precision Floating-Point Values
DEST[31:0] := ConvertSPFPToInteger_Imm(SRC[31:0]);
DEST[63:32] := ConvertSPFPToInteger_Imm(SRC[63:32]);
DEST[95:64] := ConvertSPFPToInteger_Imm(SRC[95:64]);
DEST[127:96] := ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;

ROUNDPS (128-bit Legacy SSE Version)
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] := RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] := RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] := RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[MAXVL-1:128] := 0

VROUNDPS (VEX.128 Encoded Version)
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] := RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] := RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] := RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[MAXVL-1:128] := 0

VROUNDPS (VEX.256 Encoded Version)
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] := RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] := RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] := RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[159:128] := RoundToInteger(SRC[159:128], ROUND_CONTROL)
DEST[191:160] := RoundToInteger(SRC[191:160], ROUND_CONTROL)
DEST[223:192] := RoundToInteger(SRC[223:192], ROUND_CONTROL)
DEST[255:224] := RoundToInteger(SRC[255:224], ROUND_CONTROL)

Intel C/C++ Compiler Intrinsic Equivalent
__m128 _mm_round_ps(__m128 s1, int iRoundMode);
__m128 _mm_floor_ps(__m128 s1);
__m128 _mm_ceil_ps(__m128 s1)
__m256 _mm256_round_ps(__m256 s1, int iRoundMode);
__m256 _mm256_floor_ps(__m256 s1);
__m256 _mm256_ceil_ps(__m256 s1)

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN).
Precision (signaled only if imm[3] = ’0’; if imm[3] = ’1’, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by ROUNDPS.

Other Exceptions
See Table 2-19, “Type 2 Class Exception Conditions,” additionally:
#UD If VEX.vvvv ≠ 1111B.
**ROUNDSD—Round Scalar Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 3A 0B /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>SSE4_1</td>
<td>Round the low packed double precision floating-point value in xmm2/m64 and place the result in xmm1. The rounding mode is determined by imm8.</td>
</tr>
<tr>
<td>VEX:LIG.66.0F3A.WIG 0B /r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Round the low packed double precision floating-point value in xmm3/m64 and place the result in xmm1. The rounding mode is determined by imm8. Upper packed double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Round the double precision floating-point value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation**

```c
IF (imm[2] = ‘1’)
    THEN // rounding mode is determined by MXCSR.RC
        DEST[63:0] := ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[63:0] := ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged;
```

**ROUNDSD (128-bit Legacy SSE Version)**

DEST[63:0] := RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[MAXVL-1:64] (Unmodified)
VROUNDSD (VEX.128 Encoded Version)
DEST[63:0] := RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
ROUNDSD __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
ROUNDSD __m128d mm_floor_sd(__m128d dst, __m128d s1);
ROUNDSD __m128d mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN).
Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by ROUNDSD.

Other Exceptions
See Table 2-20, "Type 3 Class Exception Conditions."
ROUNDSS—Round Scalar Single Precision Floating-Point Values

**Description**
Round the single precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single precision floating-point input to an integer value and returns the result as a single precision floating-point value in the lowest position. The upper three single precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation**
IF (imm[2] = '1')
   THEN // rounding mode is determined by MXCSR.RC
       DEST[31:0] := ConvertSPFPToInteger_M(SRC[31:0]);
   ELSE // rounding mode is determined by IMM8.RC
       DEST[31:0] := ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged;

**ROUNDSS (128-bit Legacy SSE Version)**
DEST[31:0] := RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[MAXVL-1:32] (Unmodified)
VROUNDSS (VEX.128 Encoded Version)
DEST[31:0] := RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
ROUNDSS __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
ROUNDSS __m128 mm_floor_ss(__m128 dst, __m128 s1);
ROUNDSS __m128 mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions
Invalid (signaled only if SRC = SNaN).
Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)
Note that Denormal is not signaled by ROUNDSS.

Other Exceptions
See Table 2-20, “Type 3 Class Exception Conditions.”
RSM—Resume From System Management Mode

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AA</td>
<td>RSM</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Resume operation of interrupted program.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor’s state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode. See Chapter 32, “System Management Mode,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about SMM and the behavior of the RSM instruction.

**Operation**

```
ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_OCH )
    THEN
        ProcessorState := Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState := Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
    FI
```

**Flags Affected**

All.

**Protected Mode Exceptions**

#UD If an attempt is made to execute this instruction when the processor is not in SMM.

If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions

Same exceptions as in protected mode.
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values

### Opcode Table

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 52 /r RSQRTPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocals of the square roots of the four packed single precision floating-point values in xmm2/m128 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 52 /r VRSQRTPS xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of the square roots of packed single precision values in xmm2/mem and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 52 /r VRSQRTPS ymm1, ymm2/m256</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocals of the square roots of packed single precision values in ymm2/mem and stores the results in ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single precision floating-point values in the source operand (second operand) and stores the packed single precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a SIMD single precision floating-point operation.

The relative error for this approximation is:

$$\text{Relative Error} \leq 1.5 \times 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than −0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Operation

RSQRTPS (128-bit Legacy SSE Version)
DEST[31:0] := APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] := APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] := APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] := APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] (Unmodified)

VRSQRTPS (VEX.128 Encoded Version)
DEST[31:0] := APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] := APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] := APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] := APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] := 0

VRSQRTPS (VEX.256 Encoded Version)
DEST[31:0] := APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] := APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] := APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] := APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] := APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[255:224] := APPROXIMATE(1/SQRT(SRC2[255:224]))

Intel C/C++ Compiler Intrinsic Equivalent
RSQRTPS __m128 _mm_rsqrt_ps(__m128 a)
RSQRTPS __m256 _mm256_rsqrt_ps (__m256 a);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions," additionally:
#UD If VEX.vvvv ≠ 1111B.
RSQRTSS—Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value

### Opcode\(^*/\) Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 52 /r RSQRTSS xmm1, xmm2/m32</td>
<td>RM V/V</td>
<td>SSE</td>
<td>Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 52 /r VRSQRTSS xmm1, xmm2, xmm3/m32</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Computes the approximate reciprocal of the square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes an approximate reciprocal of the square root of the low single precision floating-point value in the source operand (second operand) stores the single precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of a scalar single precision floating-point operation.

The relative error for this approximation is:

\[|\text{Relative Error}| \leq 1.5 \times 2^{-12}\]

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \(\infty\) of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than \(-0.0\)), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

**RSQRTSS (128-bit Legacy SSE Version)**

\[
\text{DEST}[31:0] := \text{APPROXIMATE}(1/\sqrt{\text{SRC2}[31:0]})
\]

\[
\text{DEST}[\text{MAXVL}-1:32] \text{ (Unmodified)}
\]

**VRSQRTSS (VEX.128 Encoded Version)**

\[
\text{DEST}[31:0] := \text{APPROXIMATE}(1/\sqrt{\text{SRC2}[31:0]})
\]

\[
\text{DEST}[127:32] := \text{SRC1}[127:32]
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
Intel C/C++ Compiler Intrinsic Equivalent
RSQRTPS __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-22, “Type 5 Class Exception Conditions.”
**RSTORSSP—Restore Saved Shadow Stack Pointer**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 01 /5 (mod!=11, /5, memory only) RSTORSSP m64</td>
<td>M</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Restore SSP.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Restores SSP from the shadow-stack-restore token pointed to by m64. If the SSP restore was successful then the instruction replaces the shadow-stack-restore token with a previous-ssp token. The instruction sets the CF flag to indicate whether the SSP address recorded in the shadow-stack-restore token that was processed was 4 byte aligned, i.e., whether an alignment hole was created when the restore-shadow-stack token was pushed on this shadow stack.

Following RSTORSSP if a restore-shadow-stack token needs to be saved on the previous shadow stack, use the SAVEPREVSSP instruction.

If pushing a restore-shadow-stack token on the previous shadow stack is not required, the previous-ssp token can be popped using the INCSSPQ instruction. If the CF flag was set to indicate presence of an alignment hole, an additional INCSSPD instruction is needed to advance the SSP past the alignment hole.

**Operation**

IF CPL = 3
  IF (CR4.CET & IA32_U.CET.SH_STK_EN) = 0
    THEN #UD; Fl; |
  ELSE
    IF (CR4.CET & IA32_S.CET.SH_STK_EN) = 0
      THEN #UD; Fl; |
  FI;
FI;

SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
  THEN #GP(0); Fl;

previous_ssp_token = SSP | (IA32_EFER.LMA AND CS.L) | 0x02
Start Atomic Execution
restore_ssp_token = Locked shadow_stack_load 8 bytes from SSP_LA
fault = 0

IF ((restore_ssp_token & 0x03) != (IA32_EFER.LMA & CS.L))
  THEN fault = 1; Fl; (* If L flag in token does not match IA32_EFER.LMA & CS.L or bit 1 is not 0 *)

IF ((IA32_EFER.LMA AND CS.L) = 0 AND restore_ssp_token[63:32] != 0)
  THEN fault = 1; Fl; (* If compatibility/legacy mode and SSP to be restored not below 4G *)

TMP = restore_ssp_token & ~0x01
TMP = (TMP - 8)
TMP = TMP & ~0x07
IF TMP != SSP_LA
THEN fault = 1; Fl; (* If address in token does not match the requested top of stack *)

TMP = (fault == 0) ? previous_ssp_token : restore_ssp_token
shadow_stack_store 8 bytes of TMP to SSP_LA and release lock
End Atomic Execution

IF fault == 1
    THEN #CP(RSTORSSP); Fl;

SSP = SSP_LA

// Set the CF if the SSP in the restore token was 4 byte aligned, i.e., there is an alignment hole
RFLAGS.CF = (restore_ssp_token & 0x04) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF := 0;

Flags Affected
CF is set to indicate if the shadow stack pointer in the restore token was 4 byte aligned, else it is cleared. ZF, PF, AF, OF, and SF are cleared.

C/C++ Compiler Intrinsic Equivalent
RSTORSSP void _rstorssp(void *);

Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0) If linear address of memory operand not 8 byte aligned.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If destination is located in a non-writeable segment.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#CP(rstorssp) If L bit in token does not match (IA32_EFER.LMA & CS.L).
   If address in token does not match linear address of memory operand.
   If in 32-bit or compatibility mode and the address in token is not below 4G.
#PF(fault-code) If a page fault occurs.

Real-Address Mode Exceptions
#UD The RSTORSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The RSTORSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same as protected mode exceptions.
64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.

#GP(0) If linear address of memory operand not 8 byte aligned.
   If a memory address is in a non-canonical form.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#CP(rstorssp) If L bit in token does not match (IA32_EFER.LMA & CS.L).
   If address in token does not match linear address of memory operand.

#PF(fault-code) If a page fault occurs.
SAHF—Store AH Into Flags

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9E</td>
<td>SAHF</td>
<td>ZO</td>
<td>Invalid*</td>
<td>Valid</td>
<td>Loads SF, ZF, AF, PF, and CF from AH into the EFLAGS register.</td>
</tr>
</tbody>
</table>

NOTES:
1. Valid in specific steppings. See Description section.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation
IF IA-64 Mode
THEN
  IF CPUID.80000001H:ECX[0] = 1;
  THEN
    RFLAGS(SF:ZF:0:AF:0:PF:1:CF) := AH;
  ELSE
    #UD;
  FI
ELSE
  EFLAGS(SF:ZF:0:AF:0:PF:1:CF) := AH;
FI;

Flags Affected
The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

Protected Mode Exceptions
None.

Real-Address Mode Exceptions
None.

Virtual-8086 Mode Exceptions
None.

Compatibility Mode Exceptions
None.
64-Bit Mode Exceptions

#UD
If CPUID.80000001H.ECX[0] = 0.
If the LOCK prefix is used.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 /4</td>
<td>SAL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, once.</td>
</tr>
<tr>
<td>REX + D0 /4</td>
<td>SAL r/m8, 2</td>
<td>M1</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D2 /4</td>
<td>SAL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, CL times.</td>
</tr>
<tr>
<td>REX + D2 /4</td>
<td>SAL r/m8, CL</td>
<td>MC</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C0 /4 ib</td>
<td>SAL r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
</tr>
<tr>
<td>REX + C0 /4 ib</td>
<td>SAL r/m8, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D1 /4</td>
<td>SAL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D3 /4</td>
<td>SAL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SAL r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>REX.W + C1 /4 ib</td>
<td>SAL r/m64, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Multiply r/m64 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D0 /7</td>
<td>SAR r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Signed divide⁵ r/m8 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>REX + D0 /7</td>
<td>SAR r/m8, 1</td>
<td>M1</td>
<td>N.E.</td>
<td>Signed divide⁵ r/m8 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D2 /7</td>
<td>SAR r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Signed divide⁵ r/m8 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>REX + D2 /7</td>
<td>SAR r/m8, CL</td>
<td>MC</td>
<td>N.E.</td>
<td>Signed divide⁵ r/m8 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C0 /7 ib</td>
<td>SAR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Signed divide⁵ r/m8 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>REX + C0 /7 ib</td>
<td>SAR r/m8, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Signed divide⁵ r/m8 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D1 /7</td>
<td>SAR r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Signed divide⁵ r/m16 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Signed divide⁵ r/m16 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C1 /7 ib</td>
<td>SAR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Signed divide⁵ r/m16 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D1 /7</td>
<td>SAR r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Signed divide⁵ r/m32 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>REX.W + D1 /7</td>
<td>SAR r/m64, 1</td>
<td>M1</td>
<td>N.E.</td>
<td>Signed divide⁵ r/m64 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D3 /7</td>
<td>SAR r/m32, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Signed divide⁵ r/m32 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>REX.W + D3 /7</td>
<td>SAR r/m64, CL</td>
<td>MC</td>
<td>N.E.</td>
<td>Signed divide⁵ r/m64 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C1 /7 ib</td>
<td>SAR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Signed divide⁵ r/m32 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>REX.W + C1 /7 ib</td>
<td>SAR r/m64, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Signed divide⁵ r/m64 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D0 /4</td>
<td>SHL r/m8, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>REX + D0 /4</td>
<td>SHL r/m8, 2</td>
<td>M1</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D2 /4</td>
<td>SHL r/m8, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>REX + D2 /4</td>
<td>SHL r/m8, CL</td>
<td>MC</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C0 /4 ib</td>
<td>SHL r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>REX + C0 /4 ib</td>
<td>SHL r/m8, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>Multiply r/m8 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m16, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, once.</td>
<td></td>
</tr>
<tr>
<td>D3 /4</td>
<td>SHL r/m16, CL</td>
<td>MC</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, CL times.</td>
<td></td>
</tr>
<tr>
<td>C1 /4 ib</td>
<td>SHL r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Multiply r/m16 by 2, imm8 times.</td>
<td></td>
</tr>
<tr>
<td>D1 /4</td>
<td>SHL r/m32, 1</td>
<td>M1</td>
<td>Valid</td>
<td>Multiply r/m32 by 2, once.</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>ModRM:r/m (r, w)</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MC</td>
<td>ModRM:r/m (r, w)</td>
<td>CL</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r, w)</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits with a 64-bit operand). The count range is limited to 0 to 31 (or 63 with a 64-bit operand). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1).

NOTES:

1. See the IA-32 Architecture Compatibility section below.
2. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
3. Not the same form of division as IDIV; rounding is toward negative infinity.
The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position’s shifted value with the sign of the unshifted value (see Figure 7-9 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the “quotient” of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the “remainder” is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction’s default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

**IA-32 Architecture Compatibility**

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

**Operation**

```plaintext
IF OperandSize = 64
   THEN
   countMASK := 3FH;
   ELSE
   countMASK := 1FH;
   FI

tempCOUNT := (COUNT AND countMASK);
tempDEST := DEST;
WHILE (tempCOUNT ≠ 0)
   DO
   IF instruction is SAL or SHL
      THEN
         CF := MSB(DEST);
      ELSE (* Instruction is SAR or SHR *)
         CF := LSB(DEST);
      FI;
   IF instruction is SAL or SHL
      THEN
         DEST := DEST * 2;
      ELSE
         IF instruction is SAR
```
THEN
    DEST := DEST / 2; (* Signed divide, rounding toward negative infinity *)
ELSE (* Instruction is SHR *)
    DEST := DEST / 2 ; (* Unsigned divide *)
FI;
FI;
tempCOUNT := tempCOUNT – 1;
OD;
(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF := MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF := 0;
                    ELSE (* Instruction is SHR *)
                        OF := MSB(tempDEST);
                    FI;
            FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF := undefined;
        FI;
    FI;
Flags Affected
The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
    current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.
Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
SARX/SHLX/SHRX—Shift Without Affecting Flags

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LZ.F3.0F38.W0 F7 /r SARX r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Shift r/m32 arithmetically right with count specified in r32b.</td>
</tr>
<tr>
<td>VEX.LZ.66.0F38.W0 F7 /r SHLX r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Shift r/m32 logically left with count specified in r32b.</td>
</tr>
<tr>
<td>VEX.LZ.F2.0F38.W0 F7 /r SHRX r32a, r/m32, r32b</td>
<td>RMV</td>
<td>V/V</td>
<td>BMI2</td>
<td>Shift r/m32 logically right with count specified in r32b.</td>
</tr>
<tr>
<td>VEX.LZ.F3.0F38.W1 F7 /r SARX r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Shift r/m64 arithmetically right with count specified in r64b.</td>
</tr>
<tr>
<td>VEX.LZ.66.0F38.W1 F7 /r SHLX r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Shift r/m64 logically left with count specified in r64b.</td>
</tr>
<tr>
<td>VEX.LZ.F2.0F38.W1 F7 /r SHRX r64a, r/m64, r64b</td>
<td>RMV</td>
<td>V/N.E.</td>
<td>BMI2</td>
<td>Shift r/m64 logically right with count specified in r64b.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize -1, the COUNT value is masked.

SARX, SHRX, and SHLX instructions do not update flags.

Operation

```
TEMP := SRC1;
IF VEX.W1 and CS.L = 1 THEN
countMASK := 3FH;
ELSE
countMASK := 1FH;
FI
COUNT := (SRC2 AND countMASK)
DEST[OperandSize -1] = TEMP[OperandSize -1];
DO WHILE (COUNT ≠ 0)
   IF instruction is SHLX THEN
```

SARX/SHLX/SHRX—Shift Without Affecting Flags
DEST[] := DEST * 2;
ELSE IF instruction is SHRX
THEN
     DEST[] := DEST / 2; // unsigned divide
ELSE      // SARX
     DEST[] := DEST / 2; // signed divide, round toward negative infinity
FI;
COUNT := COUNT - 1;
OD

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
Auto-generated from high-level language.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-29, “Type 13 Class Exception Conditions.”
SAVEPREVSSP—Save Previous Shadow Stack Pointer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 01 EA (mod!=11, /S, RM=010) SAVEPREVSSP</td>
<td>ZO</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Save a restore-shadow-stack token on previous shadow stack.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>ZO</td>
</tr>
</tbody>
</table>

**Description**
Push a restore-shadow-stack token on the previous shadow stack at the next 8 byte aligned boundary. The previous SSP is obtained from the previous-ssp token at the top of the current shadow stack.

**Operation**
IF CPL = 3
  IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
    THEN #UD; Fl;
  ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
      THEN #UD; Fl;
  FI;
FI;

IF SSP not aligned to 8 bytes
  THEN #GP(0); Fl;

(* Pop the "previous-ssp" token from current shadow stack *)
previous_ssp_token = ShadowStackPop8B(SSP)

(* If the CF flag indicates there was a alignment hole on current shadow stack then pop that alignment hole *)
(* Note that the alignment hole must be zero and can be present only when in legacy/compatibility mode *)
IF RFLAGS.CF == 1 AND (IA32_EFER.LMA AND CS.L)
  #GP(0)
FI;

IF RFLAGS.CF == 1
  must_be_zero = ShadowStackPop4B(SSP)
  IF must_be_zero != 0 THEN #GP(0)
FI;

(* Previous SSP token must have the bit 1 set *)
IF ((previous_ssp_token & 0x02) == 0)
  THEN #GP(0); (* bit 1 was 0 *)

IF ((IA32_EFER.LMA AND CS.L) = 0 AND previous_ssp_token [63:32] != 0)
  THEN #GP(0); Fl; (* If compatibility/legacy mode and SSP not in 4G *)

(* Save Prev SSP from previous_ssp_token to the old shadow stack at next 8 byte aligned address *)
old_SSP = previous_ssp_token & ~0x07
temp := (old_SSP | (IA32_EFER.LMA & CS.L));
Shadow_stack_store 4 bytes of 0 to (old_SSP - 4)
old_SSP := old_SSP & ~0x07;
Shadow_stack_store 8 bytes of temp to (old_SSP - 8)

Flags Affected
None.

C/C++ Compiler Intrinsic Equivalent
SAVEPREVSSP void _saveprevssp(void);

Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0) If SSP not 8 byte aligned.
   If alignment hole on shadow stack is not 0.
   If bit 1 of the previous-ssp token is not set to 1.
#PF(fault-code) If in 32-bit/compatibility mode and SSP recorded in previous-ssp token is beyond 4G.

Real-Address Mode Exceptions
#UD The SAVEPREVSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The SAVEPREVSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same as protected mode exceptions.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0) If SSP not 8 byte aligned.
   If carry flag is set.
   If bit 1 of the previous-ssp token is not set to 1.
#PF(fault-code) If a page fault occurs.
### SBB—Integer Subtraction With Borrow

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1C ib</td>
<td>SBB AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm8 from AL.</td>
</tr>
<tr>
<td>1D iw</td>
<td>SBB AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm16 from AX.</td>
</tr>
<tr>
<td>1D id</td>
<td>SBB EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm32 from EAX.</td>
</tr>
<tr>
<td>REX.W + 1D id</td>
<td>SBB RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow sign-extended imm.32 to 64-bits from RAX.</td>
</tr>
<tr>
<td>80 /3 ib</td>
<td>SBB r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm8 from r/m8.</td>
</tr>
<tr>
<td>REX + 80 /3 ib</td>
<td>SBB r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow imm8 from r/m8.</td>
</tr>
<tr>
<td>81 /3 iw</td>
<td>SBB r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm16 from r/m16.</td>
</tr>
<tr>
<td>81 /3 id</td>
<td>SBB r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow imm32 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /3 id</td>
<td>SBB r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow sign-extended imm32 to 64-bits from r/m64.</td>
</tr>
<tr>
<td>83 /3 ib</td>
<td>SBB r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow sign-extended imm8 from r/m16.</td>
</tr>
<tr>
<td>83 /3 ib</td>
<td>SBB r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow sign-extended imm8 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 83 /3 ib</td>
<td>SBB r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow sign-extended imm8 from r/m64.</td>
</tr>
<tr>
<td>18 /r</td>
<td>SBB r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r8 from r/m8.</td>
</tr>
<tr>
<td>REX + 18 /r</td>
<td>SBB r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r8 from r/m8.</td>
</tr>
<tr>
<td>19 /r</td>
<td>SBB r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r16 from r/m16.</td>
</tr>
<tr>
<td>19 /r</td>
<td>SBB r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r32 from r/m32.</td>
</tr>
<tr>
<td>REX.W + 19 /r</td>
<td>SBB r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r64 from r/m64.</td>
</tr>
<tr>
<td>1A /r</td>
<td>SBB r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r/m8 from r8.</td>
</tr>
<tr>
<td>REX + 1A /r</td>
<td>SBB r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r/m8 from r8.</td>
</tr>
<tr>
<td>1B /r</td>
<td>SBB r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r/m16 from r16.</td>
</tr>
<tr>
<td>1B /r</td>
<td>SBB r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Subtract with borrow r/m32 from r32.</td>
</tr>
<tr>
<td>REX.W + 1B /r</td>
<td>SBB r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Subtract with borrow r/m64 from r64.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (w)</td>
<td>ModRMr/reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMr/ (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location.
(However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST := (DEST – (SRC + CF));

**Intel C/C++ Compiler Intrinsic Equivalent**

SBB extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);
SBB extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);
SBB extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *diff_out);
SBB extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);

**Flags Affected**

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

**Protected Mode Exceptions**

<table>
<thead>
<tr>
<th>Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP</td>
<td>If the destination is located in a non-writable segment.</td>
</tr>
<tr>
<td></td>
<td>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</td>
</tr>
<tr>
<td>#SS</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
<tr>
<td>#PF</td>
<td>If a page fault occurs.</td>
</tr>
<tr>
<td>#AC</td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>#UD</td>
<td>If the LOCK prefix is used but the destination is not a memory operand.</td>
</tr>
</tbody>
</table>

**Real-Address Mode Exceptions**

<table>
<thead>
<tr>
<th>Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP</td>
<td>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</td>
</tr>
<tr>
<td>#SS</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
<tr>
<td>#UD</td>
<td>If the LOCK prefix is used but the destination is not a memory operand.</td>
</tr>
</tbody>
</table>

**Virtual-8086 Mode Exceptions**

<table>
<thead>
<tr>
<th>Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP</td>
<td>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</td>
</tr>
<tr>
<td>#SS</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
<tr>
<td>#PF</td>
<td>If a page fault occurs.</td>
</tr>
<tr>
<td>#AC</td>
<td>If alignment checking is enabled and an unaligned memory reference is made.</td>
</tr>
<tr>
<td>#UD</td>
<td>If the LOCK prefix is used but the destination is not a memory operand.</td>
</tr>
</tbody>
</table>
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
SCAS/SCASB/SCASW/SCASD—Scan String

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>SCAS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare AL with byte at ES:(E)DI or RDI, then set status flags.¹</td>
</tr>
<tr>
<td>AF</td>
<td>SCAS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare AX with word at ES:(E)DI or RDI, then set status flags.¹</td>
</tr>
<tr>
<td>AF</td>
<td>SCAS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.¹</td>
</tr>
<tr>
<td>REX.W + AF</td>
<td>SCAS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare RAX with quadword at RDI or EDI then set status flags.</td>
</tr>
<tr>
<td>AE</td>
<td>SCASB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare AL with byte at ES:(E)DI or RDI then set status flags.¹</td>
</tr>
<tr>
<td>AF</td>
<td>SCASW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare AX with word at ES:(E)DI or RDI then set status flags.¹</td>
</tr>
<tr>
<td>AF</td>
<td>SCASD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.¹</td>
</tr>
<tr>
<td>REX.W + AF</td>
<td>SCASQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Compare RAX with quadword at RDI or EDI then set status flags.</td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.
SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of status flags. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

**Non-64-bit Mode:**

IF (Byte comparison)

```
THEN
  temp := AL – SRC;
  SetStatusFlags(temp);
  IF DF = 0
    THEN (E)DI := (E)DI + 1;
    ELSE (E)DI := (E)DI - 1;
  ELSE IF (Word comparison)
    THEN
      temp := AX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (E)DI := (E)DI + 2;
        ELSE (E)DI := (E)DI - 2;
    FI;
  ELSE IF (Doubleword comparison)
    THEN
      temp := EAX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (E)DI := (E)DI + 4;
        ELSE (E)DI := (E)DI - 4;
    FI;
  FI;
FI;
```

**64-bit Mode:**

IF (Byte comparison)

```
THEN
  temp := AL – SRC;
  SetStatusFlags(temp);
  IF DF = 0
    THEN (R|E)DI := (R|E)DI + 1;
    ELSE (R|E)DI := (R|E)DI - 1;
  ELSE IF (Word comparison)
    THEN
      temp := AX – SRC;
      SetStatusFlags(temp);
      IF DF = 0
        THEN (R|E)DI := (R|E)DI + 2;
        ELSE (R|E)DI := (R|E)DI - 2;
    FI;
  FI;
```
ELSE IF (Doubleword comparison)
    THEN
        temp := EAX - SRC;
        SetStatusFlags(temp);
        IF DF = 0
            THEN (R|E)DI := (R|E)DI + 4;
            ELSE (R|E)DI := (R|E)DI - 4; FI;
        FI;
ELSE IF (Quadword comparison using REX.W)
    THEN
        temp := RAX - SRC;
        SetStatusFlags(temp);
        IF DF = 0
            THEN (R|E)DI := (R|E)DI + 8;
            ELSE (R|E)DI := (R|E)DI - 8; FI;
        FI;
FI;

Flags Affected
The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the limit of the ES segment.
    If the ES register contains a NULL segment selector.
    If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
    current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
SENDUIPI—Send User Interprocessor Interrupt

Description

The SENDUIPI instruction sends the user interprocessor interrupt (IPI) indicated by its register operand. (The operand always has 64 bits; operand-size overrides such as the prefix 66 are ignored.)

SENDUIPI uses a data structure called the user-interrupt target table (UITT). This table is located at the linear address UIITTADDR (in the IA32_UINTR_TT MSR); it comprises UITTSZ+1 16-byte entries, where UITTSZ = IA32_UINT_MISC[31:0]. SENDUIPI uses the UITT entry (UITTE) indexed by the instruction's register operand. Each UITTE has the following format:

- Bit 0: V, a valid bit.
- Bits 7:1 are reserved and must be 0.
- Bits 15:8: UV, the user-interrupt vector (in the range 0–63, so bits 15:14 must be 0).
- Bits 63:16 are reserved.
- Bits 127:64: UPIDADDR, the linear address of a user posted-interrupt descriptor (UPID). (UPIDADDR is 64-byte aligned, so bits 69:64 of each UITTE must be 0.)

Each UPID has the following format (fields and bits not referenced are reserved):

- Bit 0 (ON) indicates an outstanding notification. If this bit is set, there is a notification outstanding for one or more user interrupts in PIR.
- Bit 1 (SN) indicates that notifications should be suppressed. If this bit is set, agents (including SENDUIPI) should not send notifications when posting user interrupts in this descriptor.
- Bits 23:16 (NV) contain the notification vector. This is used by agents sending user-interrupt notifications (including SENDUIPI).
- Bits 63:32 (NDST) contain the notification destination. This is the target physical APIC ID (in xAPIC mode, bits 47:40 are the 8-bit APIC ID; in x2APIC mode, the entire field forms the 32-bit APIC ID).
- Bits 127:64 (PIF) contain posted-interrupt requests. There is one bit for each user-interrupt vector. There is a user-interrupt request for a vector if the corresponding bit is 1.

Although SENDUIPI may be executed at any privilege level, all of the instruction’s memory accesses (to a UITTE and a UPID) are performed with supervisor privilege.

SENDUIPI sends a user interrupt by posting a user interrupt with vector V in the UPID referenced by UPIDADDR and then sending, as an ordinary IPI, any notification interrupt specified in that UPID.

Operation

IF reg > UITTSZ;
    THEN #GP(0);
Fi;
read tempUITTE from 16 bytes at UIITTADDR+ (reg « 4);
IF tempUITTE.V = 0 or tempUITTE sets any reserved bit
    THEN #GP(0);
Fi;

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM.reg(r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
read tempUPID from 16 bytes at tempUITTE.UPIDADDR; // under lock
IF tempUPID sets any reserved bits or bits that must be zero
  THEN #GP(0); // release lock
FI;
tempUPID.PIR[tempUITTE.UV] := 1;
IF tempUPID.SN = tempUPID.ON = 0
  THEN
    tempUPID.ON := 1;
    sendNotify := 1;
  ELSE sendNotify := 0;
  FI;
write tempUPID to 16 bytes at tempUITTE.UPIDADDR; // release lock
IF sendNotify = 1
  THEN
    IF local APIC is in x2APIC mode
      THEN send ordinary IPI with vector tempUPID.NV
           to 32-bit physical APIC ID tempUPID.NDST;
      ELSE send ordinary IPI with vector tempUPID.NV
           to 8-bit physical APIC ID tempUPID.NDST[15:8];
    FI;
  FI;

Flags Affected
None.

Protected Mode Exceptions
#UD The SENDUIPI instruction is not recognized in protected mode.

Real-Address Mode Exceptions
#UD The SENDUIPI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The SENDUIPI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The SENDUIPI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
  If executed inside an enclave.
  If CR4.UINTR = 0.
  If IA32_UINTR_TT[0] = 0.
  If CPUID.07H.OH:EDX.UINTR[bit 5] = 0.
#PF If a page fault occurs.
#GP If the value of the register operand exceeds UITTSZ.
  If the selected UITTE is not valid or sets any reserved bits.
  If the selected UPID sets any reserved bits.
  If there is an attempt to access memory using a linear address that is not canonical relative to
  the current paging mode.
SERIALIZE—Serialize Instruction Execution

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 E8 SERIALIZE</td>
<td>ZS</td>
<td>V/V</td>
<td>SERIALIZE</td>
<td>Serialize instruction fetch and execution.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZS</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Serializes instruction execution. Before the next instruction is fetched and executed, the SERIALIZE instruction ensures that all modifications to flags, registers, and memory by previous instructions are completed, draining all buffered writes to memory. This instruction is also a serializing instruction as defined in the section “Serializing Instructions” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

SERIALIZE does not modify registers, arithmetic flags, or memory.

### Operation

Wait_On_Fetch_And_Execution_Of_Next_Instruction_Until(preceding_instructions_complete_and_preceding_stores_globally_visible);

### Intel C/C++ Compiler Intrinsic Equivalent

SERIALIZE void _serialize(void);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

#UD If the LOCK prefix is used.
If CPUID.07H.0H:EDX.SERIALIZE[bit 14] = 0.
## SETcc—Set Byte on Condition

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 97</td>
<td>SETA r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if above (CF=0) and (ZF=0).</td>
</tr>
<tr>
<td>REX + 0F 97</td>
<td>SETA r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if above (CF=0) and (ZF=0).</td>
</tr>
<tr>
<td>0F 93</td>
<td>SETAE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if above or equal (CF=0).</td>
</tr>
<tr>
<td>REX + 0F 93</td>
<td>SETAE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if above or equal (CF=0).</td>
</tr>
<tr>
<td>0F 92</td>
<td>SETB r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if below (CF=1).</td>
</tr>
<tr>
<td>REX + 0F 92</td>
<td>SETB r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if below (CF=1).</td>
</tr>
<tr>
<td>0F 96</td>
<td>SETBE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if below or equal (CF=1) or (ZF=1).</td>
</tr>
<tr>
<td>REX + 0F 96</td>
<td>SETBE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if below or equal (CF=1) or (ZF=1).</td>
</tr>
<tr>
<td>0F 92</td>
<td>SETC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if carry (CF=1).</td>
</tr>
<tr>
<td>REX + 0F 92</td>
<td>SETC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if carry (CF=1).</td>
</tr>
<tr>
<td>0F 94</td>
<td>SETE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if equal (ZF=1).</td>
</tr>
<tr>
<td>REX + 0F 94</td>
<td>SETE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if equal (ZF=1).</td>
</tr>
<tr>
<td>0F 9F</td>
<td>SETG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if greater (ZF=0) and (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9F</td>
<td>SETG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if greater (ZF=0) and (SF=OF).</td>
</tr>
<tr>
<td>0F 9D</td>
<td>SETGE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9D</td>
<td>SETGE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 9C</td>
<td>SETL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if less (SF\neq OF).</td>
</tr>
<tr>
<td>REX + 0F 9C</td>
<td>SETL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if less (SF\neq OF).</td>
</tr>
<tr>
<td>0F 9E</td>
<td>SETLE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if less or equal (ZF=1) or (SF\neq OF).</td>
</tr>
<tr>
<td>REX + 0F 9E</td>
<td>SETLE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if less or equal (ZF=1) or (SF\neq OF).</td>
</tr>
<tr>
<td>0F 96</td>
<td>SETNA r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not above (CF=1) or (ZF=1).</td>
</tr>
<tr>
<td>REX + 0F 96</td>
<td>SETNA r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not above (CF=1) or (ZF=1).</td>
</tr>
<tr>
<td>0F 92</td>
<td>SETNAE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not above or equal (CF=1).</td>
</tr>
<tr>
<td>REX + 0F 92</td>
<td>SETNAE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not above or equal (CF=1).</td>
</tr>
<tr>
<td>0F 93</td>
<td>SETNB r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not below (CF=0).</td>
</tr>
<tr>
<td>REX + 0F 93</td>
<td>SETNB r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not below (CF=0).</td>
</tr>
<tr>
<td>0F 97</td>
<td>SETNBE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not below or equal (CF=0) or (ZF=0).</td>
</tr>
<tr>
<td>REX + 0F 97</td>
<td>SETNBE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not below or equal (CF=0) or (ZF=0).</td>
</tr>
<tr>
<td>0F 93</td>
<td>SETNC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not carry (CF=0).</td>
</tr>
<tr>
<td>REX + 0F 93</td>
<td>SETNC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not carry (CF=0).</td>
</tr>
<tr>
<td>0F 95</td>
<td>SETNE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not equal (ZF=0).</td>
</tr>
<tr>
<td>REX + 0F 95</td>
<td>SETNE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not equal (ZF=0).</td>
</tr>
<tr>
<td>0F 9E</td>
<td>SETNG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not greater (ZF=1) or (SF\neq OF)</td>
</tr>
<tr>
<td>REX + 0F 9E</td>
<td>SETNG r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not greater (ZF=1) or (SF\neq OF)</td>
</tr>
<tr>
<td>0F 9C</td>
<td>SETNGE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not greater or equal (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9C</td>
<td>SETNGE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not greater or equal (SF=OF).</td>
</tr>
<tr>
<td>0F 9D</td>
<td>SETNL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not less (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9D</td>
<td>SETNL r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not less (SF=OF).</td>
</tr>
<tr>
<td>0F 9F</td>
<td>SETNLE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set byte if not less or equal (ZF=0) and (SF=OF).</td>
</tr>
<tr>
<td>REX + 0F 9F</td>
<td>SETNLE r/m8</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set byte if not less or equal (ZF=0) and (SF=OF).</td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (cc) indicates the condition being tested for.

The terms “above” and “below” are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SETcc instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, “EFLAGS Condition Codes,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SETcc instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.
The reg field of the ModR/M byte is not used for the SETCC instruction and those opcode bits are ignored by the processor.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction’s operation is the same as in legacy mode and compatibility mode.

**Operation**

\[
\text{IF condition} \\
\quad \text{THEN } \text{DEST} := 1; \\
\quad \text{ELSE } \text{DEST} := 0; \\
\text{FI;}
\]

**Flags Affected**

None.

**Protected Mode Exceptions**

- #GP(0) If the destination is located in a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #UD If the LOCK prefix is used.
SETSSBSY—Mark Shadow Stack Busy

**Description**

The SETSSBSY instruction verifies the presence of a non-busy supervisor shadow stack token at the address in the IA32_PL0_SSP MSR and marks it busy. Following successful execution of the instruction, the SSP is set to the value of the IA32_PL0_SSP MSR.

**Operation**

1. IF (CR4.CET = 0)  
   THEN #UD; Fi;
2. IF (IA32_S_CET.SH_STK_EN = 0)  
   THEN #UD; Fi;
3. IF CPL > 0  
   THEN GP(0); Fi;
4. SSP_LA = IA32_PL0_SSP  
   If SSP_LA not aligned to 8 bytes  
   THEN #GP(0); Fi;
5. expected_token_value = SSP_LA  
   (* busy bit must not be set *)
6. new_token_value = SSP_LA | BUSY_BIT  
   (* set busy bit; bit position 0 *)
7. IF shadow_stack_lock_cmpxchg8B(SSP_LA, new_token_value, expected_token_value) != expected_token_value  
   THEN #CP(SETSSBSY); Fi;
8. SSP = SSP_LA

**Flags Affected**

None.

**C/C++ Compiler Intrinsic Equivalent**

SETSSBSY

void _setssbsy(void);

**Protected Mode Exceptions**

- #UD  
  If the LOCK prefix is used.
  If CR4.CET = 0.
  If IA32_S_CET.SH_STK_EN = 0.
- #GP(0)  
  If IA32_PL0_SSP not aligned to 8 bytes.
  If CPL is not 0.
- #CP(setssbsy)  
  If busy bit in token is set.
  If in 32-bit or compatibility mode, and the address in token is not below 4G.
#PF(fault-code)  If a page fault occurs.

**Real-Address Mode Exceptions**
#UD  The SETSSBSY instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**
#UD  The SETSSBSY instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**
Same as protected mode exceptions.

**64-Bit Mode Exceptions**
Same as protected mode exceptions.
**SFENCE—Store Fence**

**Opcode**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE F8</td>
<td>SFENCE</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Serializes store operations.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Orders processor execution relative to all memory stores prior to the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible. The SFENCE instruction is ordered with respect to memory stores, other SFENCE instructions, MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to memory loads or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

**Operation**

Wait_On_Following_Stores_Until(preceding_stores_globally_visible);

**Intel C/C++ Compiler Intrinsic Equivalent**

```c
void _mm_sfence(void)
```

**Exceptions (All Operating Modes)**

#UD If CPUID.01H:EDX.SSE[bit 25] = 0.

If the LOCK prefix is used.
SGDT—Store Global Descriptor Table Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /0</td>
<td>SGDT m</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Store GDTR to m.</td>
</tr>
</tbody>
</table>

NOTES:
1. See the IA-32 Architecture Compatibility section below.

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

## Description
Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 or 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for information on loading the GDTR and IDTR.

## IA-32 Architecture Compatibility
The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

## Operation
IF instruction is SGDT
  IF OperandSize = 16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
    THEN
      DEST[0:15] := GDTR(Limit);
      DEST[16:47] := GDTR(Base); (* Full 32-bit base address stored *)
      Fl;
    ELSE (* 64-bit Mode *)
      DEST[0:15] := GDTR(Limit);
      DEST[16:79] := GDTR(Base); (* Full 64-bit base address stored *)
      Fl;
  FI;

## Flags Affected
None.
Protected Mode Exceptions

#UD 
If the LOCK prefix is used.

#GP(0) 
If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If CR4.UMIP = 1 and CPL > 0.

#SS(0) 
If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) 
If a page fault occurs.

#AC(0) 
If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

Real-Address Mode Exceptions

#UD 
If the LOCK prefix is used.

#GP 
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS 
If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD 
If the LOCK prefix is used.

#GP(0) 
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If CR4.UMIP = 1.

#SS(0) 
If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) 
If a page fault occurs.

#AC(0) 
If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) 
If a memory address referencing the SS segment is in a non-canonical form.

#UD 
If the LOCK prefix is used.

#GP(0) 
If the memory address is in a non-canonical form.
If CR4.UMIP = 1 and CPL > 0.

#PF(fault-code) 
If a page fault occurs.

#AC(0) 
If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
### SHA1RNDS4—Perform Four Rounds of SHA1 Operation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3A CC /r ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA1RNDS4 xmm1,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmm2/m128, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

### Operation

**SHA1RNDS4**

The function \( f() \) and Constant \( K \) are dependent on the value of the immediate.

\[
\begin{align*}
\text{IF } (\text{imm8}[1:0] = 0) & \text{ THEN } f() := f0(), K := K_0; \\
\text{ELSE IF } (\text{imm8}[1:0] = 1) & \text{ THEN } f() := f1(), K := K_1; \\
\text{ELSE IF } (\text{imm8}[1:0] = 2) & \text{ THEN } f() := f2(), K := K_2; \\
\text{ELSE IF } (\text{imm8}[1:0] = 3) & \text{ THEN } f() := f3(), K := K_3; \\
\end{align*}
\]

\( f_i; \)

\[
\begin{align*}
A & := \text{SRC1}[127:96]; \\
B & := \text{SRC1}[95:64]; \\
C & := \text{SRC1}[63:32]; \\
D & := \text{SRC1}[31:0]; \\
W_0E & := \text{SRC2}[127:96]; \\
W_1 & := \text{SRC2}[95:64]; \\
W_2 & := \text{SRC2}[63:32]; \\
W_3 & := \text{SRC2}[31:0]; \\
\end{align*}
\]

Round \( i = 0 \) operation:
\[
\begin{align*}
A_{-1} & := f(B, C, D) + (A \text{ ROL } 5) + W_0E + K; \\
B_{-1} & := A; \\
C_{-1} & := B \text{ ROL } 30; \\
D_{-1} & := C; \\
E_{-1} & := D; \\
\end{align*}
\]

FOR \( i = 1 \) to 3
\[
\begin{align*}
A_{(i + 1)} & := f(B_i, C_i, D_i) + (A_i \text{ ROL } 5) + W_i + E_{-i} + K; \\
\end{align*}
\]
B_(i +1) := A_i;
C_(i +1) := B_i ROL 30;
D_(i +1) := C_i;
E_(i +1) := D_i;
ENDFOR

DEST[127:96] := A_4;
DEST[95:64] := B_4;
DEST[63:32] := C_4;
DEST[31:0] := D_4;

**Intel C/C++ Compiler Intrinsic Equivalent**
SHA1RND4 __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);

**Flags Affected**
None.

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-21, "Type 4 Class Exception Conditions."
SHA1NEXTE—Calculate SHA1 State Variable E After Four Rounds

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP OF 3B C8 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

**Operation**

SHA1NEXTE

\[
\text{TMP} := (\text{SRC1}[127:96] \text{ ROL } 30);
\]

\[
\text{DEST}[127:96] := \text{SRC2}[127:96] + \text{TMP}; \\
\text{DEST}[95:64] := \text{SRC2}[95:64]; \\
\text{DEST}[63:32] := \text{SRC2}[63:32]; \\
\text{DEST}[31:0] := \text{SRC2}[31:0];
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA1NEXTE __m128i _mm_sha1nexte_epu32(__m128i, __m128i);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, “Type 4 Class Exception Conditions.”
SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

**Operation**

SHA1MSG1

\[
\begin{align*}
W0 & := \text{SRC1}[127:96]; \\
W1 & := \text{SRC1}[95:64]; \\
W2 & := \text{SRC1}[63:32]; \\
W3 & := \text{SRC1}[31:0]; \\
W4 & := \text{SRC2}[127:96]; \\
W5 & := \text{SRC2}[95:64]; \\
\text{DEST}[127:96] & := W2 \oplus W0; \\
\text{DEST}[95:64] & := W3 \oplus W1; \\
\text{DEST}[63:32] & := W4 \oplus W2; \\
\text{DEST}[31:0] & := W5 \oplus W3;
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA1MSG1 __m128i _mm_sha1msg1_epu32(__m128i, __m128i);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions."
SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 CA /r</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.</td>
</tr>
<tr>
<td>SHA1MSG2 xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM reg (r, w)</td>
<td>ModRM r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

**Operation**

SHA1MSG2

W13 := SRC2[95:64];
W14 := SRC2[63:32];
W15 := SRC2[31:0];
W16 := (SRC1[127:96] XOR W13 ) ROL 1;
W17 := (SRC1[95:64] XOR W14) ROL 1;
W18 := (SRC1[63:32] XOR W15) ROL 1;
W19 := (SRC1[31:0] XOR W16) ROL 1;

DEST[127:96] := W16;
DEST[95:64] := W17;
DEST[63:32] := W18;
DEST[31:0] := W19;

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA1MSG2 __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions."
SHA256RNDS2—Perform Two Rounds of SHA256 Operation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 38 CB /r</td>
<td>RMI</td>
<td>V/V</td>
<td>SHA</td>
<td>Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM reg (r, w)</td>
<td>ModRM r/m (r)</td>
<td>Implicit XMM0 (r)</td>
</tr>
</tbody>
</table>

**Description**

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

**Operation**

SHA256RNDS2

A_0 := SRC2[127:96];
B_0 := SRC2[95:64];
C_0 := SRC1[127:96];
D_0 := SRC1[95:64];
E_0 := SRC2[63:32];
F_0 := SRC2[31:0];
G_0 := SRC1[63:32];
H_0 := SRC1[31:0];
WK0 := XMM0[31:0];
WK1 := XMM0[63:32];

FOR i = 0 to 1
A_(i +1) := Ch (E_i, F_i, G_i) + Σ1(E_i) +WK_i + H_i + Maj(A_i , B_i, C_i) +Σ0(A_i);
B_(i +1) := A_i;
C_(i +1) := B_i;
D_(i +1) := C_i;
E_(i +1) := Ch (E_i, F_i, G_i) + Σ1(E_i) +WK_i + H_i + D_i;
F_(i +1) := E_i;
G_(i +1) := F_i;
H_(i +1) := G_i;
ENDFOR

DEST[127:96] := A_2;
DEST[95:64] := B_2;
DEST[63:32] := E_2;
DEST[31:0] := F_2;
Intel C/C++ Compiler Intrinsic Equivalent
SHA256RND52 __m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected
None.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, “Type 4 Class Exception Conditions.”
SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

**Description**
The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

**Operation**
SHA256MSG1

\[
\begin{align*}
W4 &:= \text{SRC2}[31:0]; \\
W3 &:= \text{SRC1}[127:96]; \\
W2 &:= \text{SRC1}[95:64]; \\
W1 &:= \text{SRC1}[63:32]; \\
W0 &:= \text{SRC1}[31:0]; \\
\end{align*}
\]

\[
\begin{align*}
\text{DEST}[127:96] &:= W3 + \sigma_0( W4); \\
\text{DEST}[95:64] &:= W2 + \sigma_0( W3); \\
\text{DEST}[63:32] &:= W1 + \sigma_0( W2); \\
\text{DEST}[31:0] &:= W0 + \sigma_0( W1); \\
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**
SHA256MSG1 __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

**Flags Affected**
None.

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-21, "Type 4 Class Exception Conditions."
SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 3B CD Jr</td>
<td>RM</td>
<td>V/V</td>
<td>SHA</td>
<td>Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

**Operation**

SHA256MSG2

\[
\begin{align*}
W14 & := \text{SRC2}[95:64] \\
W15 & := \text{SRC2}[127:96] \\
W16 & := \text{SRC1}[31:0] + \sigma_1(W14) \\
W17 & := \text{SRC1}[63:32] + \sigma_1(W15) \\
W18 & := \text{SRC1}[95:64] + \sigma_1(W16) \\
W19 & := \text{SRC1}[127:96] + \sigma_1(W17) \\
\end{align*}
\]

\[
\begin{align*}
\text{DEST}[127:96] & := W19 \\
\text{DEST}[95:64] & := W18 \\
\text{DEST}[63:32] & := W17 \\
\text{DEST}[31:0] & := W16 \\
\end{align*}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

SHA256MSG2 __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, "Type 4 Class Exception Conditions."
**SHLD—Double Precision Shift Left**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F A4 /r ib</td>
<td>SHLD r/m16, r16, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to left imm8 places while shifting bits from r16 in from the right.</td>
</tr>
<tr>
<td>0F A5 /r</td>
<td>SHLD r/m16, r16, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to left CL places while shifting bits from r16 in from the right.</td>
</tr>
<tr>
<td>0F A4 /r ib</td>
<td>SHLD r/m32, r32, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to left imm8 places while shifting bits from r32 in from the right.</td>
</tr>
<tr>
<td>REX.W + 0F A4 /r ib</td>
<td>SHLD r/m64, r64, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to left imm8 places while shifting bits from r64 in from the right.</td>
</tr>
<tr>
<td>0F A5 /r</td>
<td>SHLD r/m32, r32, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to left CL places while shifting bits from r32 in from the right.</td>
</tr>
<tr>
<td>REX.W + 0F A5 /r</td>
<td>SHLD r/m64, r64, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to left CL places while shifting bits from r64 in from the right.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>MRC</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>CL</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT := COUNT MOD 64;
    ELSE COUNT := COUNT MOD 32;
FI
SIZE := OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
IF COUNT > SIZE
THEN (* Bad parameters *)
    DEST is undefined;
    CF, OF, SF, ZF, AF, PF are undefined;
ELSE (* Perform the shift *)
    CF := BIT[DEST, SIZE – COUNT];
    (* Last bit shifted out on exit *)
    FOR i := SIZE – 1 DOWN TO COUNT
        DO
            Bit(DEST, i) := Bit(DEST, i - COUNT);
            OD;
    FOR i := COUNT – 1 DOWN TO 0
        DO
            BIT[DEST, i] := BIT[SRC, i - COUNT + SIZE];
            OD;
    FI;
FI;

Flags Affected
If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
    If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used.
**SHRD—Double Precision Shift Right**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF AC /r ib</td>
<td>SHRD r/m16, r16, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to right imm8 places while shifting bits from r16 in from the left.</td>
</tr>
<tr>
<td>OF AD /r</td>
<td>SHRD r/m16, r16, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m16 to right CL places while shifting bits from r16 in from the left.</td>
</tr>
<tr>
<td>OF AC /r ib</td>
<td>SHRD r/m32, r32, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to right imm8 places while shifting bits from r32 in from the left.</td>
</tr>
<tr>
<td>REX.W + OF AC /r ib</td>
<td>SHRD r/m64, r64, imm8</td>
<td>MRI</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to right imm8 places while shifting bits from r64 in from the left.</td>
</tr>
<tr>
<td>OF AD /r</td>
<td>SHRD r/m32, r32, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>Valid</td>
<td>Shift r/m32 to right CL places while shifting bits from r32 in from the left.</td>
</tr>
<tr>
<td>REX.W + OF AD /r</td>
<td>SHRD r/m64, r64, CL</td>
<td>MRC</td>
<td>Valid</td>
<td>N.E.</td>
<td>Shift r/m64 to right CL places while shifting bits from r64 in from the left.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>MRC</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>CL</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT := COUNT MOD 64;
    ELSE COUNT := COUNT MOD 32;
FI
SIZE := OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
IF COUNT > SIZE
   THEN (* Bad parameters *)
      DEST is undefined;
      CF, OF, SF, ZF, AF, PF are undefined;
   ELSE (* Perform the shift *)
      CF := BIT[DEST, COUNT - 1]; (* Last bit shifted out on exit *)
      FOR i := 0 TO SIZE - 1 - COUNT
         DO
            BIT[DEST, i] := BIT[DEST, i + COUNT];
         OD;
      FOR i := SIZE - COUNT TO SIZE - 1
         DO
            BIT[DEST, i] := BIT[SRC, i + COUNT - SIZE];
         OD;
   FI;
FI;

Flags Affected
If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF,
and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred;
otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is unde-
fined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags
are undefined.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
SHUFPD—Packed Interleave Shuffle of Pairs of Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Shuffle two pairs of double precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 C6 /r ib VSHUFPD xmm1[k1][z], xmm2, xmm3/m128/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle two pairs of double precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8, AVX512VL AVX512F to select from each pair, store interleaved results in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 C6 /r ib VSHUFPD ymm1[k1][z], ymm2, ymm3/m256/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle four pairs of double precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8, AVX512VL AVX512F to select from each pair, store interleaved results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 C6 /r ib VSHUFPD zmm1[k1][z], zmm2, zmm3/m512/m64bcst, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle eight pairs of double precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8, AVX512F to select from each pair, store interleaved results in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction_operand_encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Selects a double precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4] are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of...
the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2] are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2] are ignored.

**Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of Double Precision Floating-Point Values**

**Operation**

**VSHUFPD (EVEX Encoded Versions When SRC2 is a Vector Register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF IMM0[0] = 0
THEN TMP_DEST[63:0] := SRC1[63:0]
ELSE TMP_DEST[63:0] := SRC1[127:64] FI;

IF IMM0[1] = 0
THEN TMP_DEST[127:64] := SRC2[63:0]
ELSE TMP_DEST[127:64] := SRC2[127:64] FI;

IF VL >= 256
IF IMM0[2] = 0

IF IMM0[3] = 0

FI;

IF VL >= 512
IF IMM0[4] = 0

IF IMM0[5] = 0

IF IMM0[6] = 0

IF IMM0[7] = 0

FI;
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:j] := TMP_DEST[i+63:j]
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:j] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:j] := 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VSHUFPD (EVEX Encoded Versions When SRC2 is Memory)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
THEN TMP_DEST[63:0] := SRC1[63:0]
ELSE TMP_DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
THEN TMP_DEST[127:64] := TMP_SRC2[63:0]
IF VL >= 256
    IF IMM0[2] = 0
    IF IMM0[3] = 0
    IF IMM0[4] = 0
    IF IMM0[5] = 0
    IF IMM0[6] = 0
    IF IMM0[7] = 0
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:j] := TMP_DEST[i+63:j]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST*[i+63:i] remains unchanged*
  ELSE *zeroing-masking* ; zeroing-masking
    DEST*[i+63:i] := 0
  FI
ENDIF;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VSHUFPD (VEX.256 Encoded Version)**

IF IMM0[0] = 0
  THEN DEST[63:0] := SRC1[63:0]
  ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
  THEN DEST[127:64] := SRC2[63:0]
  ELSE DEST[127:64] := SRC2[127:64] FI;
IF IMM0[2] = 0
IF IMM0[3] = 0
DEST[MAXVL-1:256] (Unmodified)

**VSHUFPD (VEX.128 Encoded Version)**

IF IMM0[0] = 0
  THEN DEST[63:0] := SRC1[63:0]
  ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
  THEN DEST[127:64] := SRC2[63:0]
  ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] := 0

**VSHUFPD (128-bit Legacy SSE Version)**

IF IMM0[0] = 0
  THEN DEST[63:0] := SRC1[63:0]
  ELSE DEST[63:0] := SRC1[127:64] FI;
IF IMM0[1] = 0
  THEN DEST[127:64] := SRC2[63:0]
  ELSE DEST[127:64] := SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSHUFPD _m512d _mm512_shuffle_pd(_m512d a, _m512d b, int imm);
VSHUFPD _m512d _mm512_mask_shuffle_pd(_m512d s, _mmask8 k, _m512d a, _m512d b, int imm);
VSHUFPD _m512d _mm512_maskz_shuffle_pd(_mmask8 k, _m512d a, _m512d b, int imm);
VSHUFPD _m256d _mm256_shuffle_pd(_m256d a, _m256d b, const int select);
VSHUFPD _m256d _mm256_mask_shuffle_pd(_mmask8 k, _m256d a, _m256d b, int imm);
VSHUFPD _m256d _mm256_maskz_shuffle_pd(_mmask8 k, _m256d a, _m256d b, int imm);
SHUFPD _m128d _mm_shuffle_pd(_m128d a, _m128d b, const int select);
VSHUFPD _m128d _mm_mask_shuffle_pd(_m128d s, _mmask8 k, _m128d a, _m128d b, int imm);
VSHUFPD _m128d _mm_maskz_shuffle_pd(_mmask8 k, _m128d a, _m128d b, int imm);
SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
SHUFPS—Packed Interleave Shuffle of Quadruplets of Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C6 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG C6 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG C6 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Select from quadruplet of single precision floating-point values in ymm1 and ymm2/m256 using imm8, interleaved result pairs are stored in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 C6 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Select from quadruplet of single precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 C6 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Select from quadruplet of single precision floating-point values in ymm1 and ymm2/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 C6 /r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Select from quadruplet of single precision floating-point values in zmm1 and zmm2/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:reg/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Selects a single precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.
128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

**Operation**

```
Select4(SRC, control) {
    CASE (control[1:0]) OF
        0: TMP := SRC[31:0];
        1: TMP := SRC[63:32];
        2: TMP := SRC[95:64];
        3: TMP := SRC[127:96];
    ESAC;
    RETURN TMP
}
```

**VPSHUFPS (EVEX Encoded Versions When SRC2 is a Vector Register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
TMP_DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
FI;
IF VL >= 512
    TMP_DEST[287:256] := Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[415:384] := Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[511:480] := Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[i] OR *no writemask*
```

**Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result**

...
THEN \(Dest[i+31:i] := \text{TMP}\_Dest[i+31:i] \)
ELSE
    IF *merging-masking*; merging-masking
        THEN \(Dest[i+31:i] \) remains unchanged*
    ELSE *zeroing-masking*; zeroing-masking
        \(Dest[i+31:i] := 0 \)
    FI;
ENDFOR

\(Dest[\text{MAXVL}-1:VL] := 0 \)

**VPHUFPS (EVEX Encoded Versions When SRC2 is Memory)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 \) TO \(KL-1\)
    \(i := j \ast 32\)
    IF (EVEX.b = 1)
        THEN \(\text{TMP}\_SRC2[i+31:i] := SRC2[31:0] \)
    ELSE \(\text{TMP}\_SRC2[i+31:i] := SRC2[i+31:i] \)
    FI;
ENDFOR;

\(\text{TMP}\_Dest[31:0] := \text{Select4}(SRC1[127:0], imm8[1:0]); \)
\(\text{TMP}\_Dest[63:32] := \text{Select4}(SRC1[127:0], imm8[3:2]); \)
\(\text{TMP}\_Dest[95:64] := \text{Select4}(\text{TMP}\_SRC2[127:0], imm8[5:4]); \)
\(\text{TMP}\_Dest[127:96] := \text{Select4}(\text{TMP}\_SRC2[127:0], imm8[7:6]); \)

IF \(VL >= 256\)
    THEN \(\text{TMP}\_Dest[159:128] := \text{Select4}(SRC1[255:128], imm8[1:0]); \)
    \(\text{TMP}\_Dest[191:160] := \text{Select4}(SRC1[255:128], imm8[3:2]); \)
    \(\text{TMP}\_Dest[223:192] := \text{Select4}(\text{TMP}\_SRC2[255:128], imm8[5:4]); \)
    \(\text{TMP}\_Dest[255:224] := \text{Select4}(\text{TMP}\_SRC2[255:128], imm8[7:6]); \)
FI;

IF \(VL >= 512\)
    THEN \(\text{TMP}\_Dest[287:256] := \text{Select4}(SRC1[383:256], imm8[1:0]); \)
    \(\text{TMP}\_Dest[319:288] := \text{Select4}(SRC1[383:256], imm8[3:2]); \)
    \(\text{TMP}\_Dest[351:320] := \text{Select4}(\text{TMP}\_SRC2[383:256], imm8[5:4]); \)
    \(\text{TMP}\_Dest[383:352] := \text{Select4}(\text{TMP}\_SRC2[383:256], imm8[7:6]); \)
    \(\text{TMP}\_Dest[415:384] := \text{Select4}(SRC1[511:384], imm8[1:0]); \)
    \(\text{TMP}\_Dest[447:416] := \text{Select4}(SRC1[511:384], imm8[3:2]); \)
    \(\text{TMP}\_Dest[479:448] := \text{Select4}(\text{TMP}\_SRC2[511:384], imm8[5:4]); \)
    \(\text{TMP}\_Dest[511:480] := \text{Select4}(\text{TMP}\_SRC2[511:384], imm8[7:6]); \)
FI;

FOR \(j := 0 \) TO \(KL-1\)
    \(i := j \ast 32\)
    IF \(k1[i] \) OR *no writemask*
        THEN \(Dest[i+31:i] := \text{TMP}\_Dest[i+31:i] \)
    ELSE
        IF *merging-masking*; merging-masking
            THEN \(Dest[i+31:i] \) remains unchanged*
        ELSE *zeroing-masking*; zeroing-masking
            \(Dest[i+31:i] := 0 \)
        FI;
    FI;
ENDFOR

\(Dest[\text{MAXVL}-1:VL] := 0 \)
VSHUFPS (VEX.256 Encoded Version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[191:160] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:256] := 0

VSHUFPS (VEX.128 Encoded Version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0

SHUFPS (128-bit Legacy SSE Version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[95:64] := Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
SIDT—Store Interrupt Descriptor Table Register

**Opcode**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 /1</td>
<td>SIDT m</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Store IDTR to m.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. See the IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**
Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for information on loading the GDTR and IDTR.

**IA-32 Architecture Compatibility**
The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

**Operation**
IF instruction is SIDT
THEN
  IF OperandSize = 16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
  THEN
    DEST[0:15] := IDTR(Limit);
    DEST[16:47] := IDTR(Base); Fl; (* Full 32-bit base address stored *)
  ELSE (* 64-bit Mode *)
    DEST[0:15] := IDTR(Limit);
    DEST[16:79] := IDTR(Base); (* Full 64-bit base address stored *)
  FI;
FI;

**Flags Affected**
None.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If CR4.UMIP = 1 and CPL > 0.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If CR4.UMIP = 1.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#UD If the LOCK prefix is used.
#GP(0) If the memory address is in a non-canonical form.
If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
SLDT—Store Local Descriptor Table Register

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /0</td>
<td>SLDT r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Stores segment selector from LDTR in r/m16.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description
Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

### Operation

\[
\text{DEST} := \text{LDTR}(\text{SegmentSelector});
\]

### Flags Affected
None.

### Protected Mode Exceptions

- **#GP(0)**: If the destination is located in a non-writable segment.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
- **#UD**: If the LOCK prefix is used.

### Real-Address Mode Exceptions

- **#UD**: The SLDT instruction is not recognized in real-address mode.
Virtual-8086 Mode Exceptions

#UD    The SLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)    If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)    If the memory address is in a non-canonical form.
    If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)    If a page fault occurs.
#AC(0)    If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD    If the LOCK prefix is used.
SMSW—Store Machine Status Word

### Description
Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:
- SMSW r16 operand size 16, store CR0[15:0] in r16
- SMSW r32 operand size 32, zero-extend CR0[31:0], and store in r32
- SMSW r64 operand size 64, zero-extend CR0[63:0], and store in r64
- SMSW m16 operand size 16, store CR0[15:0] in m16
- SMSW m16 operand size 32, store CR0[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CR0[15:0] in m16 (not m64)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs if CR4.UMIP = 0. It is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on IA-32 and Intel 64 processors beginning with the Intel386 processors should use the MOV CR instruction to load the machine status word.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

\[
\text{DEST} := \text{CR0}[15:0];
\]
(* Machine status word *)

### Flags Affected
None.
Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If CR4.UMIP = 1 and CPL > 0.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If CR4.UMIP = 1.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made.

#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
If CR4.UMIP = 1 and CPL > 0.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

#UD If the LOCK prefix is used.
**SQRTPD—Square Root of Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 51 /r SQRTPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes Square Roots of the packed double precision floating-point values in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes Square Roots of the packed double precision floating-point values in ymm2/m256 and stores the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes Square Roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes Square Roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes Square Roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD computation of the square roots of the two, four or eight packed double precision floating-point values in the source operand (the second operand) stores the packed double precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

**VSQRTPD (EVEX Encoded Versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*) THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*) THEN
      DEST[i+63:i] := SQRT(SRC[i+63:i])
    ELSE
      DEST[i+63:i] := SQRT(SRC[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] := 0

**VSQRTPD (VEX.256 Encoded Version)**

DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[255:192] := SQRT(SRC[255:192])
DEST[MAXVL-1:256] := 0

**VSQRTPD (VEX.128 Encoded Version)**

DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] := 0

**SQRTPD (128-bit Legacy SSE Version)**

DEST[63:0] := SQRT(SRC[63:0])
DEST[127:64] := SQRT(SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);
VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);
VSQRTPD __m512d _mm512_maskz_sqrt_round_pd( __mmask8 k, __m512d a, int r);
VSQRTPD __m256d _mm256_sqrt_pd( __m256d a);
VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);
VSQRTPD __m256d _mm256_maskz_sqrt_pd( __mmask8 k, __m256d a, int r);
SQRTPD __m128d _mm_sqrt_pd( __m128d a);
VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);
VSQRTPD __m128d _mm_maskz_sqrt_pd( __mmask8 k, __m128d a, int r);
**SIMD Floating-Point Exceptions**
Invalid, Precision, Denormal.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-19, “Type 2 Class Exception Conditions,” additionally:
#UD If VEX.vvv != 1111B.

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions,” additionally:
#UD If EVEX.vvv != 1111B.
SQRTPS—Square Root of Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 51 /r SQRTPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes Square Roots of the packed single precision floating-point values in xmm2/m128 and stores the result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 51 /r VSQRTPS ymm1, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes Square Roots of the packed single precision floating-point values in ymm2/m256 and stores the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 51 /r VSQRTPS xmm1 [k1]{z}, xmm2/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes Square Roots of the packed single precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 51 /r VSQRTPS ymm1 [k1]{z}, ymm2/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes Square Roots of the packed single precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 51 /r VSQRTPS zmm1 [k1]{z}, zmm2/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes Square Roots of the packed single precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single precision floating-point values in the source operand (second operand) stores the packed single precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

EVEX.128 encoded version: The source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
Operation

**VSQRTPS (EVEX Encoded Versions)**

$$(KL, VL) = (4, 128), (8, 256), (16, 512)$$

IF $(VL = 512)$ AND $(EVEX.b = 1)$ AND $(SRC \ast is \ register\ast)$

THEN

$$SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);$$

ELSE

$$SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);$$

FI;

FOR $j := 0 \ TO \ KL-1$

$$i := j \ast 32$$

IF $k1[j]$ OR $*no\ writemask* \ THEN$

IF $(EVEX.b = 1)$ AND $(SRC \ast is \ memory\ast)$

THEN $DEST[i+31:i] := SQRT(SRC[31:0])$

ELSE $DEST[i+31:i] := SQRT(SRC[i+31:i])$

FI;

ELSE

IF $*merging\-masking* \ THEN$

THEN $*DEST[i+31:i] \ remains \ unchanged*$$

ELSE $\ast zeroing\-masking$

$$DEST[i+31:i] := 0$$

FI

FI;

ENDFOR

$DEST[MAXVL-1:VL] := 0$

**VSQRTPS (VEX.256 Encoded Version)**

$DEST[31:0] := SQRT(SRC[31:0])$

$DEST[63:32] := SQRT(SRC[63:32])$

$DEST[95:64] := SQRT(SRC[95:64])$

$DEST[127:96] := SQRT(SRC[127:96])$

$DEST[159:128] := SQRT(SRC[159:128])$


$DEST[255:224] := SQRT(SRC[255:224])$

**VSQRTPS (VEX.128 Encoded Version)**

$DEST[31:0] := SQRT(SRC[31:0])$

$DEST[63:32] := SQRT(SRC[63:32])$

$DEST[95:64] := SQRT(SRC[95:64])$

$DEST[127:96] := SQRT(SRC[127:96])$

$DEST[MAXVL-1:128] := 0$

**SQRTPS (128-bit Legacy SSE Version)**

$DEST[31:0] := SQRT(SRC[31:0])$

$DEST[63:32] := SQRT(SRC[63:32])$

$DEST[95:64] := SQRT(SRC[95:64])$

$DEST[127:96] := SQRT(SRC[127:96])$

$DEST[MAXVL-1:128] (Unmodified)$
Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 __mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 __mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 __mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 __mm256_sqrt_ps(__m256 a);
VSQRTPS __m256 __mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 __mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 __mm_sqrt_ps(__m128 a);
VSQRTPS __m128 __mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 __mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-19, "Type 2 Class Exception Conditions," additionally:
#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions," additionally:
#UD If EVEX.vvvv != 1111B.
**SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 51/r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Computes square root of the low double precision floating-point value in xmm2/m64 and stores the results in xmm1.</td>
</tr>
<tr>
<td>SQRTSD xmm1,xmm2/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.W1 51/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
<tr>
<td>VSQRTSD xmm1,xmm2, xmm3/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 51/r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes square root of the low double precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
<tr>
<td>VSQRTSD xmm1 {k1}[2], xmm2, xmm3/m64{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes the square root of the low double precision floating-point value in the second source operand and stores the double precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

**VSQRTSD (EVEX Encoded Version)**

IF (EVEX.b = 1) AND (SRC2 *is register*)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k[1][0] or *no writemask*
THEN
  DEST[63:0] := SQRT(SRC2[63:0])
ELSE
  IF *merging-masking* ; merging-masking
  THEN *DEST[63:0] remains unchanged* ; zeroing-masking
  ELSE

SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value
THEN DEST[63:0] := 0
FI;

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**VSQRTSD (VEX.128 Encoded Version)**
DEST[63:0] := SQRT(SRC2[63:0])
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**SQRTSD (128-bit Legacy SSE Version)**
DEST[63:0] := SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

**SIMD Floating-Point Exceptions**
Invalid, Precision, Denormal.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
SQRTSS—Compute Square Root of Scalar Single Precision Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 51 /r SQRTSS</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Computes square root of the low single precision floating-point value in xmm2/m32 and stores the results in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 51 /r VSQRTSS</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W0 51 /r VSQRTSS</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes square root of the low single precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes the square root of the low single precision floating-point value in the second source operand and stores the single precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

**VSQRTSS (EVEX Encoded Version)**

IF (EVEX.b = 1) AND (SRC2 *is register*)
THEN
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

IF k1[0] or *no writemask*
THEN DEST[31:0] := SQRT(SRC2[31:0])
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
         DEST[31:0] := 0
   FI;
**INSTRUCTION SET REFERENCE, M-U**

```
Ft:
DEST[MAXVL-1:128] := 0

**VSQRTSS (VEX.128 Encoded Version)**
DEST[31:0] := SQRT(SRC2[31:0])
DEST[MAXVL-1:128] := 0

**SQRTPSS (128-bit Legacy SSE Version)**
DEST[31:0] := SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 __mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 __mm_maskz_sqrt_round_ss(__mmask8 k, __m128 a, __m128 b, int r);
SQRTPSS __m128 __mm_sqrt_ss(__m128 a)

**SIMD Floating-Point Exceptions**
Invalid, Precision, Denormal.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-47, “Type E3 Class Exception Conditions.”
STAC—Set AC Flag in EFLAGS Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 CB STAC</td>
<td>ZO</td>
<td>V/V</td>
<td>SMAP</td>
<td>Set the AC flag in the EFLAGS register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register. This instruction’s operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

Operation
EFLAGS.AC := 1;

Flags Affected
AC set. Other flags are unaffected.

Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If the CPL > 0.
   If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.
   If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions
#UD The STAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.
   If the CPL > 0.
   If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
   If the CPL > 0.
   If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.
STC—Set Carry Flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F9</td>
<td>STC</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set CF flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Sets the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation
CF := 1;

Flags Affected
The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)
#UD If the LOCK prefix is used.
STD—Set Direction Flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td>STD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set DF flag.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

**Operation**

\[ DF := 1; \]

**Flags Affected**

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD  
If the LOCK prefix is used.
STI—Set Interrupt Flag

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. This allows the processor to respond to maskable hardware interrupts.

If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI. (The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure or subroutine. For instance, if an STI instruction is followed by a RET instruction, the RET instruction is allowed to execute before external interrupts are recognized. No interrupts can be recognized if an execution of CLI immediately follow such an execution of STI.) The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and nonmaskable interrupts (NMIs). However, NMIs (and system-management interrupts) may be inhibited on the instruction boundary following an execution of STI that begins with IF = 0.

Operation is different in two modes defined as follows:

• **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
• **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-19 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

Table 4-19. Decision Table for STI Results

<table>
<thead>
<tr>
<th>Mode</th>
<th>IOPL</th>
<th>EFLAGS.VIP</th>
<th>STI Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-address</td>
<td>X</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>Protected, not PVI&lt;sup&gt;2&lt;/sup&gt;</td>
<td>≥ CPL</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>Protected, PVI&lt;sup&gt;3&lt;/sup&gt;</td>
<td>3</td>
<td>0</td>
<td>VIF = 1</td>
</tr>
<tr>
<td>Virtual-8086, not VME&lt;sup&gt;3&lt;/sup&gt;</td>
<td>3</td>
<td>X</td>
<td>IF = 1</td>
</tr>
<tr>
<td>Virtual-8086, VME&lt;sup&gt;3&lt;/sup&gt;</td>
<td>0-2</td>
<td>X</td>
<td>#GP fault</td>
</tr>
</tbody>
</table>

NOTES:
1. X = This setting has no effect on instruction operation.
2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

**Operation**

IF CR0.PE = 0 (* Executing in real-address mode *)
THEN IF := 1; (* Set Interrupt Flag *)
ELSE
  IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
  THEN IF := 1; (* Set Interrupt Flag *)
  ELSE
    IF VME mode OR PVI mode
    THEN
      IF EFLAGS.VIP = 0
      THEN VIF := 1; (* Set Virtual Interrupt Flag *)
      ELSE #GP(0);
    FI;
    ELSE #GP(0);
  FI;
FI;

**Flags Affected**

Either the IF flag or the VIF flag is set to 1. Other flags are unaffected.

**Protected Mode Exceptions**

#GP(0) If CPL is greater than IOPL and PVI mode is not active.
If CPL is greater than IOPL and EFLAGS.VIP = 1.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If IOPL is less than 3 and VME mode is not active.
If IOPL is less than 3 and EFLAGS.VIP = 1.
#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
STMXCSR—Store MXCSR Register State

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /3 STMXCSR m32</td>
<td>M</td>
<td>V/V</td>
<td>SSE</td>
<td>Store contents of MXCSR register to m32.</td>
</tr>
<tr>
<td>VEX.LZ.0F.WIG AE /3 VSTMXCSR m32</td>
<td>M</td>
<td>V/V</td>
<td>AVX</td>
<td>Store contents of MXCSR register to m32.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s. This instruction’s operation is the same in non-64-bit modes and 64-bit mode. VEX.L must be 0, otherwise instructions will #UD. Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

m32 := MXCSR;

**Intel C/C++ Compiler Intrinsic Equivalent**

_mm_getcsr(void)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-22, “Type 5 Class Exception Conditions,” additionally:

- **#UD**
  - If VEX.L= 1,
  - If VEX.vvvv ≠ 1111B.

---

**Intel C/C++ Compiler Intrinsic Equivalent**

_mm_getcsr(void)
STOS/STOSB/STOSW/STOSD/STOSQ—Store String

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>STOS m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m16</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOS m32</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.</td>
</tr>
<tr>
<td>REX.W + AB</td>
<td>STOS m64</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Store RAX at address RDI or EDI.</td>
</tr>
<tr>
<td>AA</td>
<td>STOSB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOSW</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.</td>
</tr>
<tr>
<td>AB</td>
<td>STOSD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.</td>
</tr>
<tr>
<td>REX.W + AB</td>
<td>STOSQ</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Store RAX at address RDI or EDI.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct location. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

NOTE

To improve performance, more recent processors support modifications to the processor’s operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional information on fast-string operation.
In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block stores of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See “REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

**Operation**

**Non-64-bit Mode:**

IF (Byte store)

    THEN
    DEST := AL;
    THEN IF DF = 0
    THEN (E)DI := (E)DI + 1;
    ELSE (E)DI := (E)DI - 1;
    FI;
ELSE IF (Word store)

    THEN
    DEST := AX;
    THEN IF DF = 0
    THEN (E)DI := (E)DI + 2;
    ELSE (E)DI := (E)DI - 2;
    FI;
ELSE IF (Doubleword store)

    THEN
    DEST := EAX;
    THEN IF DF = 0
    THEN (E)DI := (E)DI + 4;
    ELSE (E)DI := (E)DI - 4;
    FI;
FI;

**64-bit Mode:**

IF (Byte store)

    THEN
    DEST := AL;
    THEN IF DF = 0
    THEN (R|E)DI := (R|E)DI + 1;
    ELSE (R|E)DI := (R|E)DI - 1;
    FI;
ELSE IF (Word store)

    THEN
    DEST := AX;
    THEN IF DF = 0
    THEN (R|E)DI := (R|E)DI + 2;
    ELSE (R|E)DI := (R|E)DI - 2;
    FI;
ELSE IF (Doubleword store)
THEN
  DEST := EAX;
  THEN IF DF = 0
    THEN (R|E)DI := (R|E)DI + 4;
    ELSE (R|E)DI := (R|E)DI - 4;
  FI;
  FI;
ELSE IF (Quadword store using REX.W )
  THEN
    DEST := RAX;
    THEN IF DF = 0
      THEN (R|E)DI := (R|E)DI + 8;
      ELSE (R|E)DI := (R|E)DI - 8;
    FI;
  FI;
FI;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If the destination is located in a non-writable segment.
If a memory operand effective address is outside the limit of the ES segment.
If the ES register contains a NULL segment selector.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the ES segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the ES segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.
### STR—Store Task Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /1</td>
<td>STR r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Stores segment selector from TR in r/m16.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

#### Operation

\[
\text{DEST} := \text{TR} (\text{SegmentSelector});
\]

#### Flags Affected

None.

#### Protected Mode Exceptions

- **#GP(0)**
  - If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
  - If CR4.UMIP = 1 and CPL > 0.

- **#SS(0)**
  - If a memory operand effective address is outside the SS segment limit.

- **#PF(fault-code)**
  - If a page fault occurs.

- **#AC(0)**
  - If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

- **#UD**
  - If the LOCK prefix is used.

#### Real-Address Mode Exceptions

- **#UD**
  - The STR instruction is not recognized in real-address mode.

#### Virtual-8086 Mode Exceptions

- **#UD**
  - The STR instruction is not recognized in virtual-8086 mode.

#### Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
   If CR4.UMIP = 1 and CPL > 0.
#SS(0) If the stack address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
current privilege level is 3.
#UD If the LOCK prefix is used.
STTILECFG—Store Tile Configuration

Op Code/ Instruction | Op/En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
VEX.128.66.0F38.W0 49 (11):000:bbb STTILECFG m512 | A | V/N.E. | AMX-TILE | Store tile configuration in m512.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The STTILECFG instruction takes a pointer to a 64-byte memory location (described in Table 3-10 in the “LDTILECFG—Load Tile Configuration” entry) that will, after successful execution of this instruction, contain the description of the tiles that were configured. In order to configure tiles, the AMX-TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

If the tiles are not configured, then STTILECFG stores 64B of zeros to the indicated memory location.

Any attempt to execute the STTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

STTILECFG mem

if TILES_CONFIGURED == 0:
  // write 64 bytes of zeros at mem pointer
  buf[0..63] := 0
  write_memory(mem, 64, buf)
else:
  buf.byte[0] := tilecfg.palette_id
  buf.byte[1] := tilecfg.start_row
  buf.byte[2..15] := 0
  p := 16
  for n in 0 ... palette_table[tilecfg.palette_id].max_names-1:
    buf.word[p/2] := tilecfg.t[n].cols
    p := p + 2
    if p < 47:
      buf.byte[p..47] := 0
    p := 48
  for n in 0 ... palette_table[tilecfg.palette_id].max_names-1:
    buf.byte[p++] := tilecfg.t[n].rows
    if p < 63:
      buf.byte[p..63] := 0
  write_memory(mem, 64, buf)

Intel C/C++ Compiler Intrinsic Equivalent

STTILECFGvoid _tile_storeconfig(void *);

Flags Affected

None.
Exceptions
AMX-E2; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.
STUI—Set User Interrupt Flag

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 01 EF STUI</td>
<td>Z0</td>
<td>V/I</td>
<td>UINTR</td>
<td>Set user interrupt flag.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

STUI sets the user interrupt flag (UIF). Its effect takes place immediately; a user interrupt may be delivered on the instruction boundary following STUI. (This is in contrast with STI, whose effect is delayed by one instruction). An execution of STUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been due to an execution of STI.

Operation

UIF := 1;

Flags Affected

None.

Protected Mode Exceptions

#UD The STUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The STUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The STUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The STUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If executed inside an enclave.
If CR4.UINTR = 0.
If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.
SUB—Subtract

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (r, w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

### Notes

1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

**Operation**

DEST := (DEST – SRC);

**Flags Affected**

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

**Protected Mode Exceptions**

- #GP(0) If the destination is located in a non-writable segment.
  - If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
### SUBPD—Subtract Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 5C /r SUBPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Subtract packed double precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed double precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed double precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 5C /r VSUBPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed double precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 5C /r VSUBPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed double precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 5C /r VSUBPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Subtract packed double precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD subtract of the two, four or eight packed double precision floating-point values of the second Source operand from the first Source operand, and stores the packed double precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*


ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

DEST[63:0] := 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VSUBPD (EVEX Encoded Versions When SRC2 Operand is a Memory Source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN

IF (EVEX.b = 1)

THEN DEST[i+63:i] := SRC1[i+63:i] - SRC2[63:0];

ELSE DEST[i+63:i] := SRC1[i+63:i] - SRC2[i+63:i];

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

DEST[63:0] := 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

VSUBPD (VEX.256 Encoded Version)

DEST[63:0] := SRC1[63:0] - SRC2[63:0]


DEST[MAXVL-1:256] := 0
VSUBPD (VEX.128 Encoded Version)
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[MAXVL-1:128] := 0

SUBPD (128-bit Legacy SSE Version)
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
SUBPS—Subtract Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 5C /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Subtract packed single precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 5C /r SUBPS xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed single precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 5C /r SUBPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract packed single precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 5C /r VSUBPS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed single precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 5C /r VSUBPS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Subtract packed single precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 5C /r VSUBPS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Subtract packed single precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD subtract of the packed single precision floating-point values in the second Source operand from the First Source operand and stores the packed single precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Vector Register)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

\[
\text{IF} \ (VL = 512) \ \text{AND} \ (\text{EVEX.b = 1}) \\
\text{THEN} \\
\quad \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);} \\
\text{ELSE} \\
\quad \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);} \\
\]
FOR \( j := 0 \) TO \( KL-1 \),
\[ i := j \times 32 \]
IF \( k1[j] \) OR *no writemask*
THEN \( \text{DEST}[i+31:i] := \text{SRC1}[i+31:i] - \text{SRC2}[i+31:i] \)
ELSE
IF *merging-masking* ; merging-masking
THEN *\text{DEST}[31:0] remains unchanged*
ELSE ; zeroing-masking
\( \text{DEST}[31:0] := 0 \)
FI;
FI;
ENDFOR;
\( \text{DEST}[\text{MAXVL}-1:\text{VL}] := 0 \)

**VSUBPS (EVEX Encoded Versions When SRC2 Operand is a Memory Source)**
\((KL, VL) = (4, 128), (8, 256), (16, 512)\)
FOR \( j := 0 \) TO \( KL-1 \),
\[ i := j \times 32 \]
IF \( k1[j] \) OR *no writemask* THEN
IF (EVEX.b = 1)
THEN \( \text{DEST}[i+31:i] := \text{SRC1}[i+31:i] - \text{SRC2}[31:0] \);
ELSE \( \text{DEST}[i+31:i] := \text{SRC1}[i+31:i] - \text{SRC2}[i+31:i] \);
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *\text{DEST}[31:0] remains unchanged*
ELSE ; zeroing-masking
\( \text{DEST}[31:0] := 0 \)
FI;
FI;
ENDFOR;
\( \text{DEST}[\text{MAXVL}-1:\text{VL}] := 0 \)

**VSUBPS (VEX.256 Encoded Version)**
\( \text{DEST}[31:0] := \text{SRC1}[31:0] - \text{SRC2}[31:0] \)
\( \text{DEST}[95:64] := \text{SRC1}[95:64] - \text{SRC2}[95:64] \)
\( \text{DEST}[\text{MAXVL}-1:128] := 0 \)

**VSUBPS (VEX.128 Encoded Version)**
\( \text{DEST}[31:0] := \text{SRC1}[31:0] - \text{SRC2}[31:0] \)
\( \text{DEST}[95:64] := \text{SRC1}[95:64] - \text{SRC2}[95:64] \)
\( \text{DEST}[\text{MAXVL}-1:128] := 0 \)
**SUBPS (128-bit Legacy SSE Version)**

DEST[31:0] := SRC1[31:0] - SRC2[31:0]


DEST[95:64] := SRC1[95:64] - SRC2[95:64]


DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);

VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);

VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);

VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);

VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);

VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);

VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);

VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);

VSUBPS __m256 _mm256_maskz_sub_ps (__mmask8 k, __m256 a, __m256 b);

SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);

VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);

VSUBPS __m128 _mm_maskz_sub_ps (__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
SUBSD—Subtract Scalar Double Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 5C /r</td>
<td>A V/V</td>
<td>SSE2</td>
<td></td>
<td>Subtract the low double precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F2.0F.WIG 5C /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSUBSD xmm1,xmm2, xmm3/m64</td>
<td>B V/V</td>
<td>AVX</td>
<td></td>
<td>Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 5C /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSUBSD xmm1 (k1)[z], xmm2, xmm3/m64{er}</td>
<td>C V/V</td>
<td>AVX512F</td>
<td></td>
<td>Subtract the low double precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Subtract the low double precision floating-point value in the second source operand from the first source operand and stores the double precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSD (EVEX Encoded Version)

IF (SRC2 *is register*) AND (EVEX.b = 1)  
THEN  
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  
ELSE  
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);  
FI;  
IF k1[0] or *no writemask*  
THEN DEST[63:0] := SRC1[63:0] - SRC2[63:0]  
ELSE  
IF *merging-masking* ; merging-masking  
THEN *DEST[63:0] remains unchanged*  
ELSE ; zeroing-masking  
THEN DEST[63:0] := 0  
FI;  
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**VSUBSD (VEX.128 Encoded Version)**
DEST[63:0] := SRC1[63:0] - SRC2[63:0]
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**SUBSD (128-bit Legacy SSE Version)**
DEST[63:0] := DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**
VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

**SIMD Floating-Point Exceptions**
Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
SUBSS—Subtract Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 5C /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Subtract the low single precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.F3.0F.WIG 5C /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Subtract the low single precision floating-point value in xmm1,xmm2,xmm3/m32 from xmm1 and store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LILG.F3.0F.W0 5C /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Subtract the low single precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Subtract the low single precision floating-point value from the second source operand and the first source operand and store the double precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX,128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSS (EVEX Encoded Version)

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

IF k1[0] or *no writemask*
    THEN
        DEST[31:0] := SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] := 0
        FI;
    FI;


**SUBSS (VEX.128 Encoded Version)**

DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[MAXVL-1:128] := 0

**VSUBSS (VEX.128 Encoded Version)**

DEST[31:0] := SRC1[31:0] - SRC2[31:0]
DEST[MAXVL-1:128] := 0

**SUBSS (128-bit Legacy SSE Version)**

DEST[31:0] := DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
**SWAPGS—Swap GS Base Register**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 F8</td>
<td>SWAPGS</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Exchanges the current GS base register value with the value contained in MSR address C0000102H.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

**Operation**

If CS.L ≠ 1 (* Not in 64-Bit Mode *)

Then

#UD; Fl;

If CPL ≠ 0

Then #GP(0); Fl;

tmp := GS.base;
GS.base := IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE := tmp;

**Flags Affected**

None.

**Protected Mode Exceptions**

#UD If Mode ≠ 64-Bit.

**Real-Address Mode Exceptions**

#UD If Mode ≠ 64-Bit.

**Virtual-8086 Mode Exceptions**

#UD If Mode ≠ 64-Bit.
**Compatibility Mode Exceptions**

#UD If Mode ≠ 64-Bit.

**64-Bit Mode Exceptions**

#GP(0) If CPL ≠ 0.

#UD If the LOCK prefix is used.
SYSCALL—Fast System Call

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 05</td>
<td>SYSCALL</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Fast call to privilege level 0 system procedures.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

When shadow stacks are enabled at a privilege level where the SYSCALL instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” and Chapter 17, “Control-flow Enforcement Technology (CET),” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for additional CET details.

### Instruction ordering

Instructions following a SYSCALL may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSCALL have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

### Operation

```
IF (CS.L ≠ 1) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)  (* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD;
FI;
RCX := RIP;  (* Will contain address of next instruction *)
RIP := IA32_LSTAR;
R11 := RFLAGS;
RFLAGS := RFLAGS AND NOT(IA32_FMASK);
CS.Selector := IA32_STAR[47:32] AND FFFCH  (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0;  (* Flat segment *)
CS.Limit := FFFFFFFH;  (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11;  (* Execute/read code, accessed *)
```
CS.S := 1;
CS.DPL := 0;
CS.P := 1;
CS.L := 1; /* Entry is to 64-bit mode */
CS.D := 0; /* Required if CS.L = 1 */
CS.G := 1; /* 4-KByte granularity */

IF ShadowStackEnabled(CPL)
    THEN /* adjust so bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width */
        IA32_PL3_SSP := LA_adjust(SSP);
        (* With shadow stacks enabled the system call is supported from Ring 3 to Ring 0 *)
        (* OS supporting Ring 0 to Ring 0 system calls or Ring 1/2 to ring 0 system call *)
        (* Must preserve the contents of IA32_PL3_SSP to avoid losing ring 3 state *)
    FI;

CPL := 0;

IF ShadowStackEnabled(CPL)
    SSP := 0;
    FI;

IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
    FI;

SS.Selector := IA32_STAR[47:32] + 8; /* SS just above CS */
(* Set rest of SS to a fixed value *)
SS.Base := 0; /* Flat segment */
SS.Limit := FFFFFH; /* With 4-KByte granularity, implies a 4-GByte limit */
SS.Type := 3; /* Read/write data, accessed */
SS.S := 1;
SS.DPL := 0;
SS.P := 1;
SS.B := 1; /* 32-bit stack segment */
SS.G := 1; /* 4-KByte granularity */

Flags Affected
All.

Protected Mode Exceptions
#UD The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions
#UD The SYSCALL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The SYSCALL instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The SYSCALL instruction is not recognized in compatibility mode.
### 64-Bit Mode Exceptions

**#UD**

- If $\text{IA32_EFER.SCE} = 0$.
- If the LOCK prefix is used.
**SYSENTER—Fast System Call**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Ope/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 34</td>
<td>SYSENTER</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fast call to privilege level 0 system procedures.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Ope/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature.
flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

\[\text{IF CPUID SEP bit is set}\]
\[\quad \text{THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)}\]
\[\quad \quad \text{THEN}\]
\[\quad \quad \quad \text{SYSENTER/SYSEXIT_Not_Supported; Fl;}\]
\[\quad \quad \text{ELSE}\]
\[\quad \quad \quad \text{SYSENTER/SYSEXIT_Supported; Fl;}\]
\[\quad \text{Fl;}\]

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level where SYSENTER instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0. Refer to Chapter 6, ”Procedure Calls, Interrupts, and Exceptions,” and Chapter 17, ”Control-flow Enforcement Technology (CET),“ in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional CET details.

**Instruction ordering.** Instructions following a SYSENTER may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSENTER have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

**Operation**

IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); Fl;

RFLAGS.VM := 0; (* Ensures protected mode execution *)
RFLAGS.IF := 0; (* Mask interrupts *)
IF in IA-32e mode
THEN
    RSP := IA32_SYSENTER_ESP;
    RIP := IA32_SYSENTER_EIP;
ELSE
    ESP := IA32_SYSENTER_ESP[31:0];
    EIP := IA32_SYSENTER_EIP[31:0];
Fl;
CS.Selector := IA32_SYSENTER_CS[15:0] AND FFFCH;
(* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0; (* Flat segment *)
CS.Limit := FFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11; (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 0;
CS.P := 1;
IF in IA-32e mode
THEN
    CS.L := 1; (* Entry is to 64-bit mode *)
    CS.D := 0; (* Required if CS.L = 1 *)
ELSE
    CS.L := 0;
    CS.D := 1; (* 32-bit code segment *)
Fl;
CS.G := 1; (* 4-KByte granularity *)
IF ShadowStackEnabled(CPL)
    THEN
        IF IA32_EFER.LMA = 0
            THEN IA32_PL3_SSP := SSP;
        ELSE (* adjust so bits 63:N get the value of bit N–1, where N is the CPU's maximum linear-address width *)
            IA32_PL3_SSP := LA_adjust(SSP);
        FI;
    FI;
CPL := 0;
IF ShadowStackEnabled(CPL)
    SSP := 0;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;
SS.Selector := CS.Selector + 8; (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base := 0; (* Flat segment *)
SS.Limit := FFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type := 3; (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 0;
SS.P := 1;
SS.B := 1; (* 32-bit stack segment*)
SS.G := 1; (* 4-KByte granularity *)

Flags Affected
VM, IF (see Operation above).

Protected Mode Exceptions
#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP The SYSENTER instruction is not recognized in real-address mode.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
SYSEXIT—Fast Return from Fast System Call

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compatab/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 35</td>
<td>SYSEXIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fast return to privilege level 3 user code.</td>
</tr>
<tr>
<td>REX.W + 0F 35</td>
<td>SYSEXIT</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Fast return to 64-bit mode privilege level 3 user code.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

IF CPUID SEP bit is set
THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
     THEN
     SYSENTER/SYSEXIT_Not_SUPPORTED; Fl;
     ELSE
     SYSENTER/SYSEXIT_Supported; Fl;
     Fl;

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.
When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR. Refer to Chapter 6, “Interrupt and Exception Handling,” and Chapter 17, “Control-flow Enforcement Technology (CET),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional CET details.

**Instruction ordering.** Instructions following a SYSEXIT may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSEXIT have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

**Operation**

IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;

IF operand size is 64-bit

THEN (* Return to 64-bit mode *)

RSP := RCX;
RIP := RDX;

ELSE (* Return to protected mode or compatibility mode *)

RSP := ECX;
RIP := EDX;

FI;

IF operand size is 64-bit (* Operating system provides CS; RPL forced to 3 *)

THEN CS.Selector := IA32_SYSENTER_CS[15:0] + 32;
ELSE CS.Selector := IA32_SYSENTER_CS[15:0] + 16;

FI;

CS.Selector := CS.Selector OR 3; (* RPL forced to 3 *)

(* Set rest of CS to a fixed value *)

CS.Base := 0; (* Flat segment *)
CS.Limit := FFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11; (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 3;
CS.P := 1;

IF operand size is 64-bit (* return to 64-bit mode *)

THEN CS.L := 1; (* 64-bit code segment *)
CS.D := 0; (* Required if CS.L = 1 *)
ELSE (* return to protected mode or compatibility mode *)

CS.L := 0;
CS.D := 1; (* 32-bit code segment*)

FI;

CS.G := 1; (* 4-KByte granularity *)
CPL := 3;

IF ShadowStackEnabled(CPL)

THEN SSP := IA32_PL3_SSP;

FI;

SS.Selector := CS.Selector + 8; (* SS just above CS *)

(* Set rest of SS to a fixed value *)

SS.Base := 0; (* Flat segment *)
SS.Limit := FFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type := 3; (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 3;
SS.P := 1;
SS.B := 1; (* 32-bit stack segment*)
SS.G := 1; (* 4-KByte granularity *)

**Flags Affected**
None.

**Protected Mode Exceptions**

#GP(0) If IA32_SYSENTER_CS[15:2] = 0.
    If CPL ≠ 0.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP The SYSEXIT instruction is not recognized in real-address mode.
#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) The SYSEXIT instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0) If IA32_SYSENTER_CS = 0.
    If CPL ≠ 0.
    If RCX or RDX contains a non-canonical address.
#UD If the LOCK prefix is used.
SYSRET—Return From Fast System Call

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 07</td>
<td>SYSRET</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Return to compatibility mode from fast system call.</td>
</tr>
<tr>
<td>REX.W + 0F 07</td>
<td>SYSRET</td>
<td>ZO</td>
<td>Valid</td>
<td>Invalid</td>
<td>Return to 64-bit mode from fast system call.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.1 With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- **External interrupts.** The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- **Nonmaskable interrupts (NMIs).** The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, “Interrupt Stack Table,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).
- **General-protection exceptions (#GP).** The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR. Refer to Chapter 6, “Procedure Calls, Interrupts, and Exceptions,” and Chapter 17, “Control-flow Enforcement Technology (CET),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional CET details.

---

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.
**Instruction ordering.** Instructions following a SYSRET may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the SYSRET have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).

**Operation**

IF (CS.L ≠ 1) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL ≠ 0) THEN #GP(0); FI;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        IF (RCX is not canonical) THEN #GP(0);
        RIP := RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP := ECX;
    FI;
RFLAGS := (R11 & 3C7FD7H) | 2; (* Clear RF, VM, reserved bits; set bit 1 *)
IF (operand size is 64-bit)
    THEN CS.Selector := IA32_STAR[63:48]+16;
    ELSE CS.Selector := IA32_STAR[63:48]:
    FI;
CS.Selector := CS.Selector OR 3; (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base := 0; (* Flat segment *)
CS.Limit := FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type := 11; (* Execute/read code, accessed *)
CS.S := 1;
CS.DPL := 3;
CS.P := 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L := 1; (* 64-bit code segment *)
        CS.D := 0; (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L := 0;
        CS.D := 1; (* 32-bit code segment *)
    FI;
CS.G := 1; (* 4-KByte granularity *)
CPL := 3;
IF ShadowStackEnabled(CPL)
    SSP := IA32_PL3_SSP;
    FI;
SS.Selector := (IA32_STAR[63:48]+8) OR 3; (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base := 0; (* Flat segment *)
SS.Limit := FFFFFFFH; (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type := 3; (* Read/write data, accessed *)
SS.S := 1;
SS.DPL := 3;
SS.P := 1;
SS.B := 1; (* 32-bit stack segment*)
SS.G := 1; (* 4-KByte granularity *)
INSTRUCTION SET REFERENCE, M-U

Flags Affected
All.

Protected Mode Exceptions
#UD The SYSRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions
#UD The SYSRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The SYSRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The SYSRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions
#UD If IA32_EFER.SCE = 0.
   If the LOCK prefix is used.
#GP(0) If CPL ≠ 0.
   If the return is to 64-bit mode and RCX contains a non-canonical address.
TDPBF16PS—Dot Product of BF16 Tiles Accumulated into Packed Single Precision Tile

**Description**

This instruction performs a set of SIMD dot-products of two BF16 elements and accumulates the results into a packed single precision tile. Each dword element in input tiles tmm2 and tmm3 is interpreted as a BF16 pair. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding BF16 pairs (one pair from tmm2 and one pair from tmm3), adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1.

"Round to nearest even" rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

Any attempt to execute the TDPBF16PS instruction inside a TSX transaction will result in a transaction abort.

**Operation**

```c
define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword.
    // The bit pattern is a legal fp32 value. Return that bit pattern.
    dword := 0
    dword[31:16] := x
    return dword

TDPBF16PS tsrcdest, tsrc1, tsrc2
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of bfloat16
elements_src1 := tsrc1.colsb / 4
elements_src2 := tsrc2.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2 // Count is in bfloat16 prior to horizontal

for m in 0 ... tsrcdest.rows-1:
    temp1[0 ... elements_temp-1] := 0
    for k in 0 ... elements_src1-1:
        for n in 0 ... elements_dest-1:
            // FP32 FMA with DAZ=FTZ=1, RNE rounding.
            // MXCSR is neither consulted nor updated.
            // No exceptions raised or denoted.
            temp1.fp32[2*n+0] += make_fp32(tsrc1.row[m].bfloat16[2*k+0]) * make_fp32(tsrc2.row[k].bfloat16[2*n+0])
            temp1.fp32[2*n+1] += make_fp32(tsrc1.row[m].bfloat16[2*k+1]) * make_fp32(tsrc2.row[k].bfloat16[2*n+1])
```

**Instruction Set Reference, M-U**

Vol. 2B 4-699
for n in 0 ... elements_dest-1:
    // DAZ=FTZ=1, RNE rounding.
    // MXCSR is neither consulted nor updated.
    // No exceptions raised or denoted.
    tsrcdest.row[m].fp32[n] := tsrcdest.row[m].fp32[n] + tmpf32
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

tsrcdest.row[m].fp32[n] := tsrcdest.row[m].fp32[n] + tmpf32
write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

Intel C/C++ Compiler Intrinsic Equivalent
TDPBF16PS void _tile_dpbf16ps(__tile dst, __tile src1, __tile src2);

Flags Affected
None.

Exceptions
AMX-E4; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.
TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD—Dot Product of Signed/Unsigned Bytes with Dword Accumulation

**Description**
For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding four byte elements, one from tmm2 and one from tmm3, adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1. Each dword in input tiles tmm2 and tmm3 is interpreted as four byte elements. These may be signed or unsigned. Each letter in the two-letter pattern SU, US, SS, UU indicates the signed/unsigned nature of the values in tmm2 and tmm3, respectively.

Any attempt to execute the TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD instructions inside an Intel TSX transaction will result in a transaction abort.

**Operation**

```c
define DPBD(c;x,y)// arguments are dwords
    if *x operand is signed:
        extend_src1 := SIGN_EXTEND
    else:
        extend_src1 := ZERO_EXTEND

    if *y operand is signed:
        extend_src2 := SIGN_EXTEND
    else:
        extend_src2 := ZERO_EXTEND

    p0dword := extend_src1(x.byte[0]) * extend_src2(y.byte[0])
    p1dword := extend_src1(x.byte[1]) * extend_src2(y.byte[1])
    p2dword := extend_src1(x.byte[2]) * extend_src2(y.byte[2])
    p3dword := extend_src1(x.byte[3]) * extend_src2(y.byte[3])

    c := c + p0dword + p1dword + p2dword + p3dword
```

---

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.F2.0F38.W0 5E 11:rrr:bbb</td>
<td>A</td>
<td>V/N.E.</td>
<td>AMX-INT8</td>
<td>Matrix multiply signed byte elements from tmm2 by signed byte elements from tmm3 and accumulate the dword elements in tmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 5E 11:rrr:bbb</td>
<td>A</td>
<td>V/N.E.</td>
<td>AMX-INT8</td>
<td>Matrix multiply unsigned byte elements from tmm2 by signed byte elements from tmm3 and accumulate the dword elements in tmm1.</td>
</tr>
<tr>
<td>VEX.128.NP.0F38.W0 5E 11:rrr:bbb</td>
<td>A</td>
<td>V/N.E.</td>
<td>AMX-INT8</td>
<td>Matrix multiply unsigned byte elements from tmm2 by unsigned byte elements from tmm3 and accumulate the dword elements in tmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>VEX.vvvv (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
TDPBSSD, TDPBSUD, TDPBUSD, TDPBUUD tsrcdest, tsrc1, tsrc2 (Register Only Version)

// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

tsrc1_elements_per_row := tsrc1.colsb / 4
tsrc2_elements_per_row := tsrc2.colsb / 4
tsrcdest_elements_per_row := tsrcdest.colsb / 4

for m in 0 ... tsrcdest.rows-1:
    tmp := tsrcdest.row[m]
    for k in 0 ... tsrc1_elements_per_row-1:
        for n in 0 ... tsrcdest_elements_per_row-1:
            DPBD( tmp.dword[n], tsrc1.row[m].dword[k], tsrc2.row[k].dword[n] )
            write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tilecfg_start()

Intel C/C++ Compiler Intrinsic Equivalent

TDPBSSD void _tile_dpbbssd(__tile dst, __tile src1, __tile src2);
TDPBSUD void _tile_dpbsud(__tile dst, __tile src1, __tile src2);
TDPBUSD void _tile_dpbusd(__tile dst, __tile src1, __tile src2);
TDPBUUD void _tile_dpbuud(__tile dst, __tile src1, __tile src2);

Flags Affected

None.

Exceptions

AMX-E4; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.
TEST—Logical Compare

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A8 ib</td>
<td>TEST AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AND imm8 with AL; set SF, ZF, PF according to result.</td>
</tr>
<tr>
<td>A9 iw</td>
<td>TEST AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>AND imm16 with AX; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>A9 id</td>
<td>TEST EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>AND imm32 with EAX; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>REX.W + A9 id</td>
<td>TEST RAX, imm32</td>
<td>I</td>
<td>N.E.</td>
<td>AND imm32 sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>F6 /0 ib</td>
<td>TEST r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>AND imm8 with r/m8; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>REX + F6 /0 ib</td>
<td>TEST r/m81, imm8</td>
<td>MI</td>
<td>N.E.</td>
<td>AND imm8 with r/m8; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>F7 /0 iw</td>
<td>TEST r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>AND imm16 with r/m16; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>F7 /0 id</td>
<td>TEST r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>AND imm32 with r/m32; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>REX.W + F7 /0 id</td>
<td>TEST r/m64, imm32</td>
<td>MI</td>
<td>N.E.</td>
<td>AND imm32 sign-extended to 64-bits with r/m64; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>84 /r</td>
<td>TEST r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>AND r8 with r/m8; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>REX + 84 /r</td>
<td>TEST r/m81, r81</td>
<td>MR</td>
<td>N.E.</td>
<td>AND r8 with r/m8; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>85 /r</td>
<td>TEST r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>AND r16 with r/m16; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>85 /r</td>
<td>TEST r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>AND r32 with r/m32; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
<tr>
<td>REX.W + 85 /r</td>
<td>TEST r/m64, r64</td>
<td>MR</td>
<td>N.E.</td>
<td>AND r64 with r/m64; set SF, ZF, PF according to result.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM:r/m (r)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation
\[
\text{TEMP} := \text{SRC1 AND SRC2}; \\
\text{SF} := \text{MSB(TEMP)}; \\
\text{IF TEMP = 0} \\
\quad \text{THEN ZF} := 1; \\
\text{ELSE ZF} := 0; \\
\text{FI}; \\
\text{PF} := \text{BitwiseXNOR(TEMP[0:7])};
\]
CF := 0;
OF := 0;
(* AF is undefined *)

Flags Affected
The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
      current privilege level is 3.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the
      current privilege level is 3.
#UD If the LOCK prefix is used.
TESTUI—Determine User Interrupt Flag

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F 01 ED TESTUI</td>
<td>ZO</td>
<td>V/I</td>
<td>UINTR</td>
<td>Copies the current value of UIF into EFLAGS.CF.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

TESTUI copies the current value of the user interrupt flag (UIF) into EFLAGS.CF. This instruction can be executed regardless of CPL.

TESTUI may be executed normally inside a transactional region.

Operation

CF := UIF;
ZF := AF := OF := PF := SF := 0;

Flags Affected

The ZF, OF, AF, PF, SF flags are cleared and the CF flags to the value of the user interrupt flag.

Protected Mode Exceptions

#UD The TESTUI instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The TESTUI instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The TESTUI instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The TESTUI instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If executed inside an enclave.
If CR4.UINTR = 0.
If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.
TILELOADD/TILELOADDT1—Load Tile

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction is required to use SIB addressing. The index register serves as a stride indicator. If the SIB encoding omits an index register, the value zero is assumed for the content of the index register.

This instruction loads a tile destination with rows and columns as specified by the tile configuration. The “T1” version provides a hint to the implementation that the data would be reused but does not need to be resident in the nearest cache levels.

The TILECFG.start_row in the TILECFG data should be initialized to 0 in order to load the entire tile and is set to zero on successful completion of the TILELOADD instruction. TILELOADD is a restartable instruction and the TILECFG.start_row will be non-zero when restartable events occur during the instruction execution.

Only memory operands are supported and they can only be accessed using a SIB addressing mode, similar to the V[P]GATHER*/V[P]SCATTER* instructions.

Any attempt to execute the TILELOADD/TILELOADDT1 instructions inside an Intel TSX transaction will result in a transaction abort.

**Operation**

TILELOADD[T1] tdest, tsib

\[
\text{start} := \text{tilecfg.start\_row} \\
\text{zero\_upper\_rows(tdest.start)} \\
\text{membegin} := \text{tsib.base + displacement} \\
// if no index register in the SIB encoding, the value zero is used. \\
\text{stride} := \text{tsib.index} \ll \text{tsib.scale} \\
\text{nbytes} := \text{tdest.colsb} \\
\text{while start < tdest.rows:} \\
\hspace{1em} \text{memptr} := \text{membegin + start \* stride} \\
\hspace{1em} \text{write\_row\_and\_zero(tdest, start, read\_memory(memptr, nbytes), nbytes)} \\
\hspace{1em} \text{start} := \text{start + 1} \\
\text{zero\_tilecfg\_start()} \\
// In the case of a memory fault in the middle of an instruction, the tilecfg.start\_row := start \\
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

TILELOADD void _tile_loadd(__tile dst, const void *base, int stride);
TILELOADDT1 void _tile_stream_loadd(__tile dst, const void *base, int stride);
Flags Affected
None.

Exceptions
AMX-E3; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.
TILERERELEASE—Release Tile

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.NP.0F38.W0 49 C0 TILERERELEASE</td>
<td>A</td>
<td>V/N.E.</td>
<td>AMX-TILE</td>
<td>Initialize TILECFG and TILEDATA.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction returns TILECFG and TILEDATA to the INIT state.

Any attempt to execute the TILERERELEASE instruction inside an Intel TSX transaction will result in a transaction abort.

**Operation**

```c
zero_all_tile_data()
tilecfg := 0 // equivalent to 64B of zeros
TILES_CONFIGURED := 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

TILERERELEASE void _tile_release(void);

**Flags Affected**

None.

**Exceptions**

AMX-E6; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.
TILESTORED—Store Tile

**Description**

This instruction is required to use SIB addressing. The index register serves as a stride indicator. If the SIB encoding omits an index register, the value zero is assumed for the content of the index register.

This instruction stores a tile source of rows and columns as specified by the tile configuration. The TILECFG.start_row in the TILECFG data should be initialized to '0' in order to store the entire tile and are set to zero on successful completion of the TILESTORED instruction. TILESTORED is a restartable instruction and the TILECFG.start_row will be non-zero when restartable events occur during the instruction execution.

Only memory operands are supported and they can only be accessed using a SIB addressing mode, similar to the V[P]GATHER*/V[P]SCATTER* instructions.

Any attempt to execute the TILESTORED instruction inside an Intel TSX transaction will result in a transaction abort.

**Operation**

TILESTORED tsib, tsrc

```markdown
start := tilecfg.start_row

membegin := tsib.base + displacement
// if no index register in the SIB encoding, the value zero is used.
stride := tsib.index << tsib.scale

while start < tdest.rows:
    memptr := membegin + start * stride
    write_memory(memptr, tsrc.colsb, tsrc.row[start])
    start := start + 1
zero_tilecfg_start()
// In the case of a memory fault in the middle of an instruction, the tilecfg.start_row := start
```

**Intel C/C++ Compiler Intrinsic Equivalent**

TILESTORED void _tile_stored(__tile src, void *base, int stride);

**Flags Affected**

None.

**Exceptions**

AMX-E3; see Section 2.10, "Intel® AMX Instruction Exception Classes," for details.
TILEZERO—Zero Tile

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.F2.0F38.W0 49 11:rrr:000 TILEZERO tmm1</td>
<td>A</td>
<td>V/N.E.</td>
<td>AMX-TILE</td>
<td>Zero the destination tile.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
This instruction zeroes the destination tile.
Any attempt to execute the TILEZERO instruction inside an Intel TSX transaction will result in a transaction abort.

Operation
TILEZERO tdest

nbytes := palette_table[palette_id].bytes_per_row

for i in 0 ... palette_table[palette_id].max_rows-1:
    for j in 0 ... nbytes-1:
        tdest.row[i].byte[j] := 0
    zero_tilecfg_start()

Intel C/C++ Compiler Intrinsic Equivalent
TILEZERO void _tile_zero(__tile dst);

Flags Affected
None.

Exceptions
AMX-E5; see Section 2.10, “Intel® AMX Instruction Exception Classes,” for details.
TPAUSE—Timed PAUSE

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F AE /6 TPAUSE r32, &lt;edx&gt;, &lt;eax&gt;</td>
<td>A</td>
<td>V/V</td>
<td>WAITPKG</td>
<td>Directs the processor to enter an implementation-dependent optimized state until the TSC reaches the value in EDX:EAX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

TPAUSE instructs the processor to enter an implementation-dependent optimized state. There are two such optimized states to choose from: light-weight power/performance optimized state, and improved power/performance optimized state. The selection between the two is governed by the explicit input register bit[0] source operand.

TPAUSE is available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. TPAUSE may be executed at any privilege level. This instruction’s operation is the same in non-64-bit modes and in 64-bit mode.

Unlike PAUSE, the TPAUSE instruction will not cause an abort when used inside a transactional region, described in the chapter Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if non-zero.

**Table 4-20. TPAUSE Input Register Bit Definitions**

<table>
<thead>
<tr>
<th>Bit Value</th>
<th>State Name</th>
<th>Wakeup Time</th>
<th>Power Savings</th>
<th>Other Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit[0] = 0</td>
<td>C0.2</td>
<td>Slower</td>
<td>Larger</td>
<td>Improves performance of the other SMT thread(s) on the same core.</td>
</tr>
<tr>
<td>bit[0] = 1</td>
<td>C0.1</td>
<td>Faster</td>
<td>Smaller</td>
<td>N/A</td>
</tr>
<tr>
<td>bits[31:1]</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The instruction execution wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value.

Prior to executing the TPAUSE instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32-bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of “1” means all C0.2 requests revert to C0.1.

If the processor that executed a TPAUSE instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the read-set range within the transactional region, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal.

1. The Mod field of the ModR/M byte must have value 11B.
Other implementation-dependent events may cause the processor to exit the implementation-dependent optimized state proceeding to the instruction following TPAUSE. In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0). It should be noted that if maskable-interrupts are inhibited execution will proceed to the instruction following TPAUSE.

**Operation**

\[
\begin{align*}
\text{os\_deadline} & := \text{TSC} + (\text{IA32\_UMWAIT\_CONTROL}[31:2] << 2) \\
\text{instr\_deadline} & := \text{UINT64}(\text{EDX:EAX})
\end{align*}
\]

IF \( \text{os\_deadline} < \text{instr\_deadline} \):
- \( \text{deadline} := \text{os\_deadline} \)
- \( \text{using\_os\_deadline} := 1 \)
ELSE:
- \( \text{deadline} := \text{instr\_deadline} \)
- \( \text{using\_os\_deadline} := 0 \)

WHILE \( \text{TSC} < \text{deadline} \):
- \( \text{implementation\_dependent\_optimized\_state(Source\ register, deadline, IA32\_UMWAIT\_CONTROL[0])} \)

IF \( \text{using\_os\_deadline} \) AND \( \text{TSC} \geq \text{deadline} \):
- \( \text{RFLAGS.CF} := 1 \)
ELSE:
- \( \text{RFLAGS.CF} := 0 \)

\( \text{RFLAGS.AF,PF,SF,ZF,OF} := 0 \)

**Intel C/C++ Compiler Intrinsic Equivalent**

\`
\text{TPAUSE} \text{uint8\_t \_tpause(uint32\_t control, uint64\_t counter);}
``

**Numeric Exceptions**

None.

**Exceptions (All Operating Modes)**

- **#GP(0)**
  - If src[31:1] ! = 0.
  - If CR4.TSD = 1 and CPL ! = 0.
- **#UD**
  - If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
TZCNT—Count the Number of Trailing Zero Bits

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F BC/r TZCNT r16, r/m16</td>
<td>A</td>
<td>V/V</td>
<td>BMI1</td>
<td>Count the number of trailing zero bits in r/m16, return result in r16.</td>
</tr>
<tr>
<td>F3 0F BC/r TZCNT r32, r/m32</td>
<td>A</td>
<td>V/V</td>
<td>BMI1</td>
<td>Count the number of trailing zero bits in r/m32, return result in r32.</td>
</tr>
<tr>
<td>F3 REX.W OF BC/r TZCNT r64, r/m64</td>
<td>A</td>
<td>V/N.E.</td>
<td>BMI1</td>
<td>Count the number of trailing zero bits in r/m64, return result in r64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

temp := 0
DEST := 0
DO WHILE ((temp < OperandSize) and (SRC[ temp] = 0 ))
    temp := temp +1
    DEST := DEST + 1
OD

IF DEST = OperandSize
    CF := 1
ELSE
    CF := 0
FI

IF DEST = 0
    ZF := 1
ELSE
    ZF := 0
FI

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF, and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

TZCNT unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT unsigned __int64 _tzcnt_u64(unsigned __int64 src);
Protected Mode Exceptions

#GP(0) For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
   If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) For an illegal address in the SS segment.
#UD If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0) For an illegal address in the SS segment.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code) For a page fault.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If LOCK prefix is used.
UCOMISD—Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 2E /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F.WIG 2E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Compare low double precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F.W1 2E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare low double precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Description

Performs an unordered compare of the double precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid operation exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISD (All Versions)
RESULT := UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF := 111;
    GREATER_THAN: ZF,PF,CF := 000;
    LESS_THAN: ZF,PF,CF := 001;
    EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0; }
Intel C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions
Invalid (if SNaN operands), Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, "Type 3 Class Exception Conditions," additionally:
#UD If VEX.vvvv != 1111B.
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
UCOMISS—Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| NP 0F 2E /r  
UCOMISS xmm1, xmm2/m32 | A | V/V | SSE  
Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |
| VEX.LIG.0F.WIG 2E /r  
VUCOMISS xmm1, xmm2/m32 | A | V/V | AVX  
Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |
| EVEX.LIG.0F.WD 2E /r  
VUCOMISS xmm1, xmm2/m32{sae} | B | V/V | AVX512F  
Compare low single precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly. |

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Compares the single precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid operation exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

(V)UCOMISS (All Versions)

RESULT := UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
  UNORDERED: ZF,PF,CF := 111;
  GREATER_THAN: ZF,PF,CF := 000;
  LESS_THAN: ZF,PF,CF := 001;
  EQUAL: ZF,PF,CF := 100;
ESAC;
OF, AF, SF := 0;
}

### Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISS  
int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS  
int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS  
int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS  
int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS  
int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS  
int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS  
int _mm_ucomineq_ss(__m128 a, __m128 b);
**SIMD Floating-Point Exceptions**
Invalid (if SNaN Operands), Denormal.

**Other Exceptions**
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions,” additionally:
#UD If VEX.vvvv != 1111B.
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
UD—Undefined Instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F FF /r</td>
<td>UD0* r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Raise invalid opcode exception.</td>
</tr>
<tr>
<td>0F B9 /r</td>
<td>UD1 r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Raise invalid opcode exception.</td>
</tr>
<tr>
<td>0F 0B</td>
<td>UD2</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Raise invalid opcode exception.</td>
</tr>
</tbody>
</table>

**Operation**

#UD (* Generates invalid opcode exception *);

**Flags Affected**

None.

**Exceptions (All Operating Modes)**

#UD Raises an invalid opcode exception in all operating modes.

NOTES:

1. Some processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid-opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.
UIRET—User-Interrupt Return

**Description**

UIRET returns from the handling of a user interrupt. It can be executed regardless of CPL.

Execution of UIRET inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been due to an execution of IRET.

UIRET can be tracked by Architectural Last Branch Records (LBRs), Intel Processor Trace (Intel PT), and Performance Monitoring. For both Intel PT and LBRs, UIRET is recorded in precisely the same manner as IRET. Hence for LBRs, UIRETs fall into the OTHER_BRANCH category, which implies that IA32_LBR_CTL.OTHER_BRANCH[bit 22] must be set to record user-interrupt delivery, and that the IA32_LBR_x_INFO.BR_TYPE field will indicate OTHER_BRANCH for any recorded user interrupt. For Intel PT, control flow tracing must be enabled by setting IA32_RTIT_CTL.BranchEn[bit 13].

UIRET will also increment performance counters for which counting BR_INST RETIRED.FAR_BRANCH is enabled.

**Operation**

- Pop tempRIP;
- Pop tempRFLAGS; // see below for how this is used to load RFLAGS
- Pop tempRSP;
- IF tempRIP is not canonical in current paging mode
  THEN #GP(0);
- FI;
- IF ShadowStackEnabled(CPL)
  THEN
    PopShadowStack SSRIP;
    IF SSRIP ≠ tempRIP
      THEN #CP (FAR-RET/IRET);
    FI;
- FI;
- RIP := tempRIP;
- // update in RFLAGS only CF, PF, AF, ZF, SF, TF, DF, OF, NT, RF, AC, and ID
- RFLAGS := (RFLAGS & ~254DD5H) | (tempRFLAGS & 254DD5H);
- RSP := tempRSP;
- UIF := 1;
- Clear any cache-line monitoring established by MONITOR or UMONITOR;

**Flags Affected**

See the Operation section.

**Protected Mode Exceptions**

#UD The UIRET instruction is not recognized in protected mode.
Real-Address Mode Exceptions

#UD The UIRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The UIRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The UIRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0) If the return instruction pointer is non-canonical.
#SS(0) If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#CP If return instruction pointer from stack and shadow stack do not match.
#UD If the LOCK prefix is used.
If executed inside an enclave.
If CR4.UINTR = 0.
If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.
UMONITOR—User Level Set Up Monitor Address

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F AE /6</td>
<td>A</td>
<td>V/V</td>
<td>WAITPKG</td>
<td>Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is contained in r16/r32/r64.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM/r/m(r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

1. The Mod field of the ModR/M byte must have value 11B.

Description

The UMONITOR instruction arms address monitoring hardware using an address specified in the source register (the address range that the monitoring hardware checks for store operations can be determined by using the CPUID monitor leaf function, EAX=05H). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by UMWAIT.

The content of the source register is an effective address. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used. The address range must use memory of the write-back type. Only write-back memory is guaranteed to correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 9, “Multiple-Processor Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

The UMONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, UMONITOR sets the A-bit but not the D-bit in page tables.

UMONITOR and UMWAIT are available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. UMONITOR and UMWAIT may be executed at any privilege level. Except for the width of the source register, the instruction’s operation is the same in non-64-bit modes and in 64-bit mode.

UMONITOR does not interoperate with the legacy MWAIT instruction. If UMONITOR was executed prior to executing MWAIT and following the most recent execution of the legacy MONITOR instruction, MWAIT will not enter an optimized state. Execution will continue to the instruction following MWAIT.

The UMONITOR instruction causes a transactional abort when used inside a transactional region.

The width of the source register (16b, 32b or 64b) is determined by the effective addressing width, which is affected in the standard way by the machine mode settings and 67 prefix.

Operation

UMONITOR sets up an address range for the monitor hardware using the content of source register as an effective address and puts the monitor hardware in armed state. A store to the specified address range will trigger the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

UMONITOR void _umonitor(void *address);

Numeric Exceptions

None.

1. The Mod field of the ModR/M byte must have value 11B.
**Protected Mode Exceptions**

#GP(0) If the specified segment is not SS and the source register is outside the specified segment limit.

If the specified segment register contains a NULL segment selector.

#SS(0) If the specified segment is SS and the source register is outside the SS segment limit.

#PF(fault-code) For a page fault.

#UD If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

**Real Address Mode Exceptions**

#GP If the specified segment is not SS and the source register is outside of the effective address space from 0 to FFFFH.

#SS If the specified segment is SS and the source register is outside of the effective address space from 0 to FFFFH.

#UD If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

**Virtual 8086 Mode Exceptions**

Same exceptions as in real address mode; additionally:

#PF(fault-code) For a page fault.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0) If the specified segment is not SS and the linear address is in non-canonical form.

#SS(0) If the specified segment is SS and the source register is in non-canonical form.

#PF(fault-code) For a page fault.

#UD If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
UMWAIT—User Level Monitor Wait

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F AE /6</td>
<td>A</td>
<td>V/V</td>
<td>WAITPKG</td>
<td>A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

UMWAIT instructs the processor to enter an implementation-dependent optimized state while monitoring a range of addresses. The optimized state may be either a light-weight power/performance optimized state or an improved power/performance optimized state. The selection between the two states is governed by the explicit input register bit[0] source operand.

UMWAIT is available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. UMWAIT may be executed at any privilege level. This instruction’s operation is the same in non-64-bit modes and in 64-bit mode.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if nonzero.

<table>
<thead>
<tr>
<th>Bit Value</th>
<th>State Name</th>
<th>Wakeup Time</th>
<th>Power Savings</th>
<th>Other Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit[0] = 0</td>
<td>C0.2</td>
<td>Slower</td>
<td>Larger</td>
<td>Improves performance of the other SMT thread(s) on the same core.</td>
</tr>
<tr>
<td>bit[0] = 1</td>
<td>C0.1</td>
<td>Faster</td>
<td>Smaller</td>
<td>N/A</td>
</tr>
<tr>
<td>bits[31:1]</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The instruction wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value (if the monitoring hardware did not trigger beforehand).

Prior to executing the UMWAIT instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of “1” means all C0.2 requests revert to C0.1.

If the processor that executed a UMWAIT instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The UMWAIT instruction causes a transactional abort when used inside a transactional region.

The UMWAIT instruction operates with the UMONITOR instruction. The two instructions allow the definition of an address at which to wait (UMONITOR) and an implementation-dependent optimized operation to perform while waiting (UMWAIT). The execution of UMWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by UMONITOR. The UMWAIT instruction will not wait (will not enter an implementation-dependent optimized state) if any of the

1. The Mod field of the ModR/M byte must have value 11B.
following instructions were executed before UMWAIT and after the most recent execution of UMONITOR: IRET, MONITOR, SYSEXIT, SYSRET, and far RET (the last if it is changing CPL).

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the address range armed by the UMONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0).

Following exit from the implementation-dependent-optimized state, control passes to the instruction after the UMWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction.

Unlike the HLT instruction, the UMWAIT instruction does not restart at the UMWAIT instruction following the handling of an SMI.

If the preceding UMONITOR instruction did not successfully arm an address range or if UMONITOR was not executed prior to executing UMWAIT and following the most recent execution of the legacy MONITOR instruction (UMWAIT does not interoperate with MONITOR), then the processor will not enter an optimized state. Execution will continue to the instruction following UMWAIT.

A store to the address range armed by the UMONITOR instruction will cause the processor to exit UMWAIT if either the store was originated by other processor agents or the store was originated by a non-processor agent.

Operation

\[
\begin{align*}
os\text{\_deadline} & := TSC+\text{IA32\_UMWAIT\_CONTROL}[31:2]\ll 2 \\
\text{instr\_deadline} & := \text{UINT64}(EDX:EAX) \\
\begin{cases} 
\text{IF os\_deadline < instr\_deadline:} & \\
\qquad \text{deadline} := \text{os\_deadline} \\
\text{ELSE:} & \\
\qquad \text{deadline} := \text{instr\_deadline} \\
\end{cases} \\
\text{WHILE monitor hardware armed AND TSC < deadline:} \\
\qquad \text{implementation\_dependent\_optimized\_state(Source register, deadline, IA32\_UMWAIT\_CONTROL[0])} \\
\end{align*}
\]

\[
\begin{align*}
\text{IF using\_os\_deadline AND TSC} & \geq \text{deadline:} \\
\qquad \text{RFLAGS.CF} & := 1 \\
\text{ELSE:} & \\
\qquad \text{RFLAGS.CF} & := 0 \\
\text{RFLAGS.AF,PF,SF,ZF,OF} & := 0 \\
\end{align*}
\]

Intel C/C++ Compiler Intrinsic Equivalent

UMWAIT uint8_t _umwait(uint32_t control, uint64_t counter);

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#GP(0) If src[31:1] != 0. If CR4.TSD = 1 and CPL != 0.

#UD If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 15 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpacks and Interleaves double precision floating-point values from high quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F.W1G 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.256.66.0F.W1G 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 15 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 15 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 15 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpacks and Interleaves double precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM{reg (r, w)}</td>
<td>ModRM{r/m (r)}</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM{reg (w)}</td>
<td>VEX.vvvv (r)</td>
<td>ModRM{r/m (r)}</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM{reg (w)}</td>
<td>VEX.vvvv (r)</td>
<td>ModRM{r/m (r)}</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs an interleaved unpack of the high double precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.
EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPD (EVEX Encoded Versions When SRC2 is a Register)

(\(KL, VL\)) = (2, 128), (4, 256), (8, 512)

IF \(VL >= 128\)
   \(TMP_{DEST}[63:0] := SRC1[127:64]\)
   \(TMP_{DEST}[127:64] := SRC2[127:64]\)
FI;

IF \(VL >= 256\)
   \(TMP_{DEST}[191:128] := SRC1[255:192]\)
   \(TMP_{DEST}[255:192] := SRC2[255:192]\)
FI;

IF \(VL >= 512\)
   \(TMP_{DEST}[319:256] := SRC1[383:320]\)
   \(TMP_{DEST}[383:320] := SRC2[383:320]\)
   \(TMP_{DEST}[447:384] := SRC1[511:448]\)
   \(TMP_{DEST}[511:448] := SRC2[511:448]\)
FI;

FOR \(j := 0\) TO \(KL-1\)
   \(i := j * 64\)
   IF \(k1[j] \text{ OR *no writemask*}\)
      THEN \(DEST[i+63:i] := TMP_{DEST}[i+63:i]\)
   ELSE
      IF \text{merging-masking} \ ; \text{merging-masking}
         THEN \(*DEST[i+63:i]\) remains unchanged* \n      ELSE \text{zeroing-masking} \ ; \text{zeroing-masking}
         \(DEST[i+63:i] := 0\)
      FI
   FI;
ENDFOR

\(DEST[\text{MAXVL}-1:VL] := 0\)
VUNPCKHPD (EVEX Encoded Version When SRC2 is Memory)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;

IF VL >= 128
    TMP_DEST[63:0] := SRC1[127:64]
    TMP_DEST[127:64] := TMP_SRC2[127:64]
FI;

IF VL >= 256
FI;

IF VL >= 512
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VUNPCKHPD (VEX.256 Encoded Version)

DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:256] := 0

VUNPCKHPD (VEX.128 Encoded Version)

DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] := 0

UNPCKHPD (128-bit Legacy SSE Version)

DEST[63:0] := SRC1[127:64]
DEST[127:64] := SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKHPD __m512d _mm512_unpackhi_pd(__m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b);
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-50, “Type E4NF Class Exception Conditions.”
UNPCKHPS—Unpack and Interleave High Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Unpacks and Interleaves single precision floating-point values from high quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and Interleaves single precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Unpacks and Interleaves single precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Unpacks and Interleaves single precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX512F</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs an interleaved unpack of the high single precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.
EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

**VUNPCKHPS (EVEX Encoded Version When SRC2 is a Register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL >= 128
  TMP_DEST[31:0] := SRC1[95:64]
  TMP_DEST[63:32] := SRC2[95:64]
  TMP_DEST[95:64] := SRC1[127:96]
FI;

IF VL >= 256
FI;

IF VL >= 512
  TMP_DEST[479:448] := SRC1[511:480]
  TMP_DEST[511:480] := SRC2[511:480]
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VUNPCKHPS (EVEX Encoded Version When SRC2 is Memory)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] := SRC1[95:64]
  TMP_DEST[95:64] := SRC1[127:96]
FI;
IF VL >= 256
FI;
IF VL >= 512
  TMP_DEST[479:448] := SRC1[511:480]
  TMP_DEST[511:480] := TMP_SRC2[511:480]
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
  FI
Fi;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VUNPCKHPS (VEX.256 Encoded Version)**
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[MAXVL-1:128] := 0

**VUNPCKHPS (VEX.128 Encoded Version)**
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[MAXVL-1:128] := 0

**UNPCKHPS (128-bit Legacy SSE Version)**
DEST[31:0] := SRC1[95:64]
DEST[63:32] := SRC2[95:64]
DEST[95:64] := SRC1[127:96]
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**
VUNPCKHPS __m512 _mm512_unpackhi_ps(__m512 a, __m512 b);
VUNPCKHPS __m512 __m512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 __m512_maskz_unpackhi_ps(__mmask16 k, __m128 a, __m128 b);
VUNPCKHPS __m256 __m256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 __m256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 __m256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 __mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 __mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 __mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-50, “Type E4NF Class Exception Conditions.”
UNPCKLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 14 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Unpacks and Interleaves double precision floating-point values from low quadwords of xmm1 and xmm2/m128.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs an interleaved unpack of the low double precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.
**Operation**

**VUNPCKLPD (EVEX Encoded Versions When SRC2 is a Register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL >= 128
    TMP_DEST[63:0] := SRC1[63:0]
    TMP_DEST[127:64] := SRC2[63:0]
    FI;

IF VL >= 256
    FI;

IF VL >= 512
    FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI
ENDFOR

DEST[MAXVL-1:VL] := 0
VUNPCKLPD (EVEX Encoded Version When SRC2 is Memory)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+63:i] := SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
  FI;
ENDFOR;

IF VL >= 128
  TMP_DEST[63:0] := SRC1[63:0]
  TMP_DEST[127:64] := TMP_SRC2[63:0]
FI;

IF VL >= 256
FI;

IF VL >= 512
FI;

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] := 0
      FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VUNPCKLPD (VEX.256 Encoded Version)

DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:256] := 0

VUNPCKLPD (VEX.128 Encoded Version)

DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] := 0

UNPCKLPD (128-bit Legacy SSE Version)

DEST[63:0] := SRC1[63:0]
DEST[127:64] := SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPD __m512d _mm512_unpacklo_pd(__m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b);
VUNPCKLPD __m256d __mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d __mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m128d __mm128d unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d __mm128d_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-50, "Type E4NF Class Exception Conditions."
UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values

**Description**

Performs an interleaved unpack of the low single precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.
EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is an XMM register, conditionally updated using writemask k1.

**Operation**

**VUNPCKLPS (EVEX Encoded Version When SRC2 is a ZMM Register)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

- **EVEX.512 encoded version**
  - \( KL \geq 128 \)
    - \( \text{TMP}_\text{DEST}[31:0] := \text{SRC}_1[31:0] \)
    - \( \text{TMP}_\text{DEST}[63:32] := \text{SRC}_2[31:0] \)
    - \( \text{TMP}_\text{DEST}[95:64] := \text{SRC}_1[63:32] \)
    - \( \text{TMP}_\text{DEST}[127:96] := \text{SRC}_2[63:32] \)
  
  - **If** \( VL \geq 256 \)
    - \( \text{TMP}_\text{DEST}[159:128] := \text{SRC}_1[159:128] \)
    - \( \text{TMP}_\text{DEST}[191:160] := \text{SRC}_2[159:128] \)
    - \( \text{TMP}_\text{DEST}[223:192] := \text{SRC}_1[191:160] \)
  
  - **If** \( VL \geq 512 \)
    - \( \text{TMP}_\text{DEST}[287:256] := \text{SRC}_1[287:256] \)
    - \( \text{TMP}_\text{DEST}[319:288] := \text{SRC}_2[287:256] \)
    - \( \text{TMP}_\text{DEST}[351:320] := \text{SRC}_1[319:288] \)
    - \( \text{TMP}_\text{DEST}[383:352] := \text{SRC}_2[319:288] \)
    - \( \text{TMP}_\text{DEST}[415:384] := \text{SRC}_1[415:384] \)
    - \( \text{TMP}_\text{DEST}[447:416] := \text{SRC}_2[415:384] \)
    - \( \text{TMP}_\text{DEST}[479:448] := \text{SRC}_1[447:416] \)
    - \( \text{TMP}_\text{DEST}[511:480] := \text{SRC}_2[447:416] \)
  
  - **For** \( j := 0 \) **to** \( KL-1 \)

---

**Figure 4-28. VUNPCKLPS Operation**
UNPCKLPS—Unpack and Interleave Low Packed Single Precision Floating-Point Values

\[
i := j \times 32 \\
\text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\quad \text{THEN } \text{DEST}[i+31:i] := \text{TMP\_DEST}[i+31:i] \\
\quad \text{ELSE} \\
\quad \quad \text{IF } \text{*merging-masking*} \\
\quad \quad \quad \text{THEN } \text{*DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE } \text{*zeroing-masking*} \\
\quad \quad \quad \quad \text{DEST}[i+31:i] := 0 \\
\quad \text{FI} \\
\text{FI} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

**VUNPCKLPS (EVEX Encoded Version When SRC2 is Memory)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0\) TO \(KL-1\)
\[
i := j \times 31 \\
\text{IF } (\text{EVEX.b} = 1) \\
\quad \text{THEN } \text{TMP\_SRC2}[i+31:i] := \text{SRC2}[31:0] \\
\quad \text{ELSE } \text{TMP\_SRC2}[i+31:i] := \text{SRC2}[i+31:i] \\
\quad \text{FI} \\
\text{ENDFOR;}
\]

IF \(VL \geq 128\)
\[
\text{TMP\_DEST}[31:0] := \text{SRC1}[31:0] \\
\text{TMP\_DEST}[63:32] := \text{TMP\_SRC2}[31:0] \\
\text{TMP\_DEST}[95:64] := \text{SRC1}[63:32] \\
\text{TMP\_DEST}[127:96] := \text{TMP\_SRC2}[63:32] \\
\text{FI};
\]

IF \(VL \geq 256\)
\[
\text{TMP\_DEST}[159:128] := \text{SRC1}[159:128] \\
\text{TMP\_DEST}[223:192] := \text{SRC1}[191:160] \\
\text{FI};
\]

IF \(VL \geq 512\)
\[
\text{TMP\_DEST}[287:256] := \text{SRC1}[287:256] \\
\text{TMP\_DEST}[319:288] := \text{TMP\_SRC2}[287:256] \\
\text{TMP\_DEST}[351:320] := \text{SRC1}[319:288] \\
\text{TMP\_DEST}[383:352] := \text{TMP\_SRC2}[319:288] \\
\text{TMP\_DEST}[415:384] := \text{SRC1}[415:384] \\
\text{TMP\_DEST}[479:448] := \text{SRC1}[447:416] \\
\text{TMP\_DEST}[511:480] := \text{TMP\_SRC2}[447:416] \\
\text{FI};
\]

FOR \(j := 0\) TO \(KL-1\)
\[
i := j \times 32 \\
\text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\quad \text{THEN } \text{DEST}[i+31:i] := \text{TMP\_DEST}[i+31:i] \\
\quad \text{ELSE} \\
\quad \quad \text{IF } \text{*merging-masking*} \\
\quad \quad \quad \text{THEN } \text{*DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE } \text{*zeroing-masking*} \\
\quad \quad \quad \quad \text{DEST}[i+31:i] := 0 \\
\quad \text{FI}
\]

\[4-740\]
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

UNPCKLPS (VEX.256 Encoded Version)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[MAXVL-1:128] := 0

VUNPCKLPS (VEX.128 Encoded Version)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[MAXVL-1:128] := 0

UNPCKLPS (128-bit Legacy SSE Version)
DEST[31:0] := SRC1[31:0]
DEST[63:32] := SRC2[31:0]
DEST[95:64] := SRC1[63:32]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskunpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-50, “Type E4NF Class Exception Conditions.”
5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-11 for the column index value between 0:7, followed by Table 5-12 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a short-hand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one or more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator ‘not’ is expressed using ‘!’: Additionally, the conditional expression “A?B:C” expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g., andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g., andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g., norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g., zeros or norCB.

The 3-input expression “majorABC” returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression “minorABC” returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-11 and Table 5-12 include:

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.
Table 5-11. Lower 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>[3:0]</th>
<th>[7:4]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>00H</td>
<td>FALSE</td>
<td>andAnorBC</td>
</tr>
<tr>
<td></td>
<td>norBnandAC</td>
<td>andAIB</td>
</tr>
<tr>
<td></td>
<td>norCnandBA</td>
<td>andAIC</td>
</tr>
<tr>
<td></td>
<td>andAxorBC</td>
<td>andAnandBC</td>
</tr>
<tr>
<td>01H</td>
<td>norABC</td>
<td>norCB</td>
</tr>
<tr>
<td></td>
<td>norBxorAC</td>
<td>A?IB:norBC</td>
</tr>
<tr>
<td></td>
<td>norCxorBA</td>
<td>A?IC:norBC</td>
</tr>
<tr>
<td>02H</td>
<td>andCnorBA</td>
<td>norBxnorAC</td>
</tr>
<tr>
<td></td>
<td>andCIB</td>
<td>norBnorAC</td>
</tr>
<tr>
<td></td>
<td>C?norBA:andBA</td>
<td>C?norBA:A</td>
</tr>
<tr>
<td></td>
<td>C?IB:andBA</td>
<td>C?IB:A</td>
</tr>
<tr>
<td>03H</td>
<td>norBA</td>
<td>norBandAC</td>
</tr>
<tr>
<td></td>
<td>C?IB:norBA</td>
<td>IB</td>
</tr>
<tr>
<td></td>
<td>C?norBA:xnorBA</td>
<td>A?IC:IB</td>
</tr>
<tr>
<td>04H</td>
<td>andBnorAC</td>
<td>norCxnorBA</td>
</tr>
<tr>
<td></td>
<td>B?norAC:andAC</td>
<td>B?norACA:andAIC</td>
</tr>
<tr>
<td></td>
<td>andBIC</td>
<td>norCnorBA</td>
</tr>
<tr>
<td></td>
<td>B?CandAC</td>
<td>B?ICA</td>
</tr>
<tr>
<td>05H</td>
<td>norCA</td>
<td>norCanorBA</td>
</tr>
<tr>
<td></td>
<td>B?norAC:xnorAC</td>
<td>A?IB:C</td>
</tr>
<tr>
<td></td>
<td>B?CnorAC</td>
<td>IC</td>
</tr>
<tr>
<td>06H</td>
<td>norAxnorBC</td>
<td>A?norBC:xorBC</td>
</tr>
<tr>
<td></td>
<td>B?norAC:C</td>
<td>xorBorAC</td>
</tr>
<tr>
<td></td>
<td>C?norBA:B</td>
<td>xorCorBA</td>
</tr>
<tr>
<td></td>
<td>xorCB</td>
<td>B?CxorAC</td>
</tr>
<tr>
<td>07H</td>
<td>norAandBC</td>
<td>minorABC</td>
</tr>
<tr>
<td></td>
<td>C?IB:IA</td>
<td>nandBorAC</td>
</tr>
<tr>
<td></td>
<td>B?C:JA</td>
<td>nandCorBA</td>
</tr>
<tr>
<td></td>
<td>A?xorBC:nandBC</td>
<td>nandCB</td>
</tr>
<tr>
<td>08H</td>
<td>norAnandBC</td>
<td>A?norBC:andBC</td>
</tr>
<tr>
<td></td>
<td>A?IB:andBC</td>
<td>andBxorAC</td>
</tr>
<tr>
<td></td>
<td>A?IC:andBC</td>
<td>A?xorBC:andBC</td>
</tr>
<tr>
<td></td>
<td>xorAandBC</td>
<td></td>
</tr>
<tr>
<td>09H</td>
<td>norAxorBC</td>
<td>A?norBC:xnorBC</td>
</tr>
<tr>
<td></td>
<td>C?xorBA:norBA</td>
<td>A?IB:xnorBC</td>
</tr>
<tr>
<td></td>
<td>B?xorAC:norAC</td>
<td>A?IC:xnorBC</td>
</tr>
<tr>
<td></td>
<td>xnorABC</td>
<td>A?nandBC:xnorBC</td>
</tr>
<tr>
<td>0AH</td>
<td>andCIA</td>
<td>A?norBC:C</td>
</tr>
<tr>
<td></td>
<td>andCNandBA</td>
<td>A?IC:andBA</td>
</tr>
<tr>
<td></td>
<td>xorCA</td>
<td>xorCanorBA</td>
</tr>
<tr>
<td>0BH</td>
<td>C?IA:norBA</td>
<td>C?IA:IB</td>
</tr>
<tr>
<td></td>
<td>C?nandBA:norBA</td>
<td>C?nandBA:IB</td>
</tr>
<tr>
<td></td>
<td>C?nandBA:xnorBA</td>
<td>C?nandBA:xnorAC</td>
</tr>
<tr>
<td>0CH</td>
<td>andBIA</td>
<td>A?norBC:B</td>
</tr>
<tr>
<td></td>
<td>B?IA:andAC</td>
<td>xorBA</td>
</tr>
<tr>
<td></td>
<td>andBnandAC</td>
<td>A?IC:B</td>
</tr>
<tr>
<td></td>
<td>xorBandAC</td>
<td>A?nandBC:B</td>
</tr>
<tr>
<td>0DH</td>
<td>B?IA:norAC</td>
<td>B?IA:xnorAC</td>
</tr>
<tr>
<td></td>
<td>B?IA:orAC</td>
<td>C?xorBA:norAC</td>
</tr>
<tr>
<td></td>
<td>B?nandAC:norAC</td>
<td>B?nandAC:orAC</td>
</tr>
<tr>
<td>0EH</td>
<td>norAnorBC</td>
<td>xorAorBC</td>
</tr>
<tr>
<td></td>
<td>B?IA:C</td>
<td>A?IB:orBC</td>
</tr>
<tr>
<td></td>
<td>A?IA:B</td>
<td>A?IC:orBC</td>
</tr>
<tr>
<td></td>
<td>B?nandAC:orBC</td>
<td></td>
</tr>
<tr>
<td>0FH</td>
<td>IA</td>
<td>nandAorBC</td>
</tr>
<tr>
<td></td>
<td>C?nandBA:IA</td>
<td>nandBA</td>
</tr>
<tr>
<td></td>
<td>B?nandAC:IA</td>
<td>nandCA</td>
</tr>
<tr>
<td></td>
<td>nandAorBC</td>
<td>nandABC</td>
</tr>
</tbody>
</table>

Table 5-12 shows the half of 256-entry map corresponding to column index values 8:15.
Table 5-12. Upper 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>[7:4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3:0]</td>
<td>08H</td>
</tr>
<tr>
<td>00H</td>
<td>andABC</td>
</tr>
<tr>
<td>07H</td>
<td>xnORAandBC</td>
</tr>
<tr>
<td>08H</td>
<td>andCB</td>
</tr>
<tr>
<td>09H</td>
<td>B?CnORAC</td>
</tr>
<tr>
<td>0BH</td>
<td>B?CIA</td>
</tr>
<tr>
<td>0CH</td>
<td>A?anDcBCB</td>
</tr>
<tr>
<td>0DH</td>
<td>C?BIA</td>
</tr>
<tr>
<td>0EH</td>
<td>A?anDcB CorBC</td>
</tr>
<tr>
<td>0FH</td>
<td>nandAnandBC</td>
</tr>
</tbody>
</table>

Table 5-11 and Table 5-12 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (FOH, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result_imm8 = Table_Lookup_Entry( 0FOH, 0CCH, 0AAH)

Table_Lookup_Entry is the Boolean expression defined in Table 5-11 and Table 5-12.
5.2 INSTRUCTIONS (V)

VADDPH—Add Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5,W0 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Add packed FP16 value from xmm3/m128/m16bcst to xmm2, and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VADDPH xmm1[k1]{z}, xmm2,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmm3/m128/m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5,W0 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Add packed FP16 value from ymm3/m256/m16bcst to ymm2, and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VADDPH ymm1[k1]{z}, ymm2,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ymm3/m256/m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5,W0 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Add packed FP16 value from zmm3/m512/m16bcst to zmm2, and store result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VADDPH zmm1[k1]{z}, zmm2,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zmm3/m512/m16bcst {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction adds packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

**Operation**

**VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Register**

VL = 128, 256 or 512
KL := VL/16
IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSEIF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
    DEST[MAXVL-1:VL] := 0

**VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source**

VL = 128, 256 or 512
KL := VL/16
FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[0]
        ELSE:
            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]
    ELSEIF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
    DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VADDPH __m128h _mm_add_ph (__m128h a, __m128h b);
VADDPH __m128h _mm_mask_add_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDPH __m128h _mm_maskz_add_ph (__mmask8 k, __m128h a, __m128h b);
VADDPH __m256h _mm256_add_ph (__m256h a, __m256h b);
VADDPH __m256h _mm256_mask_add_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VADDPH __m256h _mm256_maskz_add_ph (__mmask16 k, __m256h a, __m256h b);
VADDPH __m512h _mm512_add_ph (__m512h a, __m512h b);
VADDPH __m512h _mm512_mask_add_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_maskz_add_ph (__mmask32 k, __m512h a, __m512h b);
VADDPH __m512h _mm512_add_round_ph (__m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_mask_add_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VADDPH __m512h _mm512_maskz_add_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions
Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
# VADDSH—Add Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 58 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Add the low FP16 value from xmm3/m16 to xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

## Description

This instruction adds the low FP16 value from the source operands and stores the FP16 result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

## Operation

**VADDSH (EVEX Encoded Versions)**

IF EVEX.b = 1 and SRC2 is a register:
- SET_RM(EVEX.RC)
ELSE
- SET_RM(MXCSR.RC)
IF k1[0] OR *no writemask*:
- DEST.fp16[0]: = SRC1.fp16[0] + SRC2.fp16[0]
ELSEIF *zeroing*:
- DEST.fp16[0]: = 0
// else dest.fp16[0] remains unchanged
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDSH __m128h __mm_add_round_sh (__m128h a, __m128h b, int rounding);
VADDSH __m128h __mm_mask_add_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH __m128h __mm_maskz_add_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VADDSH __m128h __mm_add_sh (__m128h a, __m128h b);
VADDSH __m128h __mm_mask_add_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VADDSH __m128h __mm_maskz_add_sh (__mmask8 k, __m128h a, __m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.wwww (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).
Operation

**VALIGND (EVEX Encoded Versions)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

IF (SRC2 *is memory*) (AND EVEX.b = 1)
  THEN
    FOR \(j := 0\) TO \(KL-1\)
      \(i := j \times 32\)
      src[\(i+31:i\)] := SRC2[31:0]
    ENDFOR;
  ELSE src := SRC2
  FI
  ; Concatenate sources
  tmp[VL-1:0] := src[VL-1:0]
  tmp[2VL-1:VL] := SRC1[VL-1:0]
  ; Shift right doubleword elements
  IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
      IF VL = 256
        THEN SHIFT = imm8[2:0]
        ELSE SHIFT = imm8[3:0]
      FI
    FI;
  Fi;
  tmp[2VL-1:0] := tmp[2VL-1:0] >> (32*SHIFT)
  ; Apply writemask
  FOR \(j := 0\) TO \(KL-1\)
    \(i := j \times 32\)
    IF k1[\(j\)] OR *no writemask*
      THEN DEST[\(i+31:i\)] := tmp[\(i+31:i\)]
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[\(i+31:i\)] remains unchanged*
        ELSE ; zeroing-masking
          DEST[\(i+31:i\)] := 0
        FI
      FI
  ; Concatenate sources
  tmp[VL-1:0] := src[VL-1:0]
  tmp[2VL-1:VL] := SRC1[VL-1:0]
  ; Shift right quadword elements
  IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
      FOR \(j := 0\) TO \(KL-1\)
        \(i := j \times 64\)
        src[\(i+63:i\)] := SRC2[63:0]
      ENDFOR;
  ELSE src := SRC2
  FI
  ; Concatenate sources
  tmp[VL-1:0] := src[VL-1:0]
  tmp[2VL-1:VL] := SRC1[VL-1:0]
IF VL = 128
    THEN SHIFT = imm8[0]
ELSE
    IF VL = 256
        THEN SHIFT = imm8[1:0]
        ELSE SHIFT = imm8[2:0]
    FI
FI;

tmp[2VL-1:0] := tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := tmp[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VALIGND __m512i _mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i _mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i _mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i _mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);

Exceptions
See Table 2-50, "Type E4NF Class Exception Conditions."
VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

**Description**
Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

**Operation**
VBLENDMPD (EVEX Encoded Versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no controlmask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:j] := SRC2[63:0]
        ELSE
          DEST[i+63:j] := SRC2[i+63:j]
VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

INSTRUCTION SET REFERENCE, V

FI;
ELSE
IF *merging-masking* ; merging-masking
THEN DEST[i+63:i] := SRC1[i+63:i]
ELSE ; zeroing-masking
    DEST[i+63:i] := 0
FI;
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VBLENDMPS (EVEX Encoded Versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC2[31:0]
                ELSE
                    DEST[i+31:i] := SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN DEST[i+31:i] := SRC1[i+31:i]
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI;
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 __m128_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, "Type E4 Class Exception Conditions."
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 18 / r VBROADCASTSS xmm1, m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast single precision floating-point element in mem to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 18 / r VBROADCASTSS ymm1, m32</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast single precision floating-point element in mem to eight locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 19 / r VBROADCASTSD ymm1, m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast double precision floating-point element in mem to four locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 1A / r VBROADCASTF128 ymm1, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 18 / r VBROADCAST SS xmm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast the low single precision floating-point element in the source operand to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 18 / r VBROADCASTSS ymm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast low single precision floating-point element in the source operand to eight locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 19 / r VBROADCASTSD ymm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low double precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 19 / r VBROADCASTSD zmm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low double precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 19 / r VBROADCASTF32X2 ymm1 (k1)[z], xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast two single precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 19 / r VBROADCASTF32X2 zmm1 (k1)[z], xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast two single precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 18 / r VBROADCASTSS xmm1 (k1)[z], xmm2/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low single precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 18 / r VBROADCASTSS ymm1 (k1)[z], xmm2/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low single precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 18 / r VBROADCASTSS zmm1 (k1)[z], xmm2/m32</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low single precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 1A / r VBROADCASTF32X4 ymm1 (k1)[z], m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast 128 bits of 4 single precision floating-point data in mem to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 1A / r VBROADCASTF32X4 zmm1 (k1)[z], m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 128 bits of 4 single precision floating-point data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 1A / r VBROADCASTF64X2 ymm1 (k1)[z], m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast 128 bits of 2 double precision floating-point data in mem to locations in ymm1 using writemask k1.</td>
</tr>
</tbody>
</table>
### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

**VBBROADCASTSD**/**VBBROADCASTSS**/**VBBROADCASTF128** load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

**VEX256-encoded versions:** The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

**EVEX-encoded versions:** The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

**VBBROADCASTF32X2/VBBROADCASTF32X4/VBBROADCASTF64X2/VBBROADCASTF32X8/VBBROADCASTF64X4** load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

**VBBROADCASTSD** and **VBBROADCASTF128**, F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. **VBBROADCASTSS** is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

**VBBROADCASTF32X2/VBBROADCASTF32X4/VBBROADCASTF32X8** have 32-bit granularity. **VBBROADCASTF64X2** and **VBBROADCASTF64X4** have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If **VBBROADCASTSD** or **VBBROADCASTF128** is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)
Operation

**VBROADCASTSS (128-bit Version VEX and Legacy)**

\[ \text{temp} := \text{SRC}[31:0] \]
\[ \text{DEST}[31:0] := \text{temp} \]
\[ \text{DEST}[63:32] := \text{temp} \]
\[ \text{DEST}[95:64] := \text{temp} \]
\[ \text{DEST}[127:96] := \text{temp} \]
\[ \text{DEST}[^{\text{MAXVL-1-128}}] := 0 \]

**VBROADCASTSS (VEX.256 Encoded Version)**

\[ \text{temp} := \text{SRC}[31:0] \]
\[ \text{DEST}[31:0] := \text{temp} \]
\[ \text{DEST}[63:32] := \text{temp} \]
\[ \text{DEST}[95:64] := \text{temp} \]
\[ \text{DEST}[127:96] := \text{temp} \]
\[ \text{DEST}[159:128] := \text{temp} \]
\[ \text{DEST}[191:160] := \text{temp} \]
\[ \text{DEST}[223:192] := \text{temp} \]
\[ \text{DEST}[255:224] := \text{temp} \]
\[ \text{DEST}[^{\text{MAXVL-1-256}}] := 0 \]

**VBROADCASTSS (EVEX Encoded Versions)**

\( (KL, VL) = (4, 128), (8, 256), (16, 512) \)

\text{FOR } j := 0 \text{ TO } KL-1 \text{ DO}

\[ i := j \times 32 \]
\[ \text{IF } k1[j] \text{ OR *no writemask*}
\]
\[ \text{THEN } \text{DEST}[i+31:i] := \text{SRC}[31:0] \]
\[ \text{ELSE}
\]
\[ \text{IF *merging-masking*} \]
\[ \text{THEN } \*\text{DEST}[i+31:i] \text{ remains unchanged*} \]
\[ \text{ELSE} \]
\[ \text{DEST}[i+31:i] := 0 \]
\[ \text{FI} \]
\[ \text{FI} \]
\text{ENDDO}
\[ \text{DEST}[^{\text{MAXVL-1-VL}}] := 0 \]
VBROADCAST (VEX.256 Encoded Version)

\[ temp := \text{SRC}[63:0] \]
\[ \text{DEST}[63:0] := temp \]
\[ \text{DEST}[127:64] := temp \]
\[ \text{DEST}[191:128] := temp \]
\[ \text{DEST}[255:192] := temp \]
\[ \text{DEST}[\text{MAXVL}-1:256] := 0 \]

VBROADCAST (EVEX Encoded Versions)

\( (KL, VL) = (4, 256), (8, 512) \)

\[ \text{FOR} j := 0 \text{ TO } KL-1 \]
\[ i := j \times 64 \]
\[ \text{IF} k1[j] \text{ OR } \text{*no writemask*} \]
\[ \text{THEN} \text{DEST}[i+63:i] := \text{SRC}[63:0] \]
\[ \text{ELSE} \]
\[ \text{IF } \text{merging-masking} \]
\[ \text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged} \]
\[ \text{ELSE} \]
\[ \text{DEST}[i+63:i] := 0 \]
\[ \text{FI} \]
\[ \text{FI} \]
\[ \text{ENDFOR} \]
\[ \text{DEST}[\text{MAXVL}-1:VL] := 0 \]

VBROADCASTF32x2 (EVEX Encoded Versions)

\( (KL, VL) = (8, 256), (16, 512) \)

\[ \text{FOR} j := 0 \text{ TO } KL-1 \]
\[ i := j \times 32 \]
\[ n := (j \mod 2) \times 32 \]
\[ \text{IF} k1[j] \text{ OR } \text{*no writemask*} \]
\[ \text{THEN} \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \]
\[ \text{ELSE} \]
\[ \text{IF } \text{merging-masking} \]
\[ \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged} \]
\[ \text{ELSE} \]
\[ \text{DEST}[i+31:i] := 0 \]
\[ \text{FI} \]
\[ \text{FI} \]
\[ \text{ENDFOR} \]
\[ \text{DEST}[\text{MAXVL}-1:VL] := 0 \]

VBROADCASTF128 (VEX.256 Encoded Version)

\[ temp := \text{SRC}[127:0] \]
\[ \text{DEST}[127:0] := temp \]
\[ \text{DEST}[255:128] := temp \]
\[ \text{DEST}[\text{MAXVL}-1:256] := 0 \]


VBROADCASTF32X4 (EVEX Encoded Versions)

(KL, VL) = (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j* 32
  n := (j modulo 4) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VBROADCASTF64X2 (EVEX Encoded Versions)

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  n := (j modulo 2) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[n+63:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
  FI;
ENDFOR;

VBROADCASTF32X8 (EVEX.U1.512 Encoded Version)

FOR j := 0 TO 15
  i := j * 32
  n := (j modulo 8) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
VBroadcastF64x4 (EVEX.512 Encoded Version)

FOR \( j := 0 \) TO 7
i := \( j \times 64 \)
n := \( (j \mod 4) \times 64 \)
IF k1\([j]\) OR *no writemask*
   THEN DEST\([i+63:i]\) := SRC\([n+63:n]\)
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST\([i+63:i]\) remains unchanged*
      ELSE ; zeroing-masking
         DEST\([i+63:i]\) := 0
      FI
   FI;
ENDFOR
DEST\([\text{MAXVL}-1:\text{VL}]\) := 0

Intel C/C++ Compiler Intrinsic Equivalent

VBroadcastF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBroadcastF32x2 __m512 _mm512_masked_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBroadcastF32x2 __m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBroadcastF32x2 __m256 _mm256_broadcast_f32x2(__m128 a);
VBroadcastF32x2 __m256 _mm256_masked_broadcast_f32x2(__m128 s, __mmask8 k, __m128 a);
VBroadcastF32x2 __m256 _mm256_maskz_broadcast_f32x2(__mmask8 k, __m128 a);
VBroadcastF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBroadcastF32x4 __m512 _mm512_masked_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBroadcastF32x4 __m512 _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBroadcastF32x4 __m256 _mm256_broadcast_f32x4(__m128 a);
VBroadcastF32x4 __m256 _mm256_masked_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBroadcastF32x4 __m256 _mm256_maskz_broadcast_f32x4(__mmask8 k, __m128 a);
VBroadcastF64x2 __m512d _mm512d_broadcast_f64x2(__m128d a);
VBroadcastF64x2 __m512d _mm512d_masked_broadcast_f64x2(__m128d s, __mmask8 k, __m128d a);
VBroadcastF64x2 __m512d _mm512d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBroadcastF64x2 __m256d _mm256d_broadcast_f64x2(__m128d a);
VBroadcastF64x2 __m256d _mm256d_masked_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBroadcastF64x2 __m256d _mm256d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBroadcastF64x4 __m512d _mm512d_broadcast_f64x4(__m128d a);
VBroadcastF64x4 __m512d _mm512d_masked_broadcast_f64x4(__m128d s, __mmask8 k, __m256d a);
VBroadcastF64x4 __m512d _mm512d_maskz_broadcast_f64x4(__mmask8 k, __m256d a);
VBroadcastF64x4 __m128d _mm128d_broadcast_f64x4(__m128d a);
VBroadcastF64x4 __m128d _mm128d_masked_broadcast_f64x4(__m128d s, __mmask8 k, __m128d a);
VBroadcastF64x4 __m128d _mm128d_maskz_broadcast_f64x4(__mmask8 k, __m128d a);
VBroadcastSSS __m512 _mm512d_broadcastss_ps(__m128 a);
VBroadcastSSS __m512 _mm512d_masked_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBroadcastSSS __m512 _mm512d_maskz_broadcastss_ps(__mmask16 k, __m128 a);
VBroadcastSSS __m256 _mm256d_broadcastss_ps(__m256 s, __mmask8 k, __m128 a);
VBroadcastSSS __m256 _mm256d_masked_broadcastss_ps(__m256 s, __mmask8 k, __m128 a);
VBroadcastSSS __m256 _mm256d_maskz_broadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m128 __m128_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 __m128_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 __m128_maskz_broadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m128 __m128_broadcast_ss(float *a);
VBROADCASTSS __m256 __m256_broadcast_ss(float *a);
VBROADCASTF128 __m256 __m256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d __m256_broadcast_pd(__m128d * a);

Exceptions

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-53, “Type E6 Class Exception Conditions.”
Additionally:
#UD If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
If EVEX.L’L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROAD-
CASTF64X2.
If EVEX.L’L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.
### VCMPPH—Compare Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.0F3A.W0 C2 /r /ib VCMPPH k1[k2], xmm2, xmm3/ m128/m16bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compare packed FP16 values in xmm3/m128/m16bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.0F3A.W0 C2 /r /ib VCMPPH k1[k2], ymm2, ymm3/ m256/m16bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compare packed FP16 values in ymm3/m256/m16bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.0F3A.W0 C2 /r /ib VCMPPH k1[k2], zmm2, zmm3/ m512/m16bcst [sae], imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compare packed FP16 values in zmm3/m512/m16bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

This instruction compares packed FP16 values from source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on each of the pairs of packed values. The destination elements are updated according to the writemask.

**Operation**

CASE (imm8 & 0x1F) OF
0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC

VCMPPH (EVEX Encoded Versions)
VL = 128, 256 or 512
KL := VL/16
FOR j := 0 TO KL-1:
  IF k2[j] OR *no writemask*:
    IF EVEX.b = 1:
      tsrc2 := SRC2.fp16[0]
    ELSE:
      tsrc2 := SRC2.fp16[j]
    DEST.bit[j] := SRC1.fp16[j] CMP_OPERATOR tsrc2
  ELSE
    DEST.bit[j] := 0
DEST[MAXKL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCMPPH __m128 __m128_mm_cmp_ph_mask (__m128h a, __m128h b, const int imm8);
VCMPPH __m128 __m128_mm_mask_cmp_ph_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);
VCMPPH __m128 __m128_mm256_cmp_ph_mask (__m256h a, __m256h b, const int imm8);
VCMPPH __m128 __m128_mm256_mask_cmp_ph_mask (__mmask16 k1, __m256h a, __m256h b, const int imm8);
VCMPPH __m128 __m128_mm512_cmp_ph_mask (__m512h a, __m512h b, const int imm8);
VCMPPH __m128 __m128_mm512_mask_cmp_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8);
VCMPPH __m128 __m128_mm512_cmp_round_ph_mask (__m512h a, __m512h b, const int imm8, const int sae);
VCMPPH __m128 __m128_mm512_mask_cmp_round_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8, const int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
**VCMPSH—Compare Scalar FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.0F3A.W0 C2 /r /ib VCMPSH k1[k2], xmm2, xmm3/m16 {sae}, imm8</td>
<td>A</td>
<td>V</td>
<td>AVX512-FP16</td>
<td>Compare low FP16 values in xmm3/m16 and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

This instruction compares the FP16 values from the lowest element of the source operands and stores the result in the destination mask operand. The compare predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on the pair of packed FP16 values. The low destination bit is updated according to the writemask. Bits MAXKL-1:1 of the destination operand are zeroed.

**Operation**

CASE (imm8 & 0x1F) OF

0: CMP_OPERATOR := EQ_OQ;
1: CMP_OPERATOR := LT_OS;
2: CMP_OPERATOR := LE_OS;
3: CMP_OPERATOR := UNORD_Q;
4: CMP_OPERATOR := NEQ_UQ;
5: CMP_OPERATOR := NLT_US;
6: CMP_OPERATOR := NLE_US;
7: CMP_OPERATOR := ORD_Q;
8: CMP_OPERATOR := EQ_UQ;
9: CMP_OPERATOR := NGE_US;
10: CMP_OPERATOR := NGT_US;
11: CMP_OPERATOR := FALSE_OQ;
12: CMP_OPERATOR := NEQ_OQ;
13: CMP_OPERATOR := GE_OS;
14: CMP_OPERATOR := GT_OS;
15: CMP_OPERATOR := TRUE_UQ;
16: CMP_OPERATOR := EQ_OS;
17: CMP_OPERATOR := LT_OQ;
18: CMP_OPERATOR := LE_OQ;
19: CMP_OPERATOR := UNORD_S;
20: CMP_OPERATOR := NEQ_US;
21: CMP_OPERATOR := NLT_UQ;
22: CMP_OPERATOR := NLE_UQ;
23: CMP_OPERATOR := ORD_S;
24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
VCMPSH (EVEX Encoded Versions)
IF k2[0] OR *no writemask*:
    DEST.bit[0] := SRC1.fp16[0] CMP_OPERATOR SRC2.fp16[0]
ELSE
    DEST.bit[0] := 0
DEST[MAXL-1:1] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCMPSH __m128h __mm_cmp_round_sh_mask (__m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __m128h __mm_mask_cmp_round_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __m128h __mm_cmp_sh_mask (__m128h a, __m128h b, const int imm8);
VCMPSH __mmask8 __mm_mask_cmp_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-47, ”Type E3 Class Exception Conditions.”
VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/ Instruc tion</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP.MAP5.W0 2F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compare low FP16 values in xmm1 and xmm2/m16, and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Oper and 1</th>
<th>Oper and 2</th>
<th>Oper and 3</th>
<th>Oper and 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VCOMISH instruction differs from the VUCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The VUCOMISH instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCOMISH SRC1, SRC2
RESULT := OrderedCompare(SRC1.fp16[0], SRC2.fp16[0])
IF RESULT is UNORDERED:
    ZF, PF, CF := 1, 1, 1
ELSE IF RESULT is GREATER_THAN:
    ZF, PF, CF := 0, 0, 0
ELSE IF RESULT is LESS_THAN:
    ZF, PF, CF := 0, 0, 1
ELSE: // RESULT is EQUALS
    ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0

Intel C/C++ Compiler Intrinsic Equivalent

| VCOMISH int _mm_comi_round_sh (__m128h a, __m128h b, const int imm8, const int sae); |
| VCOMISH int _mm_comi_sh (__m128h a, __m128h b, const int imm8); |
| VCOMISH int _mm_comieq_sh (__m128h a, __m128h b); |
| VCOMISH int _mm_comige_sh (__m128h a, __m128h b); |
| VCOMISH int _mm_comigt_sh (__m128h a, __m128h b); |
| VCOMISH int _mm_comile_sh (__m128h a, __m128h b); |
| VCOMISH int _mm_comilt_sh (__m128h a, __m128h b); |
| VCOMISH int _mm_comineq_sh (__m128h a, __m128h b); |

SIMD Floating-Point Exceptions

Invalid, Denormal.
Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 [k1][z], xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed double precision floating-point values from xmm2 to xmm1/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 [k1][z], ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed double precision floating-point values from ymm2 to ymm1/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 [k1][z], zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed double precision floating-point values from zmm2 using control mask k1 to zmm1/m512.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Compress (store) up to 8 double precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand) The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPD (EVEX Encoded Versions) Store Form

(KL, VL) = (2, 128), (4, 256), (8, 512)

\[ SIZE := 64 \]

\[ k := 0 \]

\[ \text{FOR } j := 0 \text{ TO } KL-1 \]

\[ i := j \ast 64 \]

\[ \text{IF } k1[j] \text{ OR *no writemask*} \]

\[ \text{THEN} \]

\[ \text{DEST}[k+\text{SIZE} \text{+} i] := \text{SRC}[i+63; i] \]

\[ k := k + \text{SIZE} \]

\[ \text{FI;} \]

\[ \text{ENDFOR} \]
**VCOMPRESSPD (EVEX Encoded Versions) Reg-Reg Form**

KL, VL = (2, 128), (4, 256), (8, 512)

SIZE := 64

k := 0

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN

    DEST[k+SIZE-1:k] := SRC[i+63:i]

    k := k + SIZE

ENDIF

ENDDFOR

IF *merging-masking*

THEN *DEST[VL-1:k] remains unchanged*

ELSE DEST[VL-1:k] := 0

FI

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPD _m512d _mm512_mask_compress_pd(__m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD _m512d _mm512_maskz_compress_pd(__mmask8 k, __m512d a);
VCOMPRESSPD void _mm512_mask_compressedstoreu_pd( void *d, __mmask8 k, __m512d a);
VCOMPRESSPD _m256d _mm256_mask_compress_pd(__m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD _m256d _mm256_maskz_compress_pd(__mmask8 k, __m256d a);
VCOMPRESSPD void _mm256_mask_compressedstoreu_pd( void *d, __mmask8 k, __m256d a);
VCOMPRESSPD _m128d _mm_mask_compress_pd(__m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD _m128d _mm_maskz_compress_pd(__mmask8 k, __m128d a);
VCOMPRESSPD void _mm_mask_compressedstoreu_pd( void *d, __mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.
VCOMPRESSPS—Store Sparse Packed Single Precision Floating-Point Values Into Dense Memory

**Description**

Compress (stores) up to 16 single precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VCOMPRESSPS (EVEX Encoded Versions) Store Form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE := 32

k := 0

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    DEST[k+SIZE-1:k] := SRC[i+31:i]
    k := k + SIZE
  ENDIF;
ENDFOR;

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 [k1][z], xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed single precision floating-point values from xmm2 to xmm1/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 [k1][z], ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed single precision floating-point values from ymm2 to ymm1/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 [k1][z], zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed single precision floating-point values from zmm2 using control mask k1 to zmm1/m512.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
VCOMPRESSPS (EVEX Encoded Versions) Reg-Reg Form

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

\(SIZE := 32\)

\(k := 0\)

\(\text{FOR } j := 0 \text{ TO } KL-1\)

\(i := j * 32\)

\(\text{IF } k1[j] \text{ OR } \text{*no writemask*} \)

\(\text{THEN}\)

\(\text{DEST}[k+SIZE:1:k] := \text{SRC}[i+31:i]\)

\(k := k + SIZE\)

\(\text{FI};\)

\(\text{ENDFOR}\)

\(\text{IF } \text{*merging-masking*}\)

\(\text{THEN } \text{DEST}[VL-1:k] \text{ remains unchanged}\)

\(\text{ELSE } \text{DEST}[VL-1:k] := 0\)

\(\text{FI}\)

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

Intel C/C++ Compiler Intrinsic Equivalent

\(\text{VCOMPRESSPS } __m512 \_mm512\_mask\_compress\_ps( \_m512 s, \_mmask16 k, \_m512 a);\)

\(\text{VCOMPRESSPS } __m512 \_mm512\_maskz\_compress\_ps( \_mmask16 k, \_m512 a);\)

\(\text{VCOMPRESSPS void } \_mm512\_mask\_compress\_storeu\_ps( \text{void } * d, \_mmask16 k, \_m512 a);\)

\(\text{VCOMPRESSPS } __m256 \_mm256\_mask\_compress\_ps( \_mmask8 k, \_m256 a);\)

\(\text{VCOMPRESSPS } __m256 \_mm256\_maskz\_compress\_ps( \_mmask8 k, \_m256 a);\)

\(\text{VCOMPRESSPS void } \_mm256\_mask\_compress\_storeu\_ps( \text{void } * d, \_mmask8 k, \_m256 a);\)

\(\text{VCOMPRESSPS } \_m128 \_mm\_mask\_compress\_ps( \_mmask8 k, \_m128 a);\)

\(\text{VCOMPRESSPS } \_m128 \_mm\_maskz\_compress\_ps( \_mmask8 k, \_m128 a);\)

\(\text{VCOMPRESSPS void } \_mm\_mask\_compress\_storeu\_ps( \text{void } * d, \_mmask8 k, \_m128 a);\)

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb. in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

\#UD \text{ If EVEX.vvvv } != 1111B.\)
VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 5B /&lt;i&gt;r&lt;/i&gt; VCVTDQ2PH xmm1[k1][z], xmm2/ m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 5B /&lt;i&gt;r&lt;/i&gt; VCVTDQ2PH xmm1[k1][z], ymm2/ m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 5B /&lt;i&gt;r&lt;/i&gt; VCVTDQ2PH ymm1[k1][z], zmm2/ m512/m32bcst {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed FP16 values, and store the result in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction converts four, eight, or sixteen packed signed doubleword integers in the source operand to four, eight, or sixteen packed FP16 values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 32-bit memory location. The destination operand is a YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

Operation

```
VCVTDQ2PH DEST, SRC
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
```
DEST[MAXVL-1:VL/2] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTDQ2PH __m256h _mm512_cvtepi32_ph (__m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_mask_cvtepi32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m256h _mm512_maskz_cvtepi32_ph (__mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m128h _mm_cvtepi32_ph (__m128i a);
VCVTDQ2PH __m128h _mm_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm_maskz_cvtepi32_ph (__mmask8 k, __m128i a);
VCVTDQ2PH __m128h _mm256_cvtepi32_ph (__m256i a);
VCVTDQ2PH __m128h _mm256_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTDQ2PH __m128h _mm256_maskz_cvtepi32_ph (__mmask8 k, __m256i a);
VCVTDQ2PH __m256h _mm512_cvtepi32_ph (__m512i a);
VCVTDQ2PH __m256h _mm512_mask_cvtepi32_ph (__m512h src, __mmask16 k, __m512i a);
VCVTDQ2PH __m256h _mm512_maskz_cvtepi32_ph (__mmask16 k, __m512i a);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
**VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F38.W0 72 /r VCVTNE2PS2BF16 xmm1[k1][k2], xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F38.W0 72 /r VCVTNE2PS2BF16 ymm1[k1][k2], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F38.W0 72 /r VCVTNE2PS2BF16 zmm1[k1][k2], zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_BF16</td>
<td>Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts two SIMD registers of packed single data into a single register of packed BF16 data.

This instruction does not support memory fault suppression.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated. No floating-point exceptions are generated.

### Operation

VCVTNE2PS2BF16 dest, src1, src2

VL = (128, 256, 512)

KL = VL/16

origdest := dest

FOR i := 0 to KL-1:

IF k1[i] or *no writemask*:

IF i < KL/2:

IF src2 is memory and evex.b == 1:

\[ t := \text{src2.fp32}[0] \]

ELSE:

\[ t := \text{src2.fp32}[i] \]

ELSE:

\[ t := \text{src1.fp32}[i-KL/2] \]

// See VCVTNEPS2BF16 for definition of convert helper function

dest.word[i] := convert_fp32_to_bfloat16(t)

ELSE IF *zeroing*:

\[ \text{dest.word[i]} := 0 \]

ELSE: // Merge masking, dest element unchanged

\[ \text{dest.word[i]} := \text{origdest.word[i]} \]

DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTNE2PS2BF16 __m128bh _mm_cvtnne2ps_pbh (__m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtnne2ps_pbh (__m128bh, __mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtnne2ps_pbh (__mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtnne2ps_pbh (__m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtnne2ps_pbh (__m256bh, __mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_maskz_cvtnne2ps_pbh (__mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m512bh _mm512_cvtnne2ps_pbh (__m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_mask_cvtnne2ps_pbh (__m512bh, __mmask32, __m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_maskz_cvtnne2ps_pbh (__mmask32, __m512, __m512);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-50, "Type E4NF Class Exception Conditions."
VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 72 /r VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_BF16</td>
<td>Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts one SIMD register of packed single data into a single register of packed BF16 data.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

As the instruction operand encoding table shows, the EVEX.vvvv field is not used for encoding an operand. EVEX.vvvv is reserved and must be 0b1111 otherwise instructions will #UD.

Operation

Define convert_fp32_to_bfloat16(x):

IF x is zero or denormal:
  dest[15] := x[31] // sign preserving zero (denormal go to zero)
  dest[14:0] := 0
ELSE IF x is infinity:
  dest[15:0] := x[31:16]
ELSE IF x is NaN:
  dest[15:0] := x[31:16] // truncate and set MSB of the mantissa to force QNAN
  dest[6] := 1
ELSE // normal number
  LSB := x[16]
  rounding_bias := 0x000007FF + LSB
  temp[31:0] := x[31:0] + rounding_bias // integer add
  dest[15:0] := temp[31:16]
RETURN dest
VCVTNEPS2BF16 *dest, src*

VL = (128, 256, 512)

KL = VL/16

origdest := dest

FOR i := 0 to KL/2-1:
    IF k1[i] or *no writemask*:
        IF src is memory and evex.b == 1:
            t := src.fp32[0]
        ELSE:
            t := src.fp32[i]
        dest.word[i] := convert_fp32_to_bfloat16(t)
    ELSE IF *zeroing*:
        dest.word[i] := 0
    ELSE: // Merge masking, dest element unchanged
        dest.word[i] := origdest.word[i]

DEST[MAXVL-1:VL/2] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128);
VCVTNEPS2BF16 __m128bh __mm_mask_cvtneps_pbh (__m128bh, __mmask8, __m128);
VCVTNEPS2BF16 __m128bh __mm_maskz_cvtneps_pbh (__mmask8, __m128);
VCVTNEPS2BF16 __m128bh __mm256_cvtneps_pbh (__m256);
VCVTNEPS2BF16 __m128bh __mm256_mask_cvtneps_pbh (__m128bh, __mmask8, __m256);
VCVTNEPS2BF16 __m128bh __mm256_maskz_cvtneps_pbh (__mmask8, __m256);
VCVTNEPS2BF16 __m256bh __mm512_cvtneps_pbh (__m512);
VCVTNEPS2BF16 __m256bh __mm512_mask_cvtneps_pbh (__m256bh, __mmask16, __m512);
VCVTNEPS2BF16 __m256bh __mm512_maskz_cvtneps_pbh (__mmask16, __m512);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”
VCVTPD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCVTPD2PH xmm1{k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2PH ymm1{k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four packed FP16 values, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPD2PH zmm1{k1}{z}, zmm2/m512/m64bcst[er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight packed FP16 values, and store the result in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction converts two, four, or eight packed double precision floating-point values in the source operand (second operand) to two, four, or eight packed FP16 values in the destination operand (first operand). When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasts from a 64-bit memory location. The destination operand is a XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:128/64/32) of the corresponding destination are zeroed.

EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

This instruction uses MXCSR.DAZ for handling FP64 inputs. FP16 outputs can be normal or denormal, and are not conditionally flushed to zero.
Operation

**VCVTPD2PH DEST, SRC**

VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE:
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *SRC is memory* and EVEX.b = 1:
         tsrc := SRC.double[0]
      ELSE
         tsrc := SRC.double[j]
      DEST.fp16[j] := Convert_fp64_to_fp16(tsrc)
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPD2PH __m128h _mm512_cvt_roundpd_ph (__m512d a, int rounding);
VCVTPD2PH __m128h _mm512_mask_cvt_roundpd_ph (__m128h src, __mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm512_maskz_cvt_roundpd_ph (__mmask8 k, __m512d a, int rounding);
VCVTPD2PH __m128h _mm512_cvtpd_ph (__m128d a);
VCVTPD2PH __m128h _mm512_mm_mask_cvtpd_ph (__m128h src, __mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm512_mm_maskz_cvtpd_ph (__mmask8 k, __m128d a);
VCVTPD2PH __m128h _mm512_cvtpd_ph (__m128d a);
VCVTPD2PH __m128h _mm512_mm_mask_cvtpd_ph (__m128h src, __mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm512_mm_maskz_cvtpd_ph (__mmask8 k, __m256d a);
VCVTPD2PH __m128h _mm512_cvtpd_ph (__m256d a);
VCVTPD2PH __m128h _mm256_mm512_cvtpd_ph (__m128h src, __mmask8 k, __m512d a);
VCVTPD2PH __m128h _mm256_mm512_mm_mask_cvtpd_ph (__mmask8 k, __m512d a);
VCVTPD2PH __m128h _mm256_mm512_mm_maskz_cvtpd_ph (__mmask8 k, __m512d a);
VCVTPD2PH __m128h _mm256_mm512_cvtpd_ph (__m512d a);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPD2QQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 7B /r</td>
<td>A/V/V</td>
<td>AVX512VL</td>
<td>Convert two packed double precision floating-point values from xmm1/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7B /r</td>
<td>A/V/V</td>
<td>AVX512VL</td>
<td>Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7B /r</td>
<td>A/V/V</td>
<td>AVX512DVQ</td>
<td>Convert eight packed double precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts packed double precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2\(^w\)-1, where w represents the number of bits in the destination format) is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register**

\((\text{KL, VL}) = (2, 128), (4, 256), (8, 512)\)

\(\text{IF (VL }=\text{ 512 AND (EVEX.b }=\text{ 1})\)

\(\text{THEN}\)

\(\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);}\)

\(\text{ELSE}\)

\(\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);}\)

\(\text{FI};\)

\(\text{FOR j := 0 TO KL-1}\)

\(i := j * 64\)

\(\text{IF k1[j] OR *no writemask*}\)

\(\text{THEN DEST[i+63:j] := Convert_Double>Precision_Floating_Point_To_QuadInteger(SRC[i+63:j])}\)

\(\text{ELSE}\)

\(\text{IF *merging-masking}\)

\(\text{THEN *DEST[i+63:j] remains unchanged*}\)

\(\text{ELSE}\)

\(\text{DEST[i+63:j] := 0}\)

\(\text{FI}\)
VCVTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
        ELSE
          DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ __m512i _mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i _mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i _mm256_cvtpd_epi64( __m256d src);
VCVTPD2QQ __m128i _mm_cvtpd_epi64( __m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:

#UD If EVEX.vvvv != 1111B.
VCVTPD2UDQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.256.0F.W1 79 /r VCVTPD2UDQ xmm1 (k1)(z), ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 79 /r VCVTPD2UDQ ymm1 (k1)(z), zmm2/m512/m64bcst[er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts packed double precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value 2^w – 1 is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register

KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32
k := j * 64

IF k1[j] OR *no writemask*
THEN

DEST[i+31:i] :=
Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63k])
ELSE

IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+31:i] := 0
Fi
Fi;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

VCVTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])
            ELSE
                DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
            Fi;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            Fi
        Fi
    Fi;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32( __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:
#UD If EVEX.vvvv != 1111B.
VCVTPD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert fourth packed double precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst[er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts packed double precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

IF $(VL == 512) AND (EVEX.b == 1)$

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  
ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR $j := 0$ TO $KL-1$

$i := j * 64$

IF $k1[j]$ OR *no writemask*

THEN DEST[i+63:j] := 

Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:j])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:j] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:j] := 0

FI
VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
        ELSE
          DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI;
  ENDIF;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UQQ __m512i _mm512_cvtpd_epu64( __m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64( __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64( __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvt_roundpd_epu64( __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_cvtpd_epu64( __m256d src)
VCVTPD2UQQ __m128i _mm_cvtpd_epu64( __m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:

#UD If EVEX.vvvv != 1111B.
VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers

### Op/En Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP5.W0 5B /r VCVTPH2DQ xmm1[k1][z], xmm2/m64/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAP5.W0 5B /r VCVTPH2DQ ymm1[k1][z], xmm2/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAP5.W0 5B /r VCVTPH2DQ zmm1[k1][z], xmm2/m256/m16bcst [er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert sixteen packed FP16 values in xmm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTPH2DQ DEST, SRC**

VL = 128, 256 or 512  
KL := VL / 32

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):  
  SET_RM(EVEX.RC)  
ELSE:  
  SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:  
  IF k1[j] OR *no writemask*:  
    IF *SRC is memory* and EVEX.b = 1:  
      tsrc := SRC.fp16[0]  
    ELSE  
      tsrc := SRC.fp16[j]  
    DEST.dword[j] := Convert_fp16_to_integer32(tsrc)  
  ELSE IF *zeroing*:  
    DEST.dword[j] := 0  
  // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPH2DQ __m512i _mm512_cvt_roundph_epi32 (__m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_mask_cvt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m512i _mm512_maskz_cvt_roundph_epi32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2DQ __m128i _mm_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m128i _mm_mask_cvtph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2DQ __m128i _mm_maskz_cvtph_epi32 (__mmask8 k, __m128h a);
VCVTPH2DQ __m256i _mm256_cvtph_epi32 (__m128h a);
VCVTPH2DQ __m256i _mm256_mask_cvtph_epi32 (__m256i src, __mmask8 k, __m256h a);
VCVTPH2DQ __m256i _mm256_maskz_cvtph_epi32 (__mmask8 k, __m256h a);
VCVTPH2DQ __m512i _mm512_mask_cvtph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2DQ __m512i _mm512_maskz_cvtph_epi32 (__mmask16 k, __m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPH2PD—Convert Packed FP16 Values to FP64 Values

**Description**

This instruction converts packed FP16 values to FP64 values in the destination register. The destination elements are updated according to the writemask.

This instruction handles both normal and denormal FP16 inputs.

**Operation**

VCVTPH2PD DEST, SRC

VL = 128, 256, or 512

KL := VL/64

FOR j := 0 TO KL-1:

  IF k1[j] OR *no writemask*:
    IF *SRC is memory* and EVEX.b = 1:
      tsrc := SRC.fp16[0]
    ELSE
      tsrc := SRC.fp16[j]
    DEST.fp64[j] := Convert_fp16_to_fp64(tsrc)
  ELSE IF *zeroing*:
    DEST.fp64[j] := 0
    // else dest.fp64[j] remains unchanged

DEST[MAXVL-1:VL] := 0

---

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert packed FP16 values in xmm2/m32/m16bcst to FP64 values, and store result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPH2PD xmm1{k1}{z}, xmm2/m32/m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert packed FP16 values in xmm2/m64/m16bcst to FP64 values, and store result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPH2PD ymm1{k1}{z}, xmm2/m64/m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert packed FP16 values in xmm2/m128/m16bcst to FP64 values, and store result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VCVTPH2PD zmm1{k1}{z}, xmm2/m128/m16bcst [sae]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Quarter</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

Vol. 2C 5-47
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PD __m512d _mm512_cvt_roundph_pd (__m128h a, int sae);
VCVTPH2PD __m512d _mm512_mask_cvt_roundph_pd (__m512d src, __mmask8 k, __m128h a, int sae);
VCVTPH2PD __m512d _mm512_maskz_cvt_roundph_pd (__mmask8 k, __m128h a, int sae);
VCVTPH2PD __m128d _mm_cvtpd (__m128h a);
VCVTPH2PD __m128d _mm_mask_cvtpd (__m128d src, __mmask8 k, __m128h a);
VCVTPH2PD __m128d _mm_maskz_cvtpd (__mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_cvtpd (__m128h a);
VCVTPH2PD __m256d _mm256_mask_cvtpd (__m256d src, __mmask8 k, __m128h a);
VCVTPH2PD __m256d _mm256_maskz_cvtpd (__mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_cvtpd (__m128h a);
VCVTPH2PD __m512d _mm512_mask_cvtpd (__m512d src, __mmask8 k, __m128h a);
VCVTPH2PD __m512d _mm512_maskz_cvtpd (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPH2PS/VCVTPH2PSX—Convert Packed FP16 Values to Single Precision Floating-Point Values

**Opcode/Instruction** | Op/En | 64/32 Bit Mode Support | CPUID Feature Flag | Description |
---|---|---|---|---|
VEX.128.66.0F38.W0 13 /r | A | V/V | F16C | Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point value in xmm1. |
VEX.256.66.0F38.W0 13 /r | A | V/V | F16C | Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point value in ymm1. |
EVEX.128.66.0F38.W0 13 /r | B | V/V | AVX512VL, AVX512F | Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point values in xmm1 subject to writemask k1. |
EVEX.256.66.0F38.W0 13 /r | B | V/V | AVX512VL, AVX512F | Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point values in ymm1 subject to writemask k1. |
EVEX.512.66.0F38.W0 13 /r | B | V/V | AVX512F | Convert sixteen packed FP16 values in ymm2/m256 to packed single precision floating-point values in zmm1 subject to writemask k1. |
EVEX.128.66.MAP6.W0 13 /r | C | V/V | AVX512-FP16 AVX512VL | Convert four packed FP16 values in xmm2/m64/m16bcst to four packed single precision floating-point values, and store result in xmm1 subject to writemask k1. |
EVEX.256.66.MAP6.W0 13 /r | C | V/V | AVX512-FP16 AVX512VL | Convert eight packed FP16 values in xmm2/m128/m16bcst to eight packed single precision floating-point values, and store result in ymm1 subject to writemask k1. |
EVEX.512.66.MAP6.W0 13 /r | C | V/V | AVX512-FP16 | Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen packed single precision floating-point values, and store result in zmm1 subject to writemask k1. |

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.
The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) floating-point values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

The VCVTPH2PSX instruction is a new form of the PH to PS conversion instruction, encoded in map 6. The previous version of the instruction, VCVTPH2PS, that is present in AVX512F (encoded in map 2, 0F38) does not support embedded broadcasting. The VCVTPH2PSX instruction has the embedded broadcasting option available.

The instructions associated with AVX512_FP16 always handle FP16 denormal number inputs; denormal inputs are not treated as zero.

**Operation**

```c
vCvt_h2s(SRC1[15:0])
{
    RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

**VCVTPH2PS (EVEX Encoded Versions)**

(\(KL, VL\)) = (4, 128), (8, 256), (16, 512)

FOR \(j := 0\) TO \(KL-1\)
  \(i := j \times 32\)
  \(k := j \times 16\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(DEST[i+31:i] := vCvt_h2s(SRC[k+15:k])\)
    ELSE
      IF *merging-masking* ; merging-masking
        THEN \(DEST[i+31:i] \)remains unchanged*
        ELSE ; zeroing-masking
          \(DEST[i+31:i] := 0\)
      FI
  ENDIF
ENDFOR
\(DEST[MAXVL-1:VL] := 0\)
VCVTPH2PS (VEX.256 Encoded Version)

\[
\begin{align*}
\text{DEST}[31:0] &= \text{vCvt\_h2s(SRC1[15:0])}; \\
\text{DEST}[63:32] &= \text{vCvt\_h2s(SRC1[31:16])}; \\
\text{DEST}[95:64] &= \text{vCvt\_h2s(SRC1[47:32])}; \\
\text{DEST}[127:96] &= \text{vCvt\_h2s(SRC1[63:48])}; \\
\text{DEST}[159:128] &= \text{vCvt\_h2s(SRC1[79:64])}; \\
\text{DEST}[191:160] &= \text{vCvt\_h2s(SRC1[95:80])}; \\
\text{DEST}[223:192] &= \text{vCvt\_h2s(SRC1[111:96])}; \\
\text{DEST}[255:224] &= \text{vCvt\_h2s(SRC1[127:112])}; \\
\text{DEST}[\text{MAXVL}-1:256] &= 0
\end{align*}
\]

VCVTPH2PS (VEX.128 Encoded Version)

\[
\begin{align*}
\text{DEST}[31:0] &= \text{vCvt\_h2s(SRC1[15:0])}; \\
\text{DEST}[63:32] &= \text{vCvt\_h2s(SRC1[31:16])}; \\
\text{DEST}[95:64] &= \text{vCvt\_h2s(SRC1[47:32])}; \\
\text{DEST}[127:96] &= \text{vCvt\_h2s(SRC1[63:48])}; \\
\text{DEST}[\text{MAXVL}-1:128] &= 0
\end{align*}
\]

VCVTPH2PSX DEST, SRC

\(VL = 128, 256, \text{ or } 512\)

\(KL := VL/32\)

\[
\text{FOR } j := 0 \text{ TO } KL-1:\
\text{IF } k1[j] \text{ OR *no writemask*}:
\text{IF } *\text{SRC is memory* and EVEX.b = 1:}
\quad \text{tsrc} := \text{SRC.fp16[0]}
\text{ELSE}
\quad \text{tsrc} := \text{SRC.fp16[j]}
\text{ELSE IF } *\text{zeroing*}:
\quad \text{DEST.fp32[j]} := 0
\quad // else dest.fp32[j] remains unchanged
\]

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
\text{VCVTPH2PS } &\_\_m512 \_\_mm512\_\_cvtph\_ps( \_\_m256i a); \\
\text{VCVTPH2PS } &\_\_m512 \_\_mm512\_\_mask\_cvtph\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m256i a); \\
\text{VCVTPH2PS } &\_\_m512 \_\_mm512\_\_maskz_cvtph\_ps(\_\_mmask16 k, \_\_m256i a); \\
\text{VCVTPH2PS } &\_\_m512 \_\_mm512\_\_cvt\_roundph\_ps(\_\_m256i a, int sae); \\
\text{VCVTPH2PS } &\_\_m512 \_\_mm512\_\_mask\_cvt\_roundph\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m256i a, int sae); \\
\text{VCVTPH2PS } &\_\_m512 \_\_mm512\_\_maskz\_cvt\_roundph\_ps(\_\_mmask16 k, \_\_m256i a, int sae); \\
\text{VCVTPH2PS } &\_\_m256 \_\_mm256\_\_cvtph\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m128i a); \\
\text{VCVTPH2PS } &\_\_m256 \_\_mm256\_\_maskz\_cvtph\_ps(\_\_mmask8 k, \_\_m128i a); \\
\text{VCVTPH2PS } &\_\_m128 \_\_mm\_mask\_cvtph\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128i a); \\
\text{VCVTPH2PS } &\_\_m128 \_\_mm\_maskz\_cvtph\_ps(\_\_mmask8 k, \_\_m128i a); \\
\text{VCVTPH2PS } &\_\_m128 \_\_mm\_cvtph\_ps (\_\_m128i m1); \\
\text{VCVTPH2PS } &\_\_m256 \_\_mm256\_\_cvtph\_ps (\_\_m128i m1)
\end{align*}
\]

\[
\begin{align*}
\text{VCVTPH2PSX } &\_\_m512 \_\_mm512\_\_cvt\_roundph\_ps (\_\_m256h a, int sae); \\
\text{VCVTPH2PSX } &\_\_m512 \_\_mm512\_\_mask\_cvt\_roundph\_ps (\_\_m512 src, \_\_mmask16 k, \_\_m256h a, int sae);
\end{align*}
\]
VCVTPH2PSX __m512 __mm512_maskz_cvtx_roundph_ps (__mmask16 k, __m256h a, int sae);
VCVTPH2PSX __m128 __mm_cvtxph_ps (__m128h a);
VCVTPH2PSX __m128 __mm_mask_cvtxph_ps (__m128 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m128 __mm_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m256 __mm256_cvtxph_ps (__m128h a);
VCVTPH2PSX __m256 __mm256_mask_cvtxph_ps (__m256 src, __mmask8 k, __m128h a);
VCVTPH2PSX __m256 __mm256_maskz_cvtxph_ps (__mmask8 k, __m128h a);
VCVTPH2PSX __m512 __mm512_mask_cvtxph_ps (__m512 src, __mmask16 k, __m256h a);
VCVTPH2PSX __m512 __mm512_maskz_cvtxph_ps (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

VEX-encoded instructions: Invalid.
EVEX-encoded instructions: Invalid.
EVEX-encoded instructions with broadcast (VCVTPH2PSX): Invalid, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-26, “Type 11 Class Exception Conditions” (do not report #AC).
EVEX-encoded instructions, see Table 2-60, “Type E11 Class Exception Conditions.”
EVEX-encoded instructions with broadcast (VCVTPH2PSX), see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:

#UD If VEX.W=1.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP5.W0 7B /r VCVTPH2QQ xmm1[k1][z], xmm2/m32/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16, AVX512VL</td>
<td>Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAP5.W0 7B /r VCVTPH2QQ ymm1[k1][z], xmm2/m64/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16, AVX512VL</td>
<td>Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAP5.W0 7B /r VCVTPH2QQ zmm1[k1][z], xmm2/m128/m16bcst [er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Quarter</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed FP16 values in the source operand to signed quadword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

**Operation**

`VCVTPH2QQ DEST, SRC`

`VL = 128, 256 or 512`

`KL := VL / 64`

**IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):**

- **SET_R(M(EVEX.RC))**

**ELSE:**

- **SET_R(M(MXCSR.RC))**

**FOR j := 0 TO KL-1:**

- **IF k1[j] OR *no writemask*:**

  **IF *SRC is memory* and EVEX.b = 1:**

  - `tssc := SRC.fp16[0]`

  **ELSE**

  - `tssc := SRC.fp16[j]`

  - **DEST.qword[j] := Convert_fp16_to_integer64(tssc)**

  **ELSE IF *zeroing*:**

  - **DEST.qword[j] := 0**

  // else dest.qword[j] remains unchanged

- **DEST[MAXVL-1:VL] := 0**
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2QQ__m512i_mm512_cvt_roundph_epi64 (__m128h a, int rounding);
VCVTPH2QQ__m512i_mm512_mask_cvt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2QQ__m512i_mm512_maskz_cvt_roundph_epi64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2QQ__m128i__mm_cvtph_epi64 (__m128h a);
VCVTPH2QQ__m128i_mm_mask_cvtph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2QQ__m128i__mm_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ__m256i__mm256_cvtph_epi64 (__m128h a);
VCVTPH2QQ__m256i_mm256_mask_cvtph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2QQ__m256i__mm256_maskz_cvtph_epi64 (__mmask8 k, __m128h a);
VCVTPH2QQ__m512i__mm512_cvtph_epi64 (__m128h a);
VCVTPH2QQ__m512i_mm512_mask_cvtph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2QQ__m512i__mm512_maskz_cvtph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM reg (w)</td>
<td>ModRM m/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTPH2UDQ DEST, SRC**

- \( VL = 128, 256 \) or 512
- \( KL := VL / 32 \)

Set the writemask

- If *SRC is a register* and \( VL = 512 \) and \( \text{EVEX.b} = 1 \):
  - Set \( \text{RM(EVEX.RC)} \)
- Else:
  - Set \( \text{RM(MXCSR.RC)} \)

For \( j := 0 \) to \( KL-1 \):

- If \( k1[j] \) OR *no writemask*:
  - If *SRC is memory* and \( \text{EVEX.b} = 1 \):
    - \( \text{tsrc} := \text{SRC.fp16}[0] \)
  - Else:
    - \( \text{tsrc} := \text{SRC.fp16}[j] \)
    - \( \text{DEST.dword}[j] := \text{Convert_fp16_to_unsigned_integer32}(\text{tsrc}) \)

- Else if *zeroing*:
  - \( \text{DEST.dword}[j] := 0 \)

  // else \( \text{dest.dword}[j] \) remains unchanged

\( \text{DEST[MAXVL-1:VL]} := 0 \)
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPH2UDQ __m512i __mm512_cvt_roundph_epu32 (__m256h a, int rounding);
VCVTPH2UDQ __m512i __mm512_mask_cvt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m512i __mm512_maskz_cvt_roundph_epu32 (__mmask16 k, __m256h a, int rounding);
VCVTPH2UDQ __m128i __mm_cvtph_epu32 (__m128h a);
VCVTPH2UDQ __m128i __mm_mask_cvtph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m128i __mm_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m256i __mm256_cvtph_epu32 (__m256h a);
VCVTPH2UDQ __m256i __mm256_mask_cvtph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UDQ __m256i __mm256_maskz_cvtph_epu32 (__mmask8 k, __m128h a);
VCVTPH2UDQ __m512i __mm512_cvtph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTPH2UDQ __m512i __mm512_mask_cvtph_epu32 (__mmask16 k, __m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128:66.MAP5.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert two packed FP16 values in xmm2/m32/m16bcst to two unsigned quadword integers, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256:66.MAP5.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned quadword integers, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512:66.MAP5.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned quadword integers, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

**Operation**

VCVTPH2UQQ DEST, SRC

VL = 128, 256 or 512

KL := VL / 64

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):

    SET_RM(EVEX.RC)
ELSE:

    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

    IF k1[j] OR *no writemask*:

        IF *SRC is memory* and EVEX.b = 1:

            tsrc := SRC.fp16[0]

        ELSE:

            tsrc := SRC.fp16[j]

        DEST.qword[j] := Convert_fp16_to_unsigned_integer64(tsrc)

    ELSE IF *zeroing*:

        DEST.qword[j] := 0

    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPH2UQQ __m512i _mm512_cvt_roundph_epu64 (__m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_mask_cvt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m512i _mm512_maskz_cvt_roundph_epu64 (__mmask8 k, __m128h a, int rounding);
VCVTPH2UQQ __m128i _mm_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m128i _mm_mask_cvtph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m128i _mm_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m256i _mm256_mask_cvtph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_maskz_cvtph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_cvtph_epu64 (__m128h a);
VCVTPH2UQQ __m512i _mm512_mask_cvtph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_maskz_cvtph_epu64 (__mmask8 k, __m128h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
**VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 7D /r VCVTPH2UW xmm1[k1][j], xmm2/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert packed FP16 values in xmm2/m128/m16bcst to unsigned word integers, and store the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 7D /r VCVTPH2UW ymm1[k1][j], ymm2/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert packed FP16 values in ymm2/m256/m16bcst to unsigned word integers, and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 7D /r VCVTPH2UW zmm1[k1][j], zmm2/m512/m16bcst [er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert packed FP16 values in zmm2/m512/m16bcst to unsigned word integers, and store the result in zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTPH2UW DEST, SRC**

VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.word[j] := Convert_fp16_to_unsigned_integer16(tsrc)
    ELSE IF *zeroing*:
        DEST.word[j] := 0
        // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPH2UW __m512i __m512_cvt_roundph_epu16 (__m512h a, int sae);
VCVTPH2UW __m512i __m512_mask_cvt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTPH2UW __m512i __m512_maskz_cvt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTPH2UW __m128i __m512cvtph_epu16 (__m128h a);
VCVTPH2UW __m128i __m128_mask_cvtph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UW __m128i __m128_maskz_cvtph_epu16 (__mmask8 k, __m128h a);
VCVTPH2UW __m256i __m256_cvtph_epu16 (__m256h a);
VCVTPH2UW __m256i __m256_mask_cvtph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2UW __m256i __m256_maskz_cvtph_epu16 (__mmask16 k, __m256h a);
VCVTPH2UW __m512i __m512_mask_cvtph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2UW __m512i __m512_maskz_cvtph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers

**Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed FP16 values in the source operand to signed word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

**Operation**

**VCVTPH2W DEST, SRC**

\[ VL = 128, 256 or 512 \]

\[ KL := VL / 16 \]

IF *SRC is a register* and (VL = 512) and (EVEX.b = 1):

\[ \text{SET_RM(EVEX.RC)} \]

ELSE:

\[ \text{SET_RM(MXCSR.RC)} \]

FOR \( j := 0 \) TO \( KL-1 \):

IF \( k1[j] \) OR *no writemask*:

IF *SRC is memory* and EVEX.b = 1:

\[ \text{tsrc := SRC.fp16[0]} \]

ELSE

\[ \text{tsrc := SRC.fp16[j]} \]

\[ \text{DEST.word[j] := Convert_fp16_to_integer16(tsrc)} \]

ELSE IF *zeroing*:

\[ \text{DEST.word[j] := 0} \]

// else dest.word[j] remains unchanged

\[ \text{DEST[MAXVL-1:VL] := 0} \]
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2W __m512i _mm512_cvt_roundph_epi16 (__m512h a, int rounding);
VCVTPH2W __m512i _mm512_mask_cvt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int rounding);
VCVTPH2W __m512i _mm512_maskz_cvt_roundph_epi16 (__mmask32 k, __m512h a, int rounding);
VCVTPH2W __m128i _mm_cvtph_epi16 (__m128h a);
VCVTPH2W __m128i _mm_mask_cvtph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2W __m128i _mm_maskz_cvtph_epi16 (__mmask8 k, __m128h a);
VCVTPH2W __m256i _mm256_cvtph_epi16 (__m256h a);
VCVTPH2W __m256i _mm256_mask_cvtph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTPH2W __m256i _mm256_maskz_cvtph_epi16 (__mmask16 k, __m256h a);
VCVTPH2W __m512i _mm512_cvtph_epi16 (__m512h a);
VCVTPH2W __m512i _mm512_mask_cvtph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTPH2W __m512i _mm512_maskz_cvtph_epi16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTPS2PH—Convert Single-Precision FP Value to 16-bit FP Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F3A.W0 1D / r ib VCVT2PH xmm1/m64, xmm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 1D / r ib VCVT2PH xmm1/m128, ymm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 1D / r ib VCVT2PH xmm1/m64 [k1][z], xmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 1D / r ib VCVT2PH xmm1/m128 [k1][z], ymm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1D / r ib VCVT2PH ymm1/m256 [k1][z], zmm2{sae}, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.</td>
</tr>
</tbody>
</table>

**Description**

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 5-7. VCVT2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-13.
Table 5-13. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Name/value</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm[1:0]</td>
<td>RC=00B</td>
<td>Round to nearest even</td>
<td>If imm[2] = 0</td>
</tr>
<tr>
<td></td>
<td>RC=01B</td>
<td>Round down</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=10B</td>
<td>Round up</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=11B</td>
<td>Truncate</td>
<td></td>
</tr>
<tr>
<td>Imm[2]</td>
<td>MS1=0</td>
<td>Use imm[1:0] for rounding</td>
<td>Ignore MXCSR.RC</td>
</tr>
<tr>
<td></td>
<td>MS1=1</td>
<td>Use MXCSR.RC for rounding</td>
<td></td>
</tr>
<tr>
<td>Imm[7:3]</td>
<td>Ignored</td>
<td>Ignored by processor</td>
<td></td>
</tr>
</tbody>
</table>

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAXVL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.

Operation

\[
\text{vCvt_s2h}(\text{SRC}[31:0])
\]

\[
\text{IF imm}[2] = 0
\]

\[
\text{THEN} \quad \text{using imm}[1:0] \text{ for rounding, see Table 5-13}
\]

\[
\text{RETURN CvtSinglePrecisionToHalfPrecision_FP_Imm(SRC[31:0])};
\]

\[
\text{ELSE} \quad \text{using MXCSR.RC for rounding}
\]

\[
\text{RETURN CvtSinglePrecisionToHalfPrecision_FP_Mxcsr(SRC[31:0])};
\]

\[
\text{FI};
\]

VCVTPS2PH (EVEX Encoded Versions) When DEST is a Register

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR \(j := 0 \text{ TO } KL-1\)

\[
i := j \times 16
\]

\[
k := j \times 32
\]

IF \(k1[j] \text{ OR } \text{no writemask}\)

\[
\text{THEN DEST}[i+15:j] :=
\]

\[
vCvt_s2h(SRC[k+31:k])
\]

ELSE

\[
\text{IF } \text{merging-masking} \quad \text{; merging-masking}
\]

\[
\text{THEN } \text{DEST}[i+15:j] \text{ remains unchanged}
\]

ELSE

\[
\text{ELSE } \quad \text{zeroing-masking}
\]

\[
\text{DEST}[i+15:i] := 0
\]

\[
\text{FI}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL-1:VL/2}] := 0
\]
VCVTPS2PH (EVEX Encoded Versions) When DEST is Memory

\( KL, VL = (4, 128), (8, 256), (16, 512) \)

FOR \( j \) := 0 TO KL-1
  \( i := j \times 16 \)
  \( k := j \times 32 \)
  IF \( k1[j] \) OR *no writemask*
    THEN \( \text{DEST}[i+15:i] := \text{vCvt_s2h(SRC}[k+31:k]) \)
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VCVTPS2PH (VEX.256 Encoded Version)

\( \text{DEST}[15:0] := \text{vCvt_s2h(SRC}[1][31:0]); \)
\( \text{DEST}[31:16] := \text{vCvt_s2h(SRC}[1][63:32]); \)
\( \text{DEST}[47:32] := \text{vCvt_s2h(SRC}[1][95:64]); \)
\( \text{DEST}[63:48] := \text{vCvt_s2h(SRC}[1][127:96]); \)
\( \text{DEST}[79:64] := \text{vCvt_s2h(SRC}[1][159:128]); \)
\( \text{DEST}[95:80] := \text{vCvt_s2h(SRC}[1][191:160]); \)
\( \text{DEST}[111:96] := \text{vCvt_s2h(SRC}[1][223:192]); \)
\( \text{DEST}[127:112] := \text{vCvt_s2h(SRC}[1][255:224]); \)
\( \text{DEST}[\text{MAXVL}-1:128] := 0 \)

VCVTPS2PH (VEX.128 Encoded Version)

\( \text{DEST}[15:0] := \text{vCvt_s2h(SRC}[1][31:0]); \)
\( \text{DEST}[31:16] := \text{vCvt_s2h(SRC}[1][63:32]); \)
\( \text{DEST}[47:32] := \text{vCvt_s2h(SRC}[1][95:64]); \)
\( \text{DEST}[63:48] := \text{vCvt_s2h(SRC}[1][127:96]); \)
\( \text{DEST}[\text{MAXVL}-1:64] := 0 \)

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

\( \text{VCVTPS2PH}\_\_m256i\_\_mm512\_\_cvtps\_ph(\_\_m512\ a); \)
\( \text{VCVTPS2PH}\_\_m256i\_\_mm512\_\_mask\_cvtps\_ph(\_\_m256i\ s, \_\_mmask\_\_m512\ a); \)
\( \text{VCVTPS2PH}\_\_m256i\_\_mm512\_\_maskz\_cvtps\_ph(\_\_mmask\_\_m512\ a, \_\_m512\ a); \)
\( \text{VCVTPS2PH}\_\_m256i\_\_mm512\_\_cvrt\_roundps\_ph(\_\_m512\ a, \_\_m512\ a, \_\_m512\ a); \)
\( \text{VCVTPS2PH}\_\_m256i\_\_mm512\_\_mask\_cvrt\_roundps\_ph(\_\_mmask\_\_m512\ a, \_\_m512\ a, \_\_m512\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm256\_\_mask\_cvtps\_ph(\_\_m128i\ s, \_\_mmask\_\_m256\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm256\_\_maskz\_cvtps\_ph(\_\_mmask\_\_m256\ a, \_\_m256\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm256\_\_maskz\_cvtps\_ph(\_\_mmask\_\_m256\ a, \_\_m256\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm\_cvtps\_ph(\_\_m128\ a, \_\_m128\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm\_cvtps\_ph(\_\_m128\ a, \_\_m128\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm\_cvtps\_ph(\_\_m256\ a, \_\_m256\ a); \)
\( \text{VCVTPS2PH}\_\_m128i\_\_mm\_cvtps\_ph(\_\_m256\ a, \_\_m256\ a); \)

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).
Other Exceptions
VEX-encoded instructions, see Table 2-26, “Type 11 Class Exception Conditions” (do not report #AC); EVEX-encoded instructions, see Table 2-60, “Type E11 Class Exception Conditions.”
Additionally:
#UD If VEX.W=1.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAPS.w0 1D /r VCVTPS2PHX xmm1[k1][k2], xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAPS.w0 1D /r VCVTPS2PHX xmm1[k1][k2], ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed single precision floating-point values in ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAPS.w0 1D /r VCVTPS2PHX ymm1[k1][k2], zmm2/m512/m32bcst (er)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert sixteen packed single precision floating-point values in zmm2/m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction converts packed single precision floating values in the source operand to FP16 values and stores to the destination operand.

The VCVTPS2PHX instruction supports broadcasting.

This instruction uses MXCSR.DAZ for handling FP32 inputs. FP16 outputs can be normal or denormal numbers, and are not conditionally flushed based on MXCSR settings.

Operation

VCVTPS2PHX DEST, SRC (AVX512_FP16 Load Version With Broadcast Support)

VL = 128, 256, or 512
KL := VL / 32

IF *SRC is a register* and (VL == 512) and (EVEX.b = 1):
  SET_RM(EVEX.RC)
ELSE:
  SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF *SRC is memory* and EVEX.b = 1:
      tsrc := SRC.fp32[0]
    ELSE
      tsrc := SRC.fp32[j]
    DEST.fp16[j] := Convert_fp32_to_fp16(tsrc)
    ELSE IF *zeroing*:
      DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
  DEST[MAXVL-1:VL/2] := 0
Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PHX __m256h _mm512_cvtx_roundps_ph (__m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_mask_cvtx_roundps_ph (__m256h src, __mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m256h _mm512_maskz_cvtx_roundps_ph (__mmask16 k, __m512 a, int rounding);
VCVTPS2PHX __m128h _mm_cvtxps_ph (__m128 a);
VCVTPS2PHX __m128h _mm_mask_cvtxps_ph (__m128 src, __mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm_maskz_cvtxps_ph (__mmask8 k, __m128 a);
VCVTPS2PHX __m128h _mm256_cvtxps_ph (__m256 a);
VCVTPS2PHX __m128h _mm256_mask_cvtxps_ph (__m128 src, __mmask8 k, __m256 a);
VCVTPS2PHX __m128h _mm256_maskz_cvtxps_ph (__mmask8 k, __m256 a);
VCVTPS2PHX __m256h _mm512_cvtxps_ph (__m512 a);
VCVTPS2PHX __m256h _mm512_mask_cvtxps_ph (__m256h src, __mmask16 k, __m512 a);
VCVTPS2PHX __m256h _mm512_maskz_cvtxps_ph (__mmask16 k, __m512 a);

SIMD Floating-Point Exceptions
Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If VEX.W=1.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>A/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}</td>
<td>A/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts eight packed single precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^w-1, where w represents the number of bits in the destination format) is returned.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

\( (KL, VL) = (2, 128), (4, 256), (8, 512) \)

\( \text{IF } (VL == 512) \text{ AND } (EVEX.b == 1) \)

\( \text{THEN} \)

\( \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC)}; \)

\( \text{ELSE} \)

\( \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC)}; \)

\( \text{Ft} \)

\( \text{FOR } j := 0 \text{ TO } KL-1 \)

\( \text{i} := j \times 64 \)

\( \text{k} := j \times 32 \)

\( \text{IF } k1[j] \text{ OR } * \text{no writemask}* \)

\( \text{THEN } \text{DEST}[i+63:j] := \text{Convert_Single_Precision_To_QuadInteger}(	ext{SRC}[k+31:k]) \)

\( \text{ELSE} \)

\( \text{IF } * \text{merging-masking} \)

\( \text{THEN } \text{DEST}[i+63:j] \text{ remains unchanged} \)

\( \text{ELSE} \)

\( \text{DEST}[i+63:j] := 0 \)

\( \text{FI} \)
VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0\) TO \(KL-1\)
\(i := j \times 64\)
\(k := j \times 32\)
IF \(k1[j] \text{ OR } \text{*no writemask*}\)
THEN
IF (EVEX.b == 1)
THEN
\(\text{DEST}[i+63:i] := \text{Convert Single Precision To QuadInteger}(\text{SRC}[31:0])\)
ELSE
\(\text{DEST}[i+63:i] := \text{Convert Single Precision To QuadInteger}(\text{SRC}[k+31:k])\)
FI;
ELSE
IF \(*\text{merging-masking}^*\)  ; merging-masking
THEN \(*\text{DEST}[i+63:i] \text{ remains unchanged*}\)
ELSE  ; zeroing-masking
\(\text{DEST}[i+63:i] := 0\)
FI
FI;
ENDFOR
\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTPS2QQ __m512i _mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

Additionally:

\#UD  \hspace{1cm} \text{If EVEX.vvvv} != 1111B.
VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts sixteen packed single precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] :=
      Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
  FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no *
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] :=
      Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])
    ELSE
      DEST[i+31:i] :=
      Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2UDQ __m512i _mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvtt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32( __m256 a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm128_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i _mm128_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm128_maskz_cvtps_epu32( __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.
VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 79 /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 79 /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 79 /r</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts up to eight packed single precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64-bit) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

IF $(VL == 512) \text{ AND } (EVEX.b == 1)$

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX:RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR:RC);

FI;

FOR $j := 0 \text{ TO } KL - 1$

$i := j \times 64$

$k := j \times 32$

IF $k1[j] \text{ OR } *\text{no writemask}*$

THEN

DEST[i+63:i] := Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged* ; zeroing-masking

ELSE

DEST[i+63:i] := 0

FI
VCVTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0, 1\)
\(i := j \times 64\)
\(k := j \times 32\)
IF \(k1[j] \lor \text{no writemask}\)
THEN
IF (EVEX.b == 1)
THEN
\(\text{DEST}[i+63:i] := \text{Convert\_Single\_Precision\_To\_UQuadInteger(SRC[31:0])}\)
ELSE
\(\text{DEST}[i+63:i] := \text{Convert\_Single\_Precision\_To\_UQuadInteger(SRC[k+31:k])}\)
FI;
ELSE
If *merging-masking* ; merging-masking
THEN \(\text{DEST}[i+63:i] \text{ remains unchanged}\)
ELSE ; zeroing-masking
\(\text{DEST}[i+63:i] := 0\)
FI
ENDFOR
\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VCVTPS2UQQ} & \quad _{\text{m512i}} \quad _{\text{mm512_cvtsp_eput64}}(_{\text{m512 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m512i}} \quad _{\text{mm512_mask_cvtsp_eput64}}(_{\text{m512i s, mmask16 k, m512 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m512i}} \quad _{\text{mm512_cvtRoundps_eput64}}(_{\text{m512 a, int r}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m512i}} \quad _{\text{mm512_mask_cvtRoundps_eput64}}(_{\text{m512i s, mmask16 k, m512 a, int r}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m512i}} \quad _{\text{mm512_cvtRoundps_eput64}}(_{\text{mmask16 k, m512 a, int r}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m256i}} \quad _{\text{mm256_cvtsp_eput64}}(_{\text{m256 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m256i}} \quad _{\text{mm256_mask_cvtsp_eput64}}(_{\text{m256i s, mmask8 k, m256 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m256i}} \quad _{\text{mm256_mask_cvtRoundps_eput64}}(_{\text{mmask8 k, m256 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m128i}} \quad _{\text{mm_cvtsp_eput64}}(_{\text{m128 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m128i}} \quad _{\text{mm_mask_cvtsp_eput64}}(_{\text{m128i s, mmask8 k, m128 a}}) \\
\text{VCVTPS2UQQ} & \quad _{\text{m128i}} \quad _{\text{mm_maskcvtsp_eput64}}(_{\text{mmask8 k, m128 a}})
\end{align*}
\]

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

Additionally:

\#UD \quad \text{If EVEX.vvvv} \neq 1111B.
VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values

### Description
Converts packed quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation
#### VCVTQQ2PD (EVEX2 Encoded Versions) When SRC Operand is a Register
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE 
            ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

### Instruction Operands

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W1/E</td>
<td>A Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1/E</td>
<td>A Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1/E</td>
<td>A Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst(er)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Converts packed quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
VCVTQQ2PD (EVEX Encoded Versions) when SRCOperand is a Memory Source

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]

\[
i := j \times 64
\]

\[
\text{IF } k1[j] \text{ OR } \text{*no writemask*} \quad \text{THEN}
\]

\[
\text{IF } (\text{EVEX.b} == 1)
\]

\[
\quad \text{DEST}[i+63:j] := \text{Convert_QuadInteger_To_Double_Precision_Floating_Point}(\text{SRC}[63:0])
\]

\[
\text{ELSE}
\]

\[
\quad \text{DEST}[i+63:j] := \text{Convert_QuadInteger_To_Double_Precision_Floating_Point}(\text{SRC}[i+63:j])
\]

\[
\quad \text{FI};
\]

\[
\text{ELSE}
\]

\[
\text{IF } \text{*merging-masking*} ; \text{merging-masking}
\]

\[
\quad \text{THEN } \text{*DEST}[i+63:j] \text{ remains unchanged*}
\]

\[
\text{ELSE} \quad ; \text{zeroing-masking}
\]

\[
\quad \text{DEST}[i+63:j] := 0
\]

\[
\quad \text{FI}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VCVTQQ2PD } \_\_m512d \_\_mm512\_cvtepi64\_pd(\_\_m512i \_a);
\]

\[
\text{VCVTQQ2PD } \_\_m512d \_\_mm512\_mask\_cvtepi64\_pd(\_\_m512d \_s, \_\_mmask16 \_k, \_\_m512i \_a);
\]

\[
\text{VCVTQQ2PD } \_\_m512d \_\_mm512\_maskz\_cvtepi64\_pd(\_\_mmask16 \_k, \_\_m512i \_a);
\]

\[
\text{VCVTQQ2PD } \_\_m512d \_\_mm512\_cvt\_roundepi64\_pd(\_\_m512i \_a, \_\_int \_r);
\]

\[
\text{VCVTQQ2PD } \_\_m512d \_\_mm512\_mask\_cvt\_roundepi64\_pd(\_\_m512d \_s, \_\_mmask8 \_k, \_\_m512i \_a, \_\_int \_r);
\]

\[
\text{VCVTQQ2PD } \_\_m512d \_\_mm512\_maskz\_cvt\_roundepi64\_pd(\_\_mmask8 \_k, \_\_m512i \_a, \_\_int \_r);
\]

\[
\text{VCVTQQ2PD } \_\_m256d \_\_mm256\_mask\_cvtepi64\_pd(\_\_m256d \_s, \_\_mmask8 \_k, \_\_m256i \_a);
\]

\[
\text{VCVTQQ2PD } \_\_m256d \_\_mm256\_maskz\_cvtepi64\_pd(\_\_mmask8 \_k, \_\_m256i \_a);
\]

\[
\text{VCVTQQ2PD } \_\_m128d \_\_mm\_mask\_cvtepi64\_pd(\_\_m128d \_s, \_\_mmask8 \_k, \_\_m128i \_a);
\]

\[
\text{VCVTQQ2PD } \_\_m128d \_\_mm\_maskz\_cvtepi64\_pd(\_\_mmask8 \_k, \_\_m128i \_a);
\]

**SIMD Floating-Point Exceptions**

**Precision.**

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

\#UD \quad \text{If } \text{EVEX.}\_\_v\_\_v\_\_v !== 1111B.
VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, xmm2/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert two packed signed quadword integers in xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, ymm2/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed signed quadword integers in ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W1 5B /r VCVTQQ2PH xmm1{k1}{z}, zmm2/m512/m64bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed signed quadword integers in zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts packed signed quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

**VCVTQQ2PH DEST, SRC**

VL = 128, 256 or 512

KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.qword[0]
        ELSE
            tsrc := SRC.qword[j]
        DEST.fp16[j] := Convert_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PH __m128h _mm512_cvt_roundepi64_ph (__m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_mask_cvt_roundepi64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm512_maskz_cvt_roundepi64_ph (__mmask8 k, __m512i a, int rounding);
VCVTQQ2PH __m128h _mm_cvtepi64_ph (__m128i a);
VCVTQQ2PH __m128h _mm_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm_maskz_cvtepi64_ph (__mmask8 k, __m128i a);
VCVTQQ2PH __m128h _mm256_cvtepi64_ph (__m256i a);
VCVTQQ2PH __m128h _mm256_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm256_maskz_cvtepi64_ph (__mmask8 k, __m256i a);
VCVTQQ2PH __m128h _mm512_cvtepi64_ph (__m512i a);
VCVTQQ2PH __m128h _mm512_mask_cvtepi64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTQQ2PH __m128h _mm512_maskz_cvtepi64_ph (__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions
Overflow, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values

**Opcode/ Instruction** | **Op / En** | **64/32 Bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVEX.128.0F.W1 5B /r | A | V/V | AVX512VL AVX512DQ | Convert two packed quadword integers from xmm2/mem to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W1 5B /r | A | V/V | AVX512VL AVX512DQ | Convert four packed quadword integers from ymm2/mem to packed single precision floating-point values in xmm1 with writemask k1.
EVEX.512.0F.W1 5B /r | A | V/V | AVX512DQ | Convert eight packed quadword integers from zmm2/mem to eight packed single precision floating-point values in ymm1 with writemask k1.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts packed quadword integers in the source operand (second operand) to packed single precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[k+31:k] := Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[k+31:k] remains unchanged*
        ELSE ; zeroing-masking
            DEST[k+31:k] := 0
        FI
    FI
ENDFOR

DEST[MAXVL-1:VL/2] := 0
VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[k+31:k] :=
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[k+31:k] remains unchanged*
      ELSE ; zeroing-masking
        DEST[k+31:k] := 0
      FI
    FI
ENDFOR

DEST[MAXVL-1:VL/2] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PS __m256 _mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 _mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 _mm512_mask_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 _mm256_cvtepi64_ps( __m256i a);
VCVTQQ2PS __m128 _mm256_mask_cvtepi64_ps( __m256 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 _mm_maskz_cvtepi64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.
VCVTSD2SH—Convert Low FP64 Value to an FP16 Value

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts the low FP64 value in the second source operand to an FP16 value, and stores the result in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

\[ \text{VCVTSD2SH} \text{ dest, src1, src2} \]

- IF *SRC2 is a register* and (EVEX.b = 1):
  - SET\_RM(EVEX.RC)
- ELSE:
  - SET\_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:

\[ \text{DEST.fp16[0]} := \text{Convert\_fp64\_to\_fp16(SRC2.fp64[0])} \]

ELSE IF *zeroing*:

\[ \text{DEST.fp16[0]} := 0 \]

// else dest.fp16[0] remains unchanged

\[ \text{DEST[127:16]} := \text{SRC1[127:16]} \]

\[ \text{DEST[MAXVL-1:128]} := 0 \]

### Intel C/C++ Compiler Intrinsic Equivalent

- VCVTSD2SH __m128h _mm_cvt_roundsd_sh (__m128h a, __m128d b, const int rounding);
- VCVTSD2SH __m128h _mm_mask_cvt_roundsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b, const int rounding);
- VCVTSD2SH __m128h _mm_maskz_cvt_roundsd_sh (__mmask8 k, __m128d b);
- VCVTSD2SH __m128h _mm_maskz_cvtsd_sh (__mmask8 k, __m128h a, __m128d b);
- VCVTSD2SH __m128h _mm_cvtsd_sh (__m128h a, __m128d b);
- VCVTSD2SH __m128h _mm_mask_cvtsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b);
- VCVTSD2SH __m128h _mm_maskz_cvtsd_sh (__mmask8 k, __m128h a, __m128d b);

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F2OF.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.</td>
</tr>
<tr>
<td>EVEX.LIG.F2OF.W1 79 /r</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
Converts a double precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

Operation
VCVTSD2USI (EVEX Encoded Version)
IF (SRC *is register*) AND (EVEX.b = 1)
THEN
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF 64-Bit Mode and OperandSize = 64
THEN DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
ELSE DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvtsd_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64_mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64_mm_cvtsd_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions
Invalid, Precision.
Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
VCVTSH2SD—Convert Low FP16 Value to an FP64 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F3.MAP5.W0 5A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert the low FP16 value in xmm3/m16 to an FP64 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:64 of xmm2 are copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts the low FP16 element in the second source operand to a FP64 element in the low element of the destination operand.

Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP64 element of the destination is updated according to the writemask.

**Operation**

VCVTSH2SD dest, src1, src2

IF k1[0] OR *no writemask*:
  DEST.fp64[0] := Convert_fp16_to_fp64(SRC2.fp16[0])
ELSE IF *zeroing*:
  DEST.fp64[0] := 0
// else dest.fp64[0] remains unchanged

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSH2SD __m128d __m128d_mm_cvt_roundsh_sd (__m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d __m128d_mm_mask_cvt_roundsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d __m128d_mm_maskz_cvt_roundsh_sd (__mmask8 k, __m128d a, __m128h b, const int sae);
VCVTSH2SD __m128d __m128d_mm_cvtsh_sd (__m128d a, __m128h b);
VCVTSH2SD __m128d __m128d_mm_mask_cvtsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b);
VCVTSH2SD __m128d __m128d_mm_maskz_cvtsh_sd (__mmask8 k, __m128d a, __m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
VCVTSH2SI—Convert Low FP16 Value to Signed Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 2D /r VCVTSH2SI r32, xmm1/m16 (er)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert the low FP16 element in xmm1/m16 to a signed integer and store the result in r32.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAP5.W1 2D /r VCVTSH2SI r64, xmm1/m16 (er)</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512-FP16</td>
<td>Convert the low FP16 element in xmm1/m16 to a signed integer and store the result in r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

Operation

VCVTSH2SI dest, src
IF *SRC is a register* and (EVEX.b = 1):
  SET_RM(EVEX.RC)
ELSE:
  SET_RM(MXCSR.RC)
IF 64-mode and OperandSize == 64:
  DEST.qword := Convert_fp16_to_integer64(SRC.fp16[0])
ELSE:
  DEST.dword := Convert_fp16_to_integer32(SRC.fp16[0])

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSH2Si int _mm_cvt_roundsh_i32 (__m128h a, int rounding);
VCVTSH2Si __int64 _mm_cvt_roundsh_i64 (__m128h a, int rounding);
VCVTSH2Si int _mm_cvtsh_i32 (__m128h a);
VCVTSH2Si __int64 _mm_cvtsh_i64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
VCVTSH2SS—Convert Low FP16 Value to FP32 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP.MAP6.W0 13 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert the low FP16 element in xmm3/m16 to an FP32 value and store in the low element of xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts the low FP16 element in the second source operand to the low FP32 element of the destination operand.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

**Operation**

VCVTSH2SS dest, src1, src2

IF k1[0] OR *no writemask*:
  DEST.fp32[0] := Convert_fp16_to_fp32(SRC2.fp16[0])
ELSE IF *zeroing*:
  DEST.fp32[0] := 0
// else dest.fp32[0] remains unchanged


DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSH2SS __m128 __mm_cvt_roundsh_ss (__m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 __mm_mask_cvt_roundsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 __mm_maskz_cvt_roundsh_ss (__mmask8 k, __m128 a, __m128h b, const int sae);
VCVTSH2SS __m128 __mm_cvtsh_ss (__m128 a, __m128h b);
VCVTSH2SS __m128 __mm_mask_cvtsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b);
VCVTSH2SS __m128 __mm_maskz_cvtsh_ss (__mmask8 k, __m128 a, __m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

---

VCVTSH2SS—Convert Low FP16 Value to FP32 Value
VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer

<table>
<thead>
<tr>
<th>Opcode/Operation</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 79 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert the low FP16 element in xmm1/m16 to an unsigned integer and store the result in r32.</td>
</tr>
<tr>
<td>VCVTSH2USI r32, xmm1/m16 [er]</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512-FP16</td>
<td>Convert the low FP16 element in xmm1/m16 to an unsigned integer and store the result in r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

### Operation

```
VCVTSH2USI dest, src
```

// SET_RM() sets the rounding mode used for this instruction.

IF *SRC is a register* and (EVEX.b = 1):
  SET_RM(EVEX.RC)
ELSE:
  SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:
  DEST.qword := Convert_fp16_to_unsigned_integer64(SRC.fp16[0])
ELSE:
  DEST.dword := Convert_fp16_to_unsigned_integer32(SRC.fp16[0])

### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSH2USI unsigned int __mm_cvt_roundsh_u32 (__m128h a, int sae);
VCVTSH2USI unsigned __int64 __mm_cvt_roundsh_u64 (__m128h a, int rounding);
VCVTSH2USI unsigned int __mm_cvtsh_u32 (__m128h a);
VCVTSH2USI unsigned __int64 __mm_cvtsh_u64 (__m128h a);
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
VCVTSI2SH—Convert a Signed Doubleword/Quadword Integer to an FP16 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 2A/r VCVTSI2SH xmm1, xmm2, r32/m32 {er}</td>
<td>A/VV</td>
<td>AVX512-FP16</td>
<td>Convert the signed doubleword integer in r32/m32 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAP5.W1 2A/r VCVTSI2SH xmm1, xmm2, r64/m64 {er}</td>
<td>A/V.N.E.</td>
<td>AVX512-FP16</td>
<td>Convert the signed quadword integer in r64/m64 to an FP16 value and store the result in xmm1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to an FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand. Bits MAXVL-1:128 of the destination register are zeroed.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

**VCVTSI2SH dest, src1, src2**

IF *SRC2 is a register* and (EVEX.b = 1):

1. SET_RM(EVEX.RC)
2. ELSE:
   1. SET_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:

1. DEST.fp16[0] := Convert_integer64_to_fp16(SRC2.qword)
2. ELSE:
   1. DEST.fp16[0] := Convert_integer32_to_fp16(SRC2.dword)

DEST[MAXVL-1:128] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTSI2SH _m128h _mm_cvt_roundi32_sh (__m128h a, int b, int rounding);
VCVTSI2SH _m128h _mm_cvt_roundi64_sh (__m128h a, __int64 b, int rounding);
VCVTSI2SH _m128h _mm_cvti32_sh (__m128h a, int b);
VCVTSI2SH _m128h _mm_cvti64_sh (__m128h a, __int64 b);
```
SIMD Floating-Point Exceptions
Overflow, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
VCVTSS2SH—Convert Low FP32 Value to an FP16 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP.MAP5,W0 1D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert low FP32 value in xmm3/m32 to an FP16 value and store in the low element of xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts the low FP32 value in the second source operand to a FP16 value in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

**Operation**

VCVTSS2SH dest, src1, src2

IF *SRC2 is a register* and (EVEX.b = 1):

- SET_RM(EVEX.RC)

ELSE:

- SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:

- DEST.fp16[0] := Convert_fp32_to_fp16(SRC2.fp32[0])

ELSE IF *zeroing*:

- DEST.fp16[0] := 0

// else dest.fp16[0] remains unchanged

DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SH __m128h __mm_cvt_roundss_sh (__m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h __mm_mask_cvt_roundss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h __mm_maskz_cvt_roundss_sh (__mmask8 k, __m128h a, __m128 b, const int rounding);
VCVTSS2SH __m128h __mm_cvtss_sh (__m128h a, __m128 b);
VCVTSS2SH __m128h __mm_mask_cvtss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b);
VCVTSS2SH __m128h __mm_maskz_cvtss_sh (__mmask8 k, __m128h a, __m128 b);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
VCVTSS2USI—Convert Scalar Single Precision Floating-Point Value to Unsigned Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}</td>
<td>A</td>
<td>V/N.E.1</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. EVEX.W1 in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**
Converts a single precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w − 1$ is returned, where $w$ represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VCVTSS2USI (EVEX Encoded Version)**

IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
ELSE
    DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2USI unsigned _mm_cvtsu_u32( __m128 a);
VCVTSS2USI unsigned _mm_cvt.roundss_u32( __m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtsu_u64( __m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64( __m128 a, int r);
**SIMD Floating-Point Exceptions**
Invalid, Precision.

**Other Exceptions**
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
VCVTTPD2QQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed double precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^w-1\), where \(w\) represents the number of bits in the destination format) is returned.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

\(KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0\) TO \(KL-1\)

\(i := j \times 64\)

IF \(k1[j] OR *no writemask*\) THEN \(DEST[i+63:j] := Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger\_Truncate(SRC[i+63:j])\)

ELSE

IF *merging-mask* ; merging-masking

THEN *DEST[i+63:j] remains unchanged* 
ELSE ; zeroing-masking

\(DEST[i+63:j] := 0\)

FI

ENDFOR

\(DEST[MAXVL-1:VL] := 0\)
VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Memory Source

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j \times 64 \\
\quad \text{IF } k1[j] \text{ OR } \text{"no writemask"} \\
\qquad \text{THEN} \\
\qquad \quad \text{IF } (EVEX.b == 1) \\
\qquad \qquad \text{DEST}[i+63:i] := \text{Convert Double-Precision Floating-Point To QuadInteger Truncate}(\text{SRC}[63:0]) \\
\qquad \quad \text{ELSE} \\
\qquad \qquad \text{DEST}[i+63:i] := \text{Convert Double-Precision Floating-Point To QuadInteger Truncate}(\text{SRC}[i+63:i]) \\
\qquad \text{FI;} \\
\quad \text{ELSE} \\
\qquad \text{IF } \text{"merging-masking"} \quad ; \text{merging-maskng} \\
\qquad \quad \text{THEN } *\text{DEST}[i+63:i] \text{ remains unchanged"} \\
\qquad \quad \text{ELSE } \quad ; \text{zeroing-masking} \\
\qquad \qquad \text{DEST}[i+63:i] := 0 \\
\quad \text{FI} \\
\text{ENDIF}
\]

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VCVTTPD2QQ } \_\text{mm512i } _\text{mm512_cvttd_epi64}(\_\text{m512d } a); \\
\text{VCVTTPD2QQ } \_\text{mm512i } _\text{mm512_mask_cvttd_epi64}(\_\text{m512i } s, \_\text{mmask8 } k, \_\text{m512d } a); \\
\text{VCVTTPD2QQ } \_\text{mm512i } _\text{mm512_maskz_cvttd_epi64}(\_\text{mmask8 } k, \_\text{m512d } a); \\
\text{VCVTTPD2QQ } \_\text{m512i } _\text{mm512_cvt_roundpd_epi64}(\_\text{m512d } a, \text{ int } sae); \\
\text{VCVTTPD2QQ } \_\text{m512i } _\text{mm512_mask_cvt_roundpd_epi64}(\_\text{m512i } s, \_\text{mmask8 } k, \_\text{m512d } a, \text{ int } sae); \\
\text{VCVTTPD2QQ } \_\text{m512i } _\text{mm512_maskz_cvt_roundpd_epi64}(\_\text{mmask8 } k, \_\text{m512d } a, \text{ int } sae); \\
\text{VCVTTPD2QQ } \_\text{m256i } _\text{mm256_mask_cvt_roundpd_epi64}(\_\text{m256i } s, \_\text{mmask8 } k, \_\text{m256d } a); \\
\text{VCVTTPD2QQ } \_\text{m256i } _\text{mm256_maskz_cvt_roundpd_epi64}(\_\text{mmask8 } k, \_\text{m256d } a); \\
\text{VCVTTPD2QQ } \_\text{m128i } _\text{mm_mask_cvt_roundpd_epi64}(\_\text{m128i } s, \_\text{mmask8 } k, \_\text{m128d } a); \\
\text{VCVTTPD2QQ } \_\text{m128i } _\text{mm_maskz_cvt_roundpd_epi64}(\_\text{mmask8 } k, \_\text{m128d } a);
\]

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:

\#UD \quad \text{If EVEX.vvvv !=} 1111B.
## VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W1 78 02 /r VCVTTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 78 /r VCVTTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts with truncation packed double precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTTPD2UDQ (EVEX Encoded Versions) When SRC2 Operand is a Register**

($KL, VL) = (2, 128), (4, 256), (8, 512)$

FOR $j := 0$ TO $KL-1$

$i := j \times 32$

$k := j \times 64$

IF $k1[j] \text{ OR *no writemask*}$

THEN

\[
\text{DEST}[i+31:i] := \text{Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])}
\]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+31:i] remains unchanged*

ELSE ; zeroing-masking

\[
\text{DEST}[i+31:i] := 0
\]

FI

FI
VCVTTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
          ELSE
          DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-mask* ; merging-mask
          THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-mask
            DEST[i+31:i] := 0
          FI
        FI
  END FOR
DEST[MAXVL-1:VL/2] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTDPD2UDQ __m256i _mm256_cvtt_roundpd_epu32(__m512d a);
VCVTTDPD2UDQ __m256i _mm256_mask_cvtt_roundpd_epu32(__m512d s, __mmask8 k, __m512d a);
VCVTTDPD2UDQ __m256i _mm256_maskz_cvtt_roundpd_epu32(__mmask8 k, __m512d a);
VCVTTDPD2UDQ __m256i _mm256_cvtt_roundpd_epu32(__m512d a, int sae);
VCVTTDPD2UDQ __m256i _mm256_mask_cvtt_roundpd_epu32(__m256d s, __mmask8 k, __m512d a, int sae);
VCVTTDPD2UDQ __m256i _mm256_maskz_cvtt_roundpd_epu32(__mmask8 k, __m512d a, int sae);
VCVTTDPD2UDQ __m128i _mm256_mask_roundpd_epu32(__m256i s, __mmask8 k, __m256d a);
VCVTTDPD2UDQ __m128i _mm_mask_roundpd_epu32(__m128d s, __mmask8 k, __m128d a);
VCVTTDPD2UDQ __m128i _mm_maskz_roundpd_epu32(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:
#UD If EVEX.vvvv != 1111B.
VCVTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1/78 EVX</td>
<td>A/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1/78 EVX</td>
<td>A/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1/78 EVX</td>
<td>A/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts with truncation packed double precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

**Note:** EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register**

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

FOR $j := 0$ TO $KL-1$

$i := j \times 64$

IF $k1[j]$ OR *no writemask* THEN

DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

Fi

ENDFOR

DEST[MAXVL-1:VL] := 0
VCVTTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

\( (KL, VL) = (2, 128), (4, 256), (8, 512) \)

FOR \( j := 0 \) TO \( KL-1 \)
    \( i := j \times 64 \)
    IF \( k1[j] \) OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    \( \text{DEST}[i+63:j] := \) Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
                ELSE
                    \( \text{DEST}[i+63:j] := \) Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:j])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged* 
            ELSE ; zeroing-masking
                \( \text{DEST}[i+63:j] := 0 \)
            FI
        FI;
    FI;
ENDFOR
\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)

Intel C/C++ Compiler Intrinsic Equivalent

\begin{align*}
\text{VCVTTPD2UQQ} \_mm<\text{size}\_\_\text{mask}[z]\_\_\text{cvtt[round]}pd\_\_\text{epu64} \\
\text{VCVTTPD2UQQ} \_\_\text{m512i}\_\_\text{mm512}\_\_\text{cvttpd}\_\_\text{epu64}\_\_\text{m512d a} \\
\text{VCVTTPD2UQQ} \_\_\text{m512i}\_\_\text{mm512}\_\_\text{mask}\_\_\text{cvttpd}\_\_\text{epu64}\_\_\text{m512i s}\_\_\text{mmask8 k}\_\_\text{m512d a} \\
\text{VCVTTPD2UQQ} \_\_\text{m512i}\_\_\text{mm512}\_\_\text{maskz}\_\_\text{cvtt}\_\_\text{roundpd}\_\_\text{epu64}\_\_\text{m512d a, int sae} \\
\text{VCVTTPD2UQQ} \_\_\text{m512i}\_\_\text{mm512}\_\_\text{maskz}\_\_\text{cvtt}\_\_\text{roundpd}\_\_\text{epu64}\_\_\text{m512i s}\_\_\text{mmask8 k}\_\_\text{m512d a, int sae} \\
\text{VCVTTPD2UQQ} \_\_\text{m256i}\_\_\text{mm256}\_\_\text{mask}\_\_\text{cvtt}\_\_\text{roundpd}\_\_\text{epu64}\_\_\text{m256d a} \\
\text{VCVTTPD2UQQ} \_\_\text{m256i}\_\_\text{mm256}\_\_\text{maskz}\_\_\text{cvtt}\_\_\text{roundpd}\_\_\text{epu64}\_\_\text{m256d a} \\
\text{VCVTTPD2UQQ} \_\_\text{m128i}\_\_\text{mm128}\_\_\text{mask}\_\_\text{cvtt}\_\_\text{epu64}\_\_\text{m128d a} \\
\text{VCVTTPD2UQQ} \_\_\text{m128i}\_\_\text{mm128}\_\_\text{maskz}\_\_\text{cvtt}\_\_\text{epu64}\_\_\text{m128d a} \\
\end{align*}

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

\#UD \quad \text{If EVEX.vvvv != 1111B.}
VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.MAP5.W0 5B /r VCVTTPH2DQ xmm1[k1]{k}[z], xmm2/m64/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert four packed FP16 values in xmm2/m64/m16bcst to four signed doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.MAP5.W0 5B /r VCVTTPH2DQ ymm1[k1]{k}[z], xmm2/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.MAP5.W0 5B /r VCVTTPH2DQ zmm1[k1]{k}[z], xmm2/m256/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert sixteen packed FP16 values in xmm2/m256/m16bcst to sixteen signed doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

Operation
VCVTTPH2DQ dest, src
VL = 128, 256 or 512
KL := VL / 32

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsr := SRC.fp16[0]
        ELSE
            tsr := SRC.fp16[j]
        END_IF
        DEST.fp32[j] := Convert_fp16_to_integer32_truncate(tsr)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged
DEST[MAXVL-1:VL] := 0
VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2DQ __m512i _mm512_cvtt_roundph_epi32 (__m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_mask_cvtt_roundph_epi32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m512i _mm512_maskz_cvtt_roundph_epi32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2DQ __m128i _mm_cvttph_epi32 (__m256h a);
VCVTTPH2DQ __m128i _mm_mask_cvttph_epi32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m128i _mm_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_cvttph_epi32 (__m128h a);
VCVTTPH2DQ __m256i _mm256_mask_cvttph_epi32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2DQ __m256i _mm256_maskz_cvttph_epi32 (__mmask8 k, __m128h a);
VCVTTPH2DQ __m512i _mm512_cvttph_epi32 (__m256h a);
VCVTTPH2DQ __m512i _mm512_mask_cvttph_epi32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2DQ __m512i _mm512_maskz_cvttph_epi32 (__mmask16 k, __m256h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTTPH2QQ—Convert with Truncation Packed FP16 Values to Signed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP5.w0 7A /r VCVTTPH2QQ xmm1[k1][z], xmm2/m32/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert two packed FP16 values in xmm2/m32/m16bcst to two signed quadword integers, and store the result in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAP5.w0 7A /r VCVTTPH2QQ ymm1[k1][z], xmm2/m64/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed FP16 values in xmm2/m64/m16bcst to four signed quadword integers, and store the result in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAP5.w0 7A /r VCVTTPH2QQ zmm1[k1][z], xmm2/m128/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed FP16 values in xmm2/m128/m16bcst to eight signed quadword integers, and store the result in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Quarter</td>
<td>ModRMreg(w)</td>
<td>ModRMr/m(r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed FP16 values in the source operand to signed quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

**Operation**

```
VCVTTPH2QQ dest, src
VL = 128, 256 or 512
KL := VL / 64

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.qword[j] := Convert_fp16_to_integer64_truncate(tsrc)
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```
Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2QQ __m512i _mm512_cvtt_roundph_epi64 (__m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_mask_cvtt_roundph_epi64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTTPH2QQ __m512i _mm512_maskz_cvtt_roundph_epi64 (__mmask8 k, __m128h a, int sae);
VCVTTHQ2QQ __m128i _mm_cvttph_epi64 (__m128h a);
VCVTTHQ2QQ __m128i _mm_mask_cvttph_epi64 (__m128i src, __mmask8 k, __m128h a);
VCVTTHQ2QQ __m128i _mm_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTHQ2QQ __m256i _mm256_cvttph_epi64 (__m128h a);
VCVTTHQ2QQ __m256i _mm256_mask_cvttph_epi64 (__m256i src, __mmask8 k, __m128h a);
VCVTTHQ2QQ __m256i _mm256_maskz_cvttph_epi64 (__mmask8 k, __m128h a);
VCVTTHQ2QQ __m512i _mm512_cvttph_epi64 (__m128h a);
VCVTTHQ2QQ __m512i _mm512_mask_cvttph_epi64 (__m512i src, __mmask8 k, __m128h a);
VCVTTHQ2QQ __m512i _mm512_maskz_cvttph_epi64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
### VCVTTPH2UDQ—Convert with Truncation Packed FP16 Values to Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5,W0 78 /r VCVTTPH2UDQ xmm1{k1}[z], xmm2/m64/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed FP16 values in xmm2/m64/m16bcst to four unsigned doubleword integers, and store the result in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5,W0 78 /r VCVTTPH2UDQ ymm1{k1}[z], xmm2/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed FP16 values in xmm2/m128/m16bcst to eight unsigned doubleword integers, and store the result in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5,W0 78 /r VCVTTPH2UDQ zmm1{k1}[z], ymm2/m256/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen unsigned doubleword integers, and store the result in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2UDQ dest, src**

VL = 128, 256 or 512

KL := VL / 32

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF *SRC is memory* and EVEX.b = 1:
      tsrc := SRC.fp16(0)
    ELSE
      tsrc := SRC.fp16[j]
    DEST.dword[j] := Convert_fp16_to_unsigned_integer32_truncate(tsrc)
  ELSE IF *zeroing*:
    DEST.dword[j] := 0
    // else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTPH2UDQ __m512i _mm512_cvtt_roundph_epu32 (__m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_mask_cvtt_roundph_epu32 (__m512i src, __mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m512i _mm512_maskz_cvtt_roundph_epu32 (__mmask16 k, __m256h a, int sae);
VCVTTPH2UDQ __m128i _mm_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m128i _mm_mask_cvttph_epu32 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m128i _mm_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_cvttph_epu32 (__m128h a);
VCVTTPH2UDQ __m256i _mm256_mask_cvttph_epu32 (__m256i src, __mmask8 k, __m128h a);
VCVTTPH2UDQ __m256i _mm256_maskz_cvttph_epu32 (__mmask8 k, __m128h a);
VCVTTPH2UDQ __m512i _mm512_cvttph_epu32 (__m512i src, __mmask16 k, __m256h a);
VCVTTPH2UDQ __m512i _mm512_mask_cvttph_epu32 (__mmask16 k, __m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
VCVTTPH2UQQ—Convert with Truncation Packed FP16 Values to Unsigned Quadword Integers

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP5,W0 78 /r VCVTTPH2UQQ xmm1[k1]{z}, xmm2/m32/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
</tr>
<tr>
<td>EVEX.256.66.MAP5,W0 78 /r VCVTTPH2UQQ ymm1[k1]{z}, xmm2/m64/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
</tr>
<tr>
<td>EVEX.512.66.MAP5,W0 78 /r VCVTTPH2UQQ zmm1[k1]{z}, xmm2/m128/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
</tr>
</tbody>
</table>

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Quarter</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

**Operation**

`VCVTTPH2UQQ dest, src`

\(VL = 128, 256\) or \(512\)

\(KL := \frac{VL}{64}\)

\(\text{FOR } j := 0 \text{ TO KL-1:}\)

\(\text{IF } k1[j]\text{ OR *no writemask*:}\)

\(\text{IF *SRC is memory* and EVEX.b = 1:}\)

\(\text{tsrc := SRC.fp16[0]}\)

\(\text{ELSE}\)

\(\text{tsrc := SRC.fp16[j]}\)

\(\text{DEST.qword[j]} := \text{Convert_fp16_to_unsigned_integer64_truncate(tsrc)}\)

\(\text{ELSE IF *zeroing*:}\)

\(\text{DEST.qword[j]} := 0\)

\(\text{// else dest.qword[j] remains unchanged}\)

\(\text{DEST[MAXVL-1:VL]} := 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2UQQ __m512i _mm512_cvtt_roundph_epu64 (__m128h a, int sae);
VCVTPH2UQQ __m512i _mm512_mask_cvtt_roundph_epu64 (__m512i src, __mmask8 k, __m128h a, int sae);
VCVTPH2UQQ __m512i _mm512_maskz_cvtt_roundph_epu64 (__mmask8 k, __m128h a, int sae);
VCVTPH2UQQ __m128i _mm_cvttph_epu64 (__m128h a);
VCVTPH2UQQ __m128i _mm_mask_cvttph_epu64 (__m128i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m128i _mm_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_cvttph_epu64 (__m128h a);
VCVTPH2UQQ __m256i _mm256_mask_cvttph_epu64 (__m256i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m256i _mm256_maskz_cvttph_epu64 (__mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_cvttph_epu64 (__m128h a);
VCVTPH2UQQ __m512i _mm512_mask_cvttph_epu64 (__m512i src, __mmask8 k, __m128h a);
VCVTPH2UQQ __m512i _mm512_maskz_cvttph_epu64 (__mmask8 k, __m128h a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
### VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0  7C /r VCVTTPH2UW xmm1{k1}{z}, xmm2/ m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed FP16 values in xmm2/ m128/m16bcst to eight unsigned word integers, and store the result in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0  7C /r VCVTTPH2UW ymm1{k1}{z}, ymm2/ m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert sixteen packed FP16 values in ymm2/ m256/m16bcst to sixteen unsigned word integers, and store the result in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0  7C /r VCVTTPH2UW zmm1{k1}{z}, zmm2/ m512/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert thirty-two packed FP16 values in zmm2/ m512/m16bcst to thirty-two unsigned word integers, and store the result in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed FP16 values in the source operand to unsigned word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

**Operation**

`VCVTTPH2UW dest, src`

VL = 128, 256 or 512
KL := VL / 16

FOR j := 0 TO KL-1:

IF k1[j] OR *no writemask*

IF *.SRC is memory* and EVEX.b = 1:

tsrc := SRC.fp16[0]
ELSE

  tsrc := SRC.fp16[j]

DEST.word[j] := Convert_fp16_to_unsigned_integer16_truncate(tsrc)

ELSE IF *zeroing*:

  DEST.word[j] := 0

// else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2UW __m512i __mm512_cvtt_roundph_epu16 (__m512h a, int sae);
VCVTTPH2UW __m512i __mm512_mask_cvtt_roundph_epu16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m512i __mm512_maskz_cvtt_roundph_epu16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2UW __m128i __mm_cvttph_epu16 (__m128h a);
VCVTTPH2UW __m128i __mm_mask_cvttph_epu16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2UW __m128i __mm_maskz_cvttph_epu16 (__mmask8 k, __m128h a);
VCVTTPH2UW __m256i __mm256_cvttph_epu16 (__m256h a);
VCVTTPH2UW __m256i __mm256_mask_cvttph_epu16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2UW __m256i __mm256_maskz_cvttph_epu16 (__mmask16 k, __m256h a);
VCVTTPH2UW __m512i __mm512_cvttph_epu16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2UW __m512i __mm512_mask_cvttph_epu16 (__mmask32 k, __m512h a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers

### Description

This instruction converts packed FP16 values in the source operand to signed word integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2W dest, src**

\(VL = 128, 256\) or 512

\(KL := VL / 16\)

\(\text{FOR } j := 0 \text{ TO } KL-1:\)

\(\text{IF } k1[j] \text{ OR } \ast \text{no writemask}\ast:\)

\(\text{IF } \ast\text{SRC is memory}\ast \text{ and EVEX.b } = 1:\)

\(\text{tsrc } := \text{SRC.fp16}[0]\)

\(\text{ELSE}\)

\(\text{tsrc } := \text{SRC.fp16}[j]\)

\(\text{DEST.word[j]} := \text{Convert_fp16_to_integer16_truncate(tsrc)}\)

\(\text{ELSE IF } \ast\text{zeroing}\ast:\)

\(\text{DEST.word[j]} := 0\)

\(// \text{ else dest.word[j]} \text{ remains unchanged}\)

\(\text{DEST[MAXVL-1:VL]} := 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPH2W __m512i _mm512_cvtt_roundph_epi16 (__m512h a, int sae);
VCVTTPH2W __m512i _mm512_mask_cvtt_roundph_epi16 (__m512i src, __mmask32 k, __m512h a, int sae);
VCVTTPH2W __m512i _mm512_maskz_cvtt_roundph_epi16 (__mmask32 k, __m512h a, int sae);
VCVTTPH2W __m128i _mm_cvttph_epi16 (__m128h a);
VCVTTPH2W __m128i _mm_mask_cvttph_epi16 (__m128i src, __mmask8 k, __m128h a);
VCVTTPH2W __m128i _mm_maskz_cvttph_epi16 (__mmask8 k, __m128h a);
VCVTTPH2W __m256i _mm256_cvttph_epi16 (__m256h a);
VCVTTPH2W __m256i _mm256_mask_cvttph_epi16 (__m256i src, __mmask16 k, __m256h a);
VCVTTPH2W __m256i _mm256_maskz_cvttph_epi16 (__mmask16 k, __m256h a);
VCVTTPH2W __m512i _mm512_cvttph_epi16 (__m512i src, __mmask32 k, __m512h a);
VCVTTPH2W __m512i _mm512_mask_cvttph_epi16 (__m512i src, __mmask32 k, __m512h a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed single precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a YMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0\) TO \(KL-1\)

\(i := j \times 64\)

\(k := j \times 32\)

IF \(k1[j]\) OR *no writemask*

THEN \(\text{DEST[i+63:i]} := \text{Convert Single Precision To QuadInteger Truncate(SRC[k+31:k])}\)

ELSE

IF *merging-masking* ; merging-masking

THEN \(\text{DEST[i+63:i]}\) remains unchanged*

ELSE ; zeroing-masking

\(\text{DEST[i+63:i]} := 0\)

FI

FI

ENDFOR

\(\text{DEST[MAXVL-1:VL]} := 0\)
VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] := Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] := Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
      FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
  ENDFOR
  DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2QQ __m512i _mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_maskz_cvtt_roundps_epi64( __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_cvttps_epi64( __m128 i, __mmask8 k, __m256 a);
VCVTTPS2QQ __m256i _mm256_mask_cvttps_epi64( __mmask8 k, __m256 a, __m128 i);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64( __mmask8 k, __m256 a, __m128 i);
VCVTTPS2QQ __m128i _mm_mask_cvttps_epi64( __m128i s, __mmask8 k, __m128 i);
VCVTTPS2QQ __m128i _mm_maskz_cvttps_epi64( __mmask8 k, __m128 i);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.
VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed single precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] :=
    Convert_Single_Precision_Floating_Point_ToUINTeger_Truncate(SRC[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
  FI
  Fi;
ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] :=
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
          FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
          FI
        FI;
  ENDFOR
  DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UDQ __m512i _mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvtt_roundps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.
VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}[z], xmm2/m64/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1 {k1}[z], xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 78 /r VCVTTPS2UQQ zmm1 {k1}[z], ymm2/m256/m32bcst{sae}</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts with truncation up to eight packed single precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*:
    THEN DEST[i+63:j] := Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:j] := 0
      FI
  Fi
ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTPS2UQQ (EVEX Encoded Versions) When SRCOperand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
   i := j * 64
   k := j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b == 1)
            THEN
               DEST[i+63:i] := Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
            ELSE
               DEST[i+63:i] := Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
            FI;
         ELSE
            IF *merging-masking* ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+63:i] := 0
            FI
         FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2UQQ_mm<size>[_mask][z]_cvtt[_round]ps_epu64
VCVTPS2UQQ__m512i__mm512_cvtps_epu64(__m256 a);
VCVTPS2UQQ__m512i__mm512_mask_cvtps_epu64(__m512i s, __mmask16 k, __m256 a);
VCVTPS2UQQ__m512i__mm512_maskz_cvtps_epu64(__mmask16 k, __m256 a);
VCVTPS2UQQ__m512i__mm512_cvtt_roundps_epu64(__m256 a, int sae);
VCVTPS2UQQ__m512i__mm512_mask_cvtt_roundps_epu64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTPS2UQQ__m512i__mm512_maskz_cvtt_roundps_epu64(__mmask16 k, __m256 a, int sae);
VCVTPS2UQQ__m256i__mm256_cvtps_epu64(__m256i s, __mmask8 k, __m128 a);
VCVTPS2UQQ__m256i__mm256_mask_cvtps_epu64(__mmask8 k, __m128 a);
VCVTPS2UQQ__m128i__mm_mask_cvtps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ__m128i__mm_maskz_cvtps_epu64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
Additionally:
#UD IF EVEX.vvvv != 1111B.
VCVTTSD2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F2.0F:W0 7B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.</td>
</tr>
<tr>
<td>VCVTTSD2USI r32, xmm1/m64{sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.F2.0F:W1 7B /r</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.</td>
</tr>
<tr>
<td>VCVTTSD2USI r64, xmm1/m64{sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts with truncation a double precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value \(2^w - 1\) is returned, where \(w\) represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Operation

VCVTTSD2USI (EVEX Encoded Version)

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
ELSE DEST[31:0] := Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
FI

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2USI unsigned int _mm_cvtt_sd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvtt_sd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."

5-118 Vol. 2C
VCVTTSH2SI—Convert with Truncation Low FP16 Value to a Signed Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 2C /r VCVTTSH2SI r32, xmm1/m16 {sae}</td>
<td>A</td>
<td>V/V¹</td>
<td>AVX512-FP16</td>
<td>Convert FP16 value in the low element of xmm1/m16 to a signed integer and store the result in r32 using truncation.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAP5.W1 2C /r VCVTTSH2SI r64, xmm1/m16 {sae}</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512-FP16</td>
<td>Convert FP16 value in the low element of xmm1/m16 to a signed integer and store the result in r64 using truncation.</td>
</tr>
</tbody>
</table>

NOTES:
1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/r (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

Operation

```
VCVTTSH2SI dest, src

IF 64-mode and OperandSize == 64:
    DEST.qword := Convert_fp16_to_integer64_truncate(SRC.fp16[0])
ELSE:
    DEST.dword := Convert_fp16_to_integer32_truncate(SRC.fp16[0])
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSH2SI int _mm_cvtt_roundsh_i32 (__m128h a, int sae);
VCVTTSH2SI __int64 _mm_cvtt_roundsh_i64 (__m128h a, int sae);
VCVTTSH2SI int _mm_cvttsh_i32 (__m128h a);
VCVTTSH2SI __int64 _mm_cvttsh_i64 (__m128h a);
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
VCVTTSH2USI—Convert with Truncation Low FP16 Value to an Unsigned Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIF3.MAP5.W0</td>
<td>A</td>
<td>V/V¹</td>
<td>AVX512-FP16</td>
<td>Convert FP16 value in the low element of xmm1/m16 to an unsigned integer and store the result in r32 using truncation.</td>
</tr>
<tr>
<td>VCVTTSH2USI r32, xmm1/m16 {sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIF3.MAP5.W1</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512-FP16</td>
<td>Convert FP16 value in the low element of xmm1/m16 to an unsigned integer and store the result in r64 using truncation.</td>
</tr>
<tr>
<td>VCVTTSH2USI r64, xmm1/m16 {sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose register.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer indefinite value is returned.

Operation

VCVTTSH2USI dest, src

IF 64-mode and OperandSize == 64:
  DEST.qword := Convert_fp16_to_unsigned_integer64_truncate(SRC.fp16[0])
ELSE:
  DEST.dword := Convert_fp16_to_unsigned_integer32_truncate(SRC.fp16[0])

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSH2USI unsigned int _mm_cvtt_roundsh_u32 (__m128h a, int sae);
VCVTTSH2USI unsigned __int64 __mm_cvtt_roundsh_u64 (__m128h a, int sae);
VCVTTSH2USI unsigned int __mm_cvttsh_u32 (__m128h a);
VCVTTSH2USI unsigned __int64 __mm_cvttsh_u64 (__m128h a);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, ”Type E3NF Class Exception Conditions.”
VCVTTSS2USI—Convert With Truncation Scalar Single Precision Floating-Point Value to Unsigned Integer

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.</td>
</tr>
</tbody>
</table>

NOTES:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Fixed</td>
<td>ModRM{reg} (w)</td>
<td>ModRM{r/m} (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Converts with truncation a single precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTSS2USI (EVEX Encoded Version)
IF 64-bit Mode and OperandSize = 64
THEN
   DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
ELSE
   DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
FI;

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2USI unsigned int _mm_cvttss_u32(__m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32(__m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64(__m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64(__m128 a, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."

Vol. 2C 5-121
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j * 64
k := j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

ENDFOR

DEST[MAXVL-1:VL] := 0
VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
   THEN
     IF (EVEX.b = 1)
        THEN
           DEST[i+63:i] :=
           Convert_UInteger_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
           DEST[i+63:i] :=
           Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
     FI;
   ELSE
     IF *merging-masking* ; merging-masking
       THEN *DEST[i+63:i] remains unchanged*
     ELSE ; zeroing-masking
        DEST[i+63:i] := 0
     FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PD __m512d _mm512_cvtepu32_pd(__m256i a);
VCVTUDQ2PD __m512d _mm512_mask_cvtepu32_pd(__m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d _mm512_maskz_cvtepu32_pd(__mmask8 k, __m256i a);
VCVTUDQ2PD __m256d _mm256_cvtepu32_pd(__m128i a);
VCVTUDQ2PD __m256d _mm256_mask_cvtepu32_pd(__m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d _mm256_maskz_cvtepu32_pd(__mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm128_cvtepu32_pd(__m128i a);
VCVTUDQ2PD __m128d _mm128_mask_cvtepu32_pd(__mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm128_maskz_cvtepu32_pd(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-51, ”Type E5 Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B.
VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values

**Opcode/Instruction** | **Op/En** | **64/32 Bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVEX.T28.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}[z], xmm2/m128/m32bcst | A | V/V | AVX512-FP16 AVX512VL | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

EVEX.Z56.F2.MAP5.W0 7A /r VCVTUDQ2PH xmm1{k1}[z], xmm2/m256/m32bcst | A | V/V | AVX512-FP16 AVX512VL | Convert eight packed unsigned doubleword integers from xmm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.

EVEX.Z12.F2.MAP5.W0 7A /r VCVTUDQ2PH ymm1{k1}[z], zmm2/m512/m32bcst {er} | A | V/V | AVX512-FP16 | Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed unsigned doubleword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

**Operation**

VCVTUDQ2PH dest, src
VL = 128, 256 or 512
KL := VL / 32

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_R(M(EVEX.RC))
ELSE:
    SET_R(M(MXCSR.RC))

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.dword[0]
        ELSE
            tsrc := SRC.dword[j]
        DEST.fp16[j] := Convert_unsigned_integer32_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/2] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PH __m256h _mm512_cvt_roundepu32_ph (__m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_mask_cvt_roundepu32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m256h _mm512_maskz_cvt_roundepu32_ph (__mmask16 k, __m512i a, int rounding);
VCVTUDQ2PH __m128h _mm_cvtepu32_ph (__m128i a);
VCVTUDQ2PH __m128h _mm_mask_cvtepu32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUDQ2PH __m128h _mm_maskz_cvtepu32_ph (__mmask8 k, __m128i a);
VCVTUDQ2PH __m256h _mm256_cvtepu32_ph (__m256i a);
VCVTUDQ2PH __m256h _mm256_mask_cvtepu32_ph (__m256h src, __mmask8 k, __m256i a);
VCVTUDQ2PH __m256h _mm256_maskz_cvtepu32_ph (__mmask8 k, __m256i a);
VCVTUDQ2PH __m512h _mm512_cvtepu32_ph (__m512i a);
VCVTUDQ2PH __m512h _mm512_mask_cvtepu32_ph (__m512h src, __mmask16 k, __m512i a);
VCVTUDQ2PH __m512h _mm512_maskz_cvtepu32_ph (__mmask16 k, __m512i a);

SIMD Floating-Point Exceptions
Overflow, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}z, xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single precision floating-point values in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}z, ymm2/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single precision floating-point values in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}z, zmm2/m512/m32bcst(er)</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts packed unsigned doubleword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

set_rounding_mode_for_this_instruction(EVEX.RC);

ELSE

set_rounding_mode_for_this_instruction(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN DEST[i:31] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:j])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i:31] remains unchanged* ; zeroing-masking

ELSE

DEST[i:31] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0
VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR "no writemask"
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+31:i] :=
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
      FI;
    ELSE
      IF "merging-masking" ; merging-masking
        THEN "DEST[i+31:i] remains unchanged"
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PS __m512 _mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 __mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_mask_cvt_roundepu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 __mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 __mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 __mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 __mm128_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 __mm128_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 __mm128_maskz_cvtepu32_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
Additionally:
#UD If EVEX.vvvv != 1111B.
**VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double precision floating-point values in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1 7A /r VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts packed unsigned quadword integers in the source operand (second operand) to packed double precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)
    THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

FOR j := 0 TO KL-1  
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Memory Source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] := Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+63:i] := Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm128_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d _mm128_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm128_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B.
**VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.T28.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1[k1]{z}, xmm2/ m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert two packed unsigned doubleword integers from xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.Z56.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1[k1]{z}, ymm2/ m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert four packed unsigned doubleword integers from ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.S12.F2.MAP5.W1 7A /r VCVTUQQ2PH xmm1[k1]{z}, zmm2/ m512/m64bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Convert eight packed unsigned doubleword integers from zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed unsigned quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

EVEVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

**Operation**

VCVTUQQ2PH dest, src
VL = 128, 256 or 512
KL := VL / 64

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
  SET_RM(EVEX.RC)
ELSE:
  SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF *SRC is memory* and EVEX.b = 1:
      tsrc := SRC.qword[0]
    ELSE
      tsrc := SRC.qword[j]
    DEST.fp16[j] := Convert_unsigned_integer64_to_fp16(tsrc)
    ELSE IF *zeroing*:
      DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
  DEST[MAXVL-1:VL/4] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUQQ2PH __m128h _mm512_cvtroundepu64_ph (__m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_mask_cvtroundepu64_ph (__m128h src, __mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm512_maskz_cvtroundepu64_ph (__mmask8 k, __m512i a, int rounding);
VCVTUQQ2PH __m128h _mm_cvtepu64_ph (__m128i a);
VCVTUQQ2PH __m128h _mm_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm_maskz_cvtepu64_ph (__mmask8 k, __m128i a);
VCVTUQQ2PH __m128h _mm256_cvtepu64_ph (__m256i a);
VCVTUQQ2PH __m128h _mm256_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm256_maskz_cvtepu64_ph (__mmask8 k, __m256i a);
VCVTUQQ2PH __m128h _mm512_cvtepu64_ph (__m512i a);
VCVTUQQ2PH __m128h _mm512_mask_cvtepu64_ph (__m128h src, __mmask8 k, __m512i a);
VCVTUQQ2PH __m128h _mm512_maskz_cvtepu64_ph (__mmask8 k, __m512i a);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
## VCVTUSI2SH—Convert Unsigned Doubleword Integer to an FP16 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 7B /r VCVTUSI2SH xmm1, xmm2, r32/m32 {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert an unsigned doubleword integer from r32/m32 to an FP16 value, and store the result in xmm1. Bits 127:16 from xmm2 are copied to xmm1[127:16].</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAP5.W1 7B /r VCVTUSI2SH xmm1, xmm2, r64/m64 {er}</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512-FP16</td>
<td>Convert an unsigned quadword integer from r64/m64 to an FP16 value, and store the result in xmm1. Bits 127:16 from xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand. Bits MAXVL-1:128 of the destination register are zeroed.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

```
VCVTUSI2SH dest, src1, src2
IF *SRC2 is a register* and (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)
IF 64-mode and OperandSize == 64:
    DEST.fp16[0] := Convert_unsigned_integer64_to_fp16(SRC2.qword)
ELSE:
    DEST.fp16[0] := Convert_unsigned_integer32_to_fp16(SRC2.dword)
DEST[MAXVL-1:128] := 0
```
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUSI2SH __m128h _mm_cvt_roundu32_sh (__m128h a, unsigned int b, int rounding);
VCVTUSI2SH __m128h _mm_cvt_roundu64_sh (__m128h a, unsigned __int64 b, int rounding);
VCVTUSI2SH __m128h _mm_cvtsi32_sh (__m128h a, unsigned int b);
VCVTUSI2SH __m128h _mm_cvtsi64_sh (__m128h a, unsigned __int64 b);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”
VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F.W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single precision floating-point values in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W1 7A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description
Converting packed unsigned quadword integers in the source operand (second operand) to single precision floating-point values in the destination operand (first operand).

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Register**

KL, VL = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 32
k := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] := Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 0
VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] := 
          Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] := 
          Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
  ENDFOR
  DEST[MAXVL-1:VL/2] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PS __m256_mm512_cvtepu64_ps(__m512i a);
VCVTUQQ2PS __m256_mm512_mask_cvtepu64_ps(__m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256_mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PS __m256_mm512_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PS __m256_mm512_mask_cvt_roundepu64_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256_mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128_mm256_cvtepu64_ps(__m256i a);
VCVTUQQ2PS __m128_mm256_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128_mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);
VCVTUQQ2PS __m128_mm256_mask_cvt_roundepu64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128_mm256_maskz_cvt_roundepu64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.

VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values  Vol. 2C  5-135
VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F2.W0.7B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one unsigned doubleword integer from r/m32 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VCVTUSI2SD xmm1, xmm2, r/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.F2.W1.7B /r</td>
<td>A</td>
<td>V/N.E.1</td>
<td>AVX512F</td>
<td>Convert one unsigned quadword integer from r/m64 to one double precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>VCVTUSI2SD xmm1, xmm2, r/m64{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

**EVEX.W1 version:** promotes the instruction to use 64-bit input value in 64-bit mode.

**EVEX.W0 version:** attempt to encode this instruction with EVEX embedded rounding is ignored.

### Operation

**VCVTUSI2SD (EVEX Encoded Version)**

IF (SRC2 *is register*) AND (EVEX.b = 1)
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

### Notes

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.
Intel C/C++ Compiler Intrinsic Equivalent
VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d _mm_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions
Precision.

Other Exceptions
See Table 2-48, “Type E3NF Class Exception Conditions” if W1; otherwise, see Table 2-59, “Type E10NF Class Exception Conditions.”
VCVTUSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.0F.W0 7B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W1 7B /r</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Converts one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

**EVEX.W1 version:** promotes the instruction to use 64-bit input value in 64-bit mode.

**Operation**

**VCVTUSI2SS (EVEX Encoded Version)**

IF (SRC2 *is register*) AND (EVEX.b = 1)  
THEN  
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  
ELSE  
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);  
FI;  
IF 64-Bit Mode And OperandSize = 64  
THEN  
DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);  
ELSE  
DEST[31:0] := Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);  
FI;  
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUSI2SS _m128 _mm_cvtsi32_ss( _m128 s, unsigned a);  
VCVTUSI2SS _m128 _mm_cvtsi64_ss( _m128 s, unsigned a, int r);  
VCVTUSI2SS _m128 _mm_cvtsi64_ss( _m128 s, unsigned __int64 a);  
VCVTUSI2SS _m128 _mm_cvtsi64_ss( _m128 s, unsigned __int64 a, int r);
SIMD Floating-Point Exceptions
Precision.

Other Exceptions
See Table 2-48, “Type E3NF Class Exception Conditions.”
**VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.MAP5.W0 7D /r VCVTUw2PH xmm1[k1][z], xmm2/ m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed unsigned word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.MAP5.W0 7D /r VCVTUw2PH ymm1[k1][z], ymm2/ m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert sixteen packed unsigned word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.MAP5.W0 7D /r VCVTUw2PH zmm1[k1][z], zmm2/ m512/m16bcst [er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert thirty-two packed unsigned word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM reg (w)</td>
<td>ModRM r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts packed unsigned word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

**Operation**

```plaintext
VCVTUW2PH dest, src
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE:
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.word[0]
        ELSE
            tsrc := SRC.word[j]
        DEST.fp16[j] := Convert_unsignd_integer16_to_fp16(tsrc)
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
```

5-140 Vol. 2C VCVTUw2PH—Convert Packed Unsigned Word Integers to FP16 Values
Intel C/C++ Compiler Intrinsic Equivalent

VCVTUW2PH __m512h _mm512_cvt_roundepu16_ph (__m512i a, int rounding);
VCVTUW2PH __m512h _mm512_mask_cvt_roundepu16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m512h _mm512_maskz_cvt_roundepu16_ph (__mmask32 k, __m512i a, int rounding);
VCVTUW2PH __m128h _mm_cvtph (__m128i a);
VCVTUW2PH __m128h _mm_mask_cvtph (__m128h src, __mmask8 k, __m128i a);
VCVTUW2PH __m128h _mm_maskz_cvtph (__mmask8 k, __m128i a);
VCVTUW2PH __m256h _mm256_cvtph (__m256i a);
VCVTUW2PH __m256h _mm256_mask_cvtph (__m256h src, __mmask16 k, __m256i a);
VCVTUW2PH __m256h _mm256_maskz_cvtph (__mmask16 k, __m256i a);
VCVTUW2PH __m512h _mm512_cvtph (__m512h src, __mmask32 k, __m512i a);
VCVTUW2PH __m512h _mm512_mask_cvtph (__m512h src, __mmask32 k, __m512i a);
VCVTUW2PH __m512h _mm512_maskz_cvtph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions
Overflow, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.MAP5.W0 7D /r VCVT2PH xmm1{kn}<a href="z">k</a>, xmm2/ m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert eight packed signed word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.MAP5.W0 7D /r VCVT2PH ymm1{kn}<a href="z">k</a>, ymm2/ m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Convert sixteen packed signed word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.MAP5.W0 7D /r VCVT2PH zmm1{kn}<a href="z">k</a>, zmm2/ m512/m16bcst [er]</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert thirty-two packed signed word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Description

This instruction converts packed signed word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls. The destination elements are updated according to the writemask.

Operation

VCVTW2PH dest, src
VL = 128, 256 or 512
KL := VL / 16

IF *SRC is a register* and (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE:
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *SRC is memory* and EVEX.b = 1:
         tsr := SRC.word[0]
      ELSE
         tsr := SRC.word[j]
      DEST.fp16[j] := Convert_integer16_to_fp16(tsr)
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTW2PH __m512h _mm512_cvtrounepi16_ph (__m512i a, int rounding);
VCVTW2PH __m512h _mm512_mask_cvtrounepi16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTW2PH __m512h _mm512_maskz_cvtrounepi16_ph (__mmask32 k, __m512i a, int rounding);
VCVTW2PH __m128h _mm_cvtepi16_ph (__m128i a);
VCVTW2PH __m128h _mm_mask_cvtepi16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTW2PH __m128h _mm_maskz_cvtepi16_ph (__mmask8 k, __m128i a);
VCVTW2PH __m256h _mm256_cvtepi16_ph (__m256i a);
VCVTW2PH __m256h _mm256_mask_cvtepi16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTW2PH __m256h _mm256_maskz_cvtepi16_ph (__mmask16 k, __m256i a);
VCVTW2PH __m512h _mm512_cvtepi16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTW2PH __m512h _mm512_mask_cvtepi16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTW2PH __m512h _mm512_maskz_cvtepi16_ph (__mmask32 k, __m512i a);

SIMD Floating-Point Exceptions
Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Op/En | Tuple Type | Operand 1 | Operand 2 | Operand 3 | Operand 4
---|---|---|---|---|---
A | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8

Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see “Tmp1” in Figure 5-8, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.

- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2 and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.

- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.
**Operation**

**VDBPSADBW (EVEX Encoded Versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Selection of quadruplets:

FOR I = 0 to VL step 128

\[
\begin{align*}
\text{TMP1}[\text{I+31}:\text{I}] & := \text{select (SRC2}[\text{I+127: I}], \text{imm8}[1:0]) \\
\text{TMP1}[\text{I+63}:\text{I+32}] & := \text{select (SRC2}[\text{I+127: I}], \text{imm8}[3:2]) \\
\text{TMP1}[\text{I+95}:\text{I+64}] & := \text{select (SRC2}[\text{I+127: I}], \text{imm8}[5:4]) \\
\text{TMP1}[\text{I+127}:\text{I+96}] & := \text{select (SRC2}[\text{I+127: I}], \text{imm8}[7:6])
\end{align*}
\]

END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64

\[
\begin{align*}
\text{TMP\_DEST}[\text{I+15}:\text{I}] & := \text{ABS}(\text{SRC1}[\text{I+7: I}] - \text{TMP1}[\text{I+7: I}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+15: I+8}] - \text{TMP1}[\text{I+15: I+8}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+23: I+16}] - \text{TMP1}[\text{I+23: I+16}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+31: I+24}] - \text{TMP1}[\text{I+31: I+24}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+47: I+40}] - \text{TMP1}[\text{I+47: I+40}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+55: I+48}] - \text{TMP1}[\text{I+55: I+48}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+63: I+56}] - \text{TMP1}[\text{I+63: I+56}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+71: I+64}] - \text{TMP1}[\text{I+71: I+64}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+79: I+72}] - \text{TMP1}[\text{I+79: I+72}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+87: I+80}] - \text{TMP1}[\text{I+87: I+80}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+95: I+88}] - \text{TMP1}[\text{I+95: I+88}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+103: I+96}] - \text{TMP1}[\text{I+103: I+96}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+111: I+104}] - \text{TMP1}[\text{I+111: I+104}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+119: I+112}] - \text{TMP1}[\text{I+119: I+112}]) + \\
& \quad \text{ABS}(\text{SRC1}[\text{I+127: I+120}] - \text{TMP1}[\text{I+127: I+120}])
\end{align*}
\]
INSTRUCTION SET REFERENCE, V

ABS(SRC1[I+23: I+16] - TMP1[I+23: I+16]) +
ABS(SRC1[I+31: I+24] - TMP1[I+31: I+24])

ABS(SRC1[I+15: I+8] - TMP1[I+23: I+16]) +
ABS(SRC1[I+23: I+16] - TMP1[I+31: I+24]) +
ABS(SRC1[I+31: I+24] - TMP1[I+39: I+32])

ABS(SRC1[I+47: I+40] - TMP1[I+31: I+24]) +
ABS(SRC1[I+55: I+48] - TMP1[I+39: I+32]) +
ABS(SRC1[I+63: I+56] - TMP1[I+47: I+40])

ABS(SRC1[I+47: I+40] - TMP1[I+39: I+32]) +
ABS(SRC1[I+55: I+48] - TMP1[I+47: I+40]) +
ABS(SRC1[I+63: I+56] - TMP1[I+55: I+48])

ENDFOR

FOR j := 0 TO KL-1
    i := j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] := TMP_DEST[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+15:i] := 0
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b int imm8);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
VDIVPH—Divide Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 5E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Divide packed FP16 values in xmm2 by packed FP16 values in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 5E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Divide packed FP16 values in ymm2 by packed FP16 values in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 5E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Divide packed FP16 values in zmm2 by packed FP16 values in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction divides packed FP16 values from the first source operand by the corresponding elements in the second source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

### Operation

**VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Register**

- VL = 128, 256 or 512
- KL := VL/16

1. IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
2. ELSE
   SET_RM(MXCSR.RC)

3. FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
     DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
   ELSE IF *zeroing*:
     DEST.fp16[j] := 0
     // else dest.fp16[j] remains unchanged

4. DEST[MAXVL-1:VL] := 0
VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[0]
    ELSE:
      DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPH __m128h _mm_div_ph (__m128h a, __m128h b);
VDIVPH __m128h _mm_mask_div_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVPH __m128h _mm_maskz_div_ph (__mmask8 k, __m128h a, __m128h b);
VDIVPH __m256h _mm256_div_ph (__m256h a, __m256h b);
VDIVPH __m256h _mm256_mask_div_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VDIVPH __m256h _mm256_maskz_div_ph (__mmask16 k, __m256h a, __m256h b);
VDIVPH __m512h _mm512_div_ph (__m512h a, __m512h b);
VDIVPH __m512h _mm512_mask_div_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_maskz_div_ph (__mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_div_round_ph (__m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_mask_div_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_maskz_div_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
VDIVSH—Divide Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 5E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Divide low FP16 value in xmm2 by low FP16 value in xmm3/m16, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction divides the low FP16 value from the first source operand by the corresponding value in the second source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

**VDIVSH (EVEX Encoded Versions)**

IF EVEX.b = 1 and SRC2 is a register:

- SET_RM(EVEX.RC)
ELSE

- SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:

- DEST.fp16[0] := SRC1.fp16[0] / SRC2.fp16[0]
ELSE IF *zeroing*:

- DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[MAXVL-1:128] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

VDIVSH __m128h_mm_div_round_sh (__m128h a, __m128h b, int rounding);
VDIVSH __m128h_mm_mask_div_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h_mm_maskz_div_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
VDIVSH __m128h_mm_div_sh (__m128h a, __m128h b);
VDIVSH __m128h_mm_mask_div_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVSH __m128h_mm_maskz_div_sh (__mmask8 k, __m128h a, __m128h b);

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

VDIVSH—Divide Scalar FP16 Values
VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Multiply BF16 pairs from xmm2 and xmm3/m128, and accumulate the resulting packed single precision results in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Multiply BF16 pairs from ymm2 and ymm3/m256, and accumulate the resulting packed single precision results in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_BF16</td>
<td>Multiply BF16 pairs from zmm2 and zmm3/m512, and accumulate the resulting packed single precision results in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD dot-product of two BF16 pairs and accumulates into a packed single precision register.

“Round to nearest even” rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

NaN propagation priorities are described in Table 5-1.

Table 5-1. NaN Propagation Priorities

<table>
<thead>
<tr>
<th>NaN Priority</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>src1 low is NaN</td>
<td>Lower part has priority over upper part, i.e., it overrides the upper part.</td>
</tr>
<tr>
<td>2</td>
<td>src2 low is NaN</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>src1 high is NaN</td>
<td>Upper part may be overridden if lower has NaN.</td>
</tr>
<tr>
<td>4</td>
<td>src2 high is NaN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>srcdest is NaN</td>
<td>Dest is propagated if no NaN is encountered by src2.</td>
</tr>
</tbody>
</table>

Operation

Define make_fp32(x):

// The x parameter is bfloat16. Pack it in to upper 16b of a dword. The bit pattern is a legal fp32 value. Return that bit pattern.

dword := 0
dword[31:16] := x
RETURN dword
VDPBF16PS srcdest, src1, src2
VL = (128, 256, 512)
KL = VL/32

origdest := srcdest
FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF src2 is memory and evex.b == 1:
      t := src2.dword[0]
    ELSE:
      t := src2.dword[i]
  // FP32 FMA with daz in, ftz out and RNE rounding. MXCSR neither consulted nor updated.
  srcdest.fp32[i] += make_fp32(src1.bfloat16[2*i+1]) * make_fp32(t.bfloat[1])
  srcdest.fp32[i] += make_fp32(src1.bfloat16[2*i+0]) * make_fp32(t.bfloat[0])
ELSE IF *zeroing*:
  srcdest.dword[i] := 0
ELSE: // merge masking, dest element unchanged
  srcdest.dword[i] := origdest.dword[i]

srcdest[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VDPBF16PS __m128 __m128_mm_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 __m128_mm_mask_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 __m128_mm_maskz_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m256 __m256_mm256_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 __m256_mm256_mask_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 __m256_mm256_maskz_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m512 __m512_mm512_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 __m512_mm512_mask_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 __m512_mm512_maskz_dpbf16_ps(__m512, __m512bh, __m512bh);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, "Type E4 Class Exception Conditions."
VERR/VERW—Verify a Segment for Reading or Writing

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /4 VERR r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set ZF=1 if segment specified with r/m16 can be read.</td>
</tr>
<tr>
<td>0F 00 /5 VERW r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set ZF=1 if segment specified with r/m16 can be written.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment’s DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector’s RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector’s value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

### Operation

IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF := 0; FI;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
    or (SegmentDescriptor(Type) ≠ conforming code segment)
    and (CPL > DPL) or (RPL > DPL)
    THEN
        ZF := 0;
    ELSE
        IF (Instruction = VERR) and (Segment readable))
        or ((Instruction = VERW) and (Segment writable))
            THEN
                ZF := 1;
            ELSE
                ZF := 0;
Flags Affected
The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions
The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD The VERR and VERW instructions are not recognized in real-address mode.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD The VERR and VERW instructions are not recognized in virtual-8086 mode.
If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.
VEXPANDPD—Load Sparse Packed Double Precision Floating-Point Values From Dense Memory

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 88 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VEXPANDPD xmm1 {k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 88 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VEXPANDPD ymm1 {k1}{z}, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 88 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VEXPANDPD zmm1 {k1}{z}, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Expand (load) up to 8/4/2, contiguous, double precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPD (EVEX Encoded Versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

k := 0

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] := SRC[k+63:k];
            k := k + 64
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    THEN DEST[i+63:i] := 0
            FI
    FI
ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPD __m512d _mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd( __mmask8 k, void * a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.
VEXPANDPS—Load Sparse Packed Single Precision Floating-Point Values From Dense Memory

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}[z], xmm2/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}[z], ymm2/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}[z], zmm2/m512</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Expand packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Expand (load) up to 16/8/4, contiguous, single precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

**VEXPANDPS (EVEX Encoded Versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

k := 0
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+31:i] := SRC[k+31:k];
      k := k + 32
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPS __m512 __m512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __m512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __m512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __m512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __m256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __m256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __m256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __m256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __m128_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 __m128_maskz_expandloadu_ps( __mmask8 k, void * a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvv != 1111B.
VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 19 / r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 128 bits of packed floating-point values from ymm2 and store results inxmm1/m128.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 19 / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of packed single precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 19 / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of packed single precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 19 / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of packed double precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 19 / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of packed double precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1B / r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of packed single precision floating-point values from zmm2 and store results in xmm1/m256 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1B / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 256 bits of packed double precision floating-point values from zmm2 and store results in xmm1/m256 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:rr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:rr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:rr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:rr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.
If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation**

**VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is a Register**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;

FOR j := 0 TO 3
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:128] := 0

**VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is Memory**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;

FOR j := 0 TO 3
  i := j * 32
  IF k1[j] OR *no writemask*
THEN DEST[i+31:i] := TMP_DEST[i+31:i]
ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
FI;
ENDIF

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is a Register
VL = 256, 512
IF VL = 256
CASE (imm8[0]) OF
0: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;
IF VL = 512
CASE (imm8[1:0]) OF
00: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;
FOR j := 0 TO 1
i := j * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:i] := TMP_DEST[i+63:i]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+63:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:128] := 0

VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is Memory
VL = 256, 512
IF VL = 256
CASE (imm8[0]) OF
0: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;
IF VL = 512
CASE (imm8[1:0]) OF
00: TMP_DEST[127:0] := SRC1[127:0]
ESAC.
FI;
FOR j := 0 TO 1
i := j * 64
IF k1[j] OR *no writemask*
   THEN DEST[i+63:i] := TMP_DEST[i+63:i]
   ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
   FI;
ENDDO

VEXTRACTF32x8 (EVEX.U1.512 Encoded Version) When Destination is a Register
VL = 512
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] := SRC1[255:0]
   1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 7
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TMP_DEST[i+31:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
               DEST[i+31:i] := 0
         FI
      FI
   FI;
ENDDO
DEST[MAXVL-1:256] := 0

VEXTRACTF32x8 (EVEX.U1.512 Encoded Version) When Destination is Memory
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] := SRC1[255:0]
   1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 7
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TMP_DEST[i+31:i]
      FI;
ENDDO

VEXTRACTF64x4 (EVEX.512 Encoded Version) When Destination is a Register
VL = 512
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] := SRC1[255:0]
   1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 3
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := TMP_DEST[i+63:i]
      ELSE

IF *merging-masking*; merging-masking
    THEN *DEST*[i+63:i] remains unchanged*
ELSE *zeroing-masking*; zeroing-masking
    DEST*[i+63:i] := 0
FI
ENDFOR
DEST[MAXVL-1:256] := 0

VEXTRACTF64x4 (EVEX.512 Encoded Version) When Destination is Memory
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC1[255:0]
    1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.
FOR j := 0 TO 3
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST*[i+63:i] := TMP_DEST*[i+63:i]
        ELSE ; merging-masking
            *DEST*[i+63:i] remains unchanged*
        FI;
    ENDFOR

VEXTRACTF128 (Memory Destination Form)
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
ESAC.

VEXTRACTF128 (Register Destination Form)
CASE (imm8[0]) OF
    0: DEST[127:0] := SRC1[127:0]
ESAC.
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 __m256_extractf32x4_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 __m256_mask_extractf32x4_ps(__m256 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m256 __m512_extractf32x4_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m256 __m512_mask_extractf32x4_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m256 __m512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d __m256_extractf64x2_pd(__m256 s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d __m256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d __m256_maskz_extractf64x2_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d __m512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF128 __m128 __m128d __m256_extractf128_ps(__m256 a, int offset);
VEXTRACTF128 __m128d __m256d_extractf128_pd(__m256d a, int offset);
VEXTRACTF128 __m128i __m256_extractf128_sx256(__m256i a, int offset);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

VEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-54, "Type E6NF Class Exception Conditions."
Additionally:

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 111B.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 39 / r ib VEXTRACTI128 xmm1/m128, ymm2, imm8</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 39 / r ib VEXTRACTI32X4 xmm1/m128 <a href="z">k1</a>, ymm2, imm8</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 39 / r ib VEXTRACTI32X4 xmm1/m128 <a href="z">k1</a>, zmm2, imm8</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 39 / r ib VEXTRACTI64X2 xmm1/m128 <a href="z">k1</a>, ymm2, imm8</td>
<td>B V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 39 / r ib VEXTRACTI64X2 xmm1/m128 <a href="z">k1</a>, zmm2, imm8</td>
<td>B V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3B / r ib VEXTRACTI32X8 ymm1/m256 <a href="z">k1</a>, zmm2, imm8</td>
<td>D V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3B / r ib VEXTRACTI64x4 ymm1/m256 <a href="z">k1</a>, zmm2, imm8</td>
<td>C V/V</td>
<td>AVX512F</td>
<td>Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.
VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.
If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation**

**VEXTRACTI32x4 (EVEX encoded versions) when destination is a register**
VL = 256, 512
IF VL = 256
  CASE (imm8[0]) OF
    0: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
FI;
FOR j := 0 TO 3
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*
      DEST[i+31:i] := 0
  FI
  FI;
ENDFOR
DEST[MAXVL-1:128] := 0

**VEXTRACTI32x4 (EVEX encoded versions) when destination is memory**
VL = 256, 512
IF VL = 256
  CASE (imm8[0]) OF
    0: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
for j := 0 to 3
  i := j * 32
  if k1[j] or *no writemask*
    then dest[i+31:i] := tmp_dest[i+31:i]
    else *dest[i+31:i] remains unchanged* ; merging-masking
  fi;
endfor

vextracti64x2 (evex encoded versions) when destination is a register
vl = 256, 512
if vl = 256
  case (imm8[0]) of
    0: tmp_dest[127:0] := src1[127:0]
  esac.
fi;
if vl = 512
  case (imm8[1:0]) of
    00: tmp_dest[127:0] := src1[127:0]
    01: tmp_dest[127:0] := src1[255:128]
  esac.
fi;
for j := 0 to 1
  i := j * 64
  if k1[j] or *no writemask*
    then dest[i+63:i] := tmp_dest[i+63:i]
    else
      if *merging-masking* ; merging-masking
        then *dest[i+63:i] remains unchanged*
      else *zeroing-masking* ; zeroing-masking
        dest[i+63:i] := 0
      fi
    fi;
endfor
dest[128:64] := 0

vextracti64x2 (evex encoded versions) when destination is memory
vl = 256, 512
if vl = 256
  case (imm8[0]) of
    0: tmp_dest[127:0] := src1[127:0]
  esac.
fi;
if vl = 512
  case (imm8[1:0]) of
    00: tmp_dest[127:0] := src1[127:0]
    01: tmp_dest[127:0] := src1[255:128]
  esac.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values


ESAC.

Fi;

FOR j := 0 TO 1
i := j * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:i] := TMP_DEST[i+63:i]
ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
FI;
ENDFOR

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
0: TMP_DEST[255:0] := SRC1[255:0]
1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
i := j * 32
IF k1[j] OR *no writemask*
THEN DEST[i+31:i] := TMP_DEST[i+31:i]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged* 
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+31:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:256] := 0

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory
CASE (imm8[0]) OF
0: TMP_DEST[255:0] := SRC1[255:0]
1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 7
i := j * 32
IF k1[j] OR *no writemask*
THEN DEST[i+31:i] := TMP_DEST[i+31:i]
ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
FI;
ENDFOR

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
0: TMP_DEST[255:0] := SRC1[255:0]
1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract Packed Integer Values
Vol. 2C 5-167
FOR j := 0 TO 3
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST*[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:256] := 0

**VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory**

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] := SRC1[255:0]
  1: TMP_DEST[255:0] := SRC1[511:256]
ESAC.

FOR j := 0 TO 3
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
  ELSE *DEST*[i+63:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

**VEXTRACTI128 (memory destination form)**

CASE (imm8[0]) OF
  0: DEST[127:0] := SRC1[127:0]
ESAC.

**VEXTRACTI128 (register destination form)**

CASE (imm8[0]) OF
  0: DEST[127:0] := SRC1[127:0]
ESAC.
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_extracti64x2_epi64(__m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_maskz_extracti64x2_epi64(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-54, “Type E6NF Class Exception Conditions.”
Additionally:
#UD IF VEX.L = 0.
#UD IF VEX.vvv != 1111B or EVEX.vvv != 1111B.
**VFCMADDQDPH/VFMADDQDPH—Complex Multiply and Accumulate FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.MAP6.W0 56 /r VFCMADDQDPH xmm1[k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.MAP6.W0 56 /r VFCMADDQDPH ymm1[k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from ymm2 and complex conjugate of ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.MAP6.W0 56 /r VFCMADDQDPH zmm1[k1][z], zmm2, zmm3/m512/m32bcst {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from zmm2 and complex conjugate of zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.MAP6.W0 56 /r VFMADDQDPH xmm1[k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.MAP6.W0 56 /r VFMADDQDPH ymm1[k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.MAP6.W0 56 /r VFMADDQDPH zmm1[k1][z], zmm2, zmm3/m512/m32bcst {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.
Operation
VFCMADDQPH dest{k1}, src1, src2 (AVX512)
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF broadcasting and src2 is memory:
      tsrc2.fp16[2*i+0] := src2.fp16[0]
      tsrc2.fp16[2*i+1] := src2.fp16[1]
    ELSE:
      tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
      tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    // conjugate version subtracts odd final term
    dest.fp16[2*i+0] := tmp[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]

ELSE IF *zeroing*:
  dest.fp16[2*i+0] := 0
  dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0

VFMADDQPH dest{k1}, src1, src2 (AVX512)
VL = 128, 256, 512
KL := VL / 32

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF broadcasting and src2 is memory:
      tsrc2.fp16[2*i+0] := src2.fp16[0]
      tsrc2.fp16[2*i+1] := src2.fp16[1]
    ELSE:
      tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
      tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    tmp[2*i+0] := dest.fp16[2*i+0] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    // non-conjugate version subtracts even term

ELSE IF *zeroing*:
dest.fp16[2*i+0] := 0
dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFCMADDCPH __m128h _mm_fcmadd_pch (__m128h a, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCPH __m128h _mm_mask3_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c, __mmask8 k);
VFCMADDCPH __m128h _mm_maskz_fmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFCMADDCPH __m256h _mm256_fcmadd_pch (__m256h a, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask_fcmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFCMADDCPH __m256h _mm256_mask3_fcmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFCMADDCPH __m256h _mm256_maskz_fcmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFCMADDCPH __m512h _mm512_fcmadd_pch (__m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask_fcmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_mask3_fcmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFCMADDCPH __m512h _mm512_maskz_fcmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h _mm512_fmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask_fmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h _mm512_mask3_fmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFCMADDCPH __m512h _mm512_maskz_fmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFMADDCPH __m128h _mm_fmadd_pch (__m128h a, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCPH __m128h _mm_mask3_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c, __mmask8 k);
VFMADDCPH __m128h _mm_maskz_fmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFMADDCPH __m256h _mm256_fmadd_pch (__m256h a, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask_fmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFMADDCPH __m256h _mm256_mask3_fmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFMADDCPH __m256h _mm256_maskz_fmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFMADDCPH __m512h _mm512_fmadd_pch (__m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask_fmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_mask3_fmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFMADDCPH __m512h _mm512_maskz_fmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h _mm512_fmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask_fmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h _mm512_mask3_fmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFMADDCPH __m512h _mm512_maskz_fmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-49, "Type E4 Class Exception Conditions."
Additionally:

#UD If (dest_reg == src1_reg) or (dest_reg == src2_reg).
VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F2.MAP6.W0 57 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAP6.W0 57 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

Operation

VFCMADDCSH dest[k1], src1, src2 (AVX512)

IF k1[0] or *no writemask*:

\[
\begin{align*}
\text{tmp}[0] & := \text{dest}.\text{fp16}[0] + \text{src1}.\text{fp16}[0] \times \text{src2}.\text{fp16}[0] \\
\text{tmp}[1] & := \text{dest}.\text{fp16}[1] + \text{src1}.\text{fp16}[1] \times \text{src2}.\text{fp16}[0]
\end{align*}
\]

// conjugate version subtracts odd final term
\[
\begin{align*}
\text{dest}.\text{fp16}[0] & := \text{tmp}[0] + \text{src1}.\text{fp16}[1] \times \text{src2}.\text{fp16}[1] \\
\text{dest}.\text{fp16}[1] & := \text{tmp}[1] - \text{src1}.\text{fp16}[0] \times \text{src2}.\text{fp16}[1]
\end{align*}
\]

ELSE IF *zeroing*:

\[
\begin{align*}
\text{dest}.\text{fp16}[0] & := 0 \\
\text{dest}.\text{fp16}[1] & := 0
\end{align*}
\]

\[
\text{DEST}[127:32] = \text{src1}[127:32] // copy upper part of src1 \\
\text{DEST}[\text{MAXVL}-1:128] = 0
\]
VFMADDCCSH dest[k1], src1, src2 (AVX512)
IF k1[0] or *no writemask*:
    \[ \text{tmp}[0] := \text{dest.fp16}[0] + \text{src1.fp16}[0] \times \text{src2.fp16}[0] \]
    \[ \text{tmp}[1] := \text{dest.fp16}[1] + \text{src1.fp16}[1] \times \text{src2.fp16}[0] \]

// non-conjugate version subtracts last even term
    \[ \text{dest.fp16}[0] := \text{tmp}[0] - \text{src1.fp16}[1] \times \text{src2.fp16}[1] \]
    \[ \text{dest.fp16}[1] := \text{tmp}[1] + \text{src1.fp16}[0] \times \text{src2.fp16}[1] \]
ELSE IF *zeroing*:
    \[ \text{dest.fp16}[0] := 0 \]
    \[ \text{dest.fp16}[1] := 0 \]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDCCSH __m128h _mm_fcmadd_round_sch (__m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCCSH __m128h _mm_mask_fcmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFMADDCCSH __m128h _mm_mask3_fcmadd_round_sch (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
VFMADDCCSH __m128h _mm_maskz_fcmadd_round_sch (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
VFMADDCCSH __m128h _mm_fcmadd_sch (__m128h a, __m128h b, __m128h c);
VFMADDCCSH __m128h _mm_mask_fcmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCCSH __m128h _mm_mask3_fcmadd_sch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCCSH __m128h _mm_maskz_fcmadd_sch (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions
Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”
Additionally:
#UD If (dest_reg == src1_reg) or (dest_reg == src2_reg).
### Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

### Operation

**VFCMULCPH dest{k1}, src1, src2 (AVX512)**

VL := 128, 256 or 512

KL := VL/32

FOR i := 0 to KL-1:

IF k1[i] or *no writemask*:

IF broadcasting and src2 is memory:

\[ \text{tSRC2.fp16[2*i+0]} := \text{src2.fp16[0]} \]

\[ \text{tSRC2.fp16[2*i+1]} := \text{src2.fp16[1]} \]

ELSE:

\[ \text{tSRC2.fp16[2*i+0]} := \text{src2.fp16[2*i+0]} \]

\[ \text{tSRC2.fp16[2*i+1]} := \text{src2.fp16[2*i+1]} \]

FOR i := 0 to KL-1:

### Instruction Set Reference, Volume 2C

**VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.MAP6.W0 D6 /r VFCMULCPH xmm1{k1}[z], xmm2, xmm3/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F2.MAP6.W0 D6 /r VFCMULCPH ymm1{k1}[z], ymm2, ymm3/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from ymm2 and complex conjugate of ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F2.MAP6.W0 D6 /r VFCMULCPH zmm1{k1}[z], zmm2, zmm3/m512/m32bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from zmm2 and complex conjugate of zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.MAP6.W0 D6 /r VFMULCPH xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.MAP6.W0 D6 /r VFMULCPH ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.MAP6.W0 D6 /r VFMULCPH zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrm/r (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
IF k1[i] or *no writemask*:
   tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
   tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

FOR i := 0 to KL-1:
   IF k1[i] or *no writemask*:
      // conjugate version subtracts odd final term
   ELSE IF *zeroing*:
      dest.fp16[2*i+0] := 0
      dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0

VFMULCPH dest{k1}, src1, src2 (AVX512)
VL = 128, 256 or 512
KL := VL/32

FOR i := 0 to KL-1:
   IF k1[i] or *no writemask*:
      IF broadcasting and src2 is memory:
         tsrc2.fp16[2*i+0] := src2.fp16[0]
         tsrc2.fp16[2*i+1] := src2.fp16[1]
      ELSE:
         tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
         tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

   FOR i := 0 to KL-1:
      IF k1[i] or *no writemask*:
      tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
      tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

   FOR i := 0 to KL-1:
      IF k1[i] or *no writemask*:
         // non-conjugate version subtracts last even term
      ELSE IF *zeroing*:
      dest.fp16[2*i+0] := 0
      dest.fp16[2*i+1] := 0

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFCMULCPH _m128h _mm_cmul_pch (__m128h a, __m128h b);
VFCMULCPH _m128h _mm_mask_cmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH _m128h _mm_maskz_cmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH _m256h _mm256_cmul_pch (__m256h a, __m256h b);
VFCMULCPH _m256h _mm256_mask_cmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH _m256h _mm256_maskz_cmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH _m512h _mm512_cmul_pch (__m512h a, __m512h b);
VFCMULCPH _m512h _mm512_mask_cmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH _m512h _mm512_maskz_cmul_pch (__mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h _mm512_cmul_round_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_cmul_round_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_cmul_round_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m128h _mm_cmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h _mm_mask_cmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h _mm_maskz_cmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h _mm256_cmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h _mm256_mask_cmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h _mm256_maskz_cmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h _mm512_cmul_pch (__m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_mask_cmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b, const int rounding);
VFCMULCPH __m512h _mm512_maskz_cmul_pch (__mmask16 k, __m512h a, __m512h b, const int rounding);

VFMULCPH __m128h _mm_fmul_pch (__m128h a, __m128h b);
VFMULCPH __m128h _mm_mask_fmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCPH __m128h _mm_maskz_fmul_pch (__mmask8 k, __m128h a, __m128h b);
VFMULCPH __m256h _mm256_fmul_pch (__m256h a, __m256h b);
VFMULCPH __m256h _mm256_mask_fmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFMULCPH __m256h _mm256_maskz_fmul_pch (__mmask8 k, __m256h a, __m256h b);
VFMULCPH __m512h _mm512_fmul_pch (__m512h a, __m512h b);
VFMULCPH __m512h _mm512_mask_fmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFMULCPH __m512h _mm512_maskz_fmul_pch (__mmask16 k, __m512h a, __m512h b);

SIMD Floating-Point Exceptions
Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”
Additionally:
#UD If (dest_reg == src1_reg) or (dest_reg == src2_reg).

VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values
Vol. 2C 5-177
**VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F2.MAP6.W0 D7 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAP6.W0 D7 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

**Operation**

**VFCMULCSH dest[k1], src1, src2 (AVX512)**

\[
KL := VL / 32
\]

IF k1[0] or *no writemask*:

\[
\begin{align*}
\text{tmp.fp16[0]} & := \text{src1.fp16[0]} * \text{src2.fp16[0]} \\
\text{tmp.fp16[1]} & := \text{src1.fp16[1]} * \text{src2.fp16[1]}
\end{align*}
\]

// conjugate version subtracts odd final term

\[
\begin{align*}
\text{dest.fp16[0]} & := \text{tmp.fp16[0]} + \text{src1.fp16[1]} * \text{src2.fp16[1]} \\
\text{dest.fp16[1]} & := \text{tmp.fp16[1]} - \text{src1.fp16[0]} * \text{src2.fp16[1]}
\end{align*}
\]

ELSE IF *zeroing*:

\[
\begin{align*}
\text{dest.fp16[0]} & := 0 \\
\text{dest.fp16[1]} & := 0
\end{align*}
\]

\[
\text{DEST[127:32]} := \text{src1[127:32]} // copy upper part of src1
\]

\[
\text{DEST[MAXVL-1:128]} := 0
\]
**VFMULCSH dest[k1], src1, src2 (AVX512)**

\( KL := VL / 32 \)

If \( k1[0] \) or *no writemask*:

- non-conjugate version subtracts last even term
  
  \[
  \text{tmp.fp16}[0] := \text{src1.fp16}[0] \times \text{src2.fp16}[0] \\
  \text{tmp.fp16}[1] := \text{src1.fp16}[1] \times \text{src2.fp16}[0] \\
  \text{dest.fp16}[0] := \text{tmp.fp16}[0] - \text{src1.fp16}[1] \times \text{src2.fp16}[1] \\
  \text{dest.fp16}[1] := \text{tmp.fp16}[1] + \text{src1.fp16}[0] \times \text{src2.fp16}[1]
  \]

ELSE IF *zeroing*:

\( \text{dest.fp16}[0] := 0 \)
\( \text{dest.fp16}[1] := 0 \)

\( \text{DEST}[127:32] := \text{src1}[127:32] \) // copy upper part of src1
\( \text{DEST}[MAXVL-1:128] := 0 \)

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_cmul_round_sch }(_m128h \text{ a}, _m128h \text{ b}, \text{const int rounding});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_mask_cmul_round_sch }(_m128h \text{ src, _mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b}, \text{const int rounding});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_maskz_cmul_round_sch }(_\text{mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_cmul_sch }(_m128h \text{ a}, _m128h \text{ b});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_mask_cmul_sch }(_m128h \text{ src, _mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_maskz_cmul_sch }(_\text{mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_fcmul_round_sch }(_m128h \text{ a}, _m128h \text{ b}, \text{const int rounding});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_mask_fcmul_round_sch }(_m128h \text{ src, _mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b}, \text{const int rounding});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_maskz_fcmul_round_sch }(_\text{mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_fmul_round_sch }(_m128h \text{ a}, _m128h \text{ b}, \text{const int rounding});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_mask_fmul_round_sch }(_m128h \text{ src, _mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b}, \text{const int rounding});
\]

\[
\text{VFCMULCSH } \_\_m128h \text{ _mm_maskz_fmul_round_sch }(_\text{mmask8 } \text{ k}, _m128h \text{ a}, _m128h \text{ b});
\]

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”

Additionally:

\#UD \( \text{If (dest\_reg == src1\_reg) or (dest\_reg == src2\_reg).} \)
VFIXUPIMMPD—Fix Up Special Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 54 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 54 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 54 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
</tr>
</tbody>
</table>

**Description**

Perform fix-up of quad-word elements encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.
Operation

```c
enum TOKEN_TYPE {
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}
```

```c
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] := ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    } ; end source special CASE(tsrc...)

token_response[3:0] = tbl3[3+4*j:4*j];
```

```c
CASE(token_response[3:0]) {
    0000: dest[63:0] := dest[63:0];  ; preserve content of DEST
    0001: dest[63:0] := tsrc[63:0];   ; pass through src1 normal input value, denormal as zero
    0010: dest[63:0] := QNaN(tsrc[63:0]);
    0011: dest[63:0] := QNAN_Indefinite;
    0100: dest[63:0] := -INF;
    0101: dest[63:0] := +INF;
    0111: dest[63:0] := -0;
    1000: dest[63:0] := +0;
    1001: dest[63:0] := -1;
    1010: dest[63:0] := +1;
    1011: dest[63:0] := ½;
    1100: dest[63:0] := 90.0;
    1101: dest[63:0] := PI/2;
    1110: dest[63:0] := MAX_FLOAT;
    1111: dest[63:0] := -MAX_FLOAT;
} ; end of token_response CASE
```

```c
; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[63:0];
} ; end of FIXUPIMM_DP()

VFIXUPIMMPD
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
                ELSE
                    DEST[i+63:i] := FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE DEST[i+63:i] := 0 ; zeroing-masking
                FI
            FI
    ENDFOR
DEST[MAXVL-1:VL] := 0

Immediate Control Description:

Figure 5-9. VFIXUPIMMPD Immediate Control Description
**Intel C/C++ Compiler Intrinsic Equivalent**

VFIXUPIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd(__m256d a, m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d a, __mmask8 k, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256i c, int imm8);
VFIXUPIMMPD __m128d _mm128_fixupimm_pd(__m128d a, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm128_mask_fixupimm_pd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm128_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128d b, 128ic, int imm8);

**SIMD Floating-Point Exceptions**

Zero, Invalid.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”
VFIXUPIMMPS—Fix Up Special Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 54 /r</td>
<td>A V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 54 /r</td>
<td>A V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 54 /r ib</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Perform fix-up of doubleword elements encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.
Operation

enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0], tbl3[31:0], imm8[7:0]){
    tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    } ; end source special CASE(tsrc...)
    token_response[3:0] = tbl3[3+4*j:4*j];
    CASE(token_response[3:0]) {
        0000: dest[31:0] := dest[31:0];  ; preserve content of DEST
        0001: dest[31:0] := tsrc[31:0];   ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] := QNaN(tsrc[31:0]);
        0011: dest[31:0] := QNAN_Indefinite;
        0100: dest[31:0] := -INF;
        0101: dest[31:0] := +INF;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011: dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    } ; end of token_response CASE
    ; The required fault reporting from imm8 is extracted
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

VFIXUPIMMPS (EVEX)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] := FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8[7:0])
        ELSE
          DEST[i+31:i] := FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8[7:0])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE DEST[i+31:i] := 0 ; zeroing-masking
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

Immediate Control Description:

![Figure 5-10. VFIXUPIMMPS Immediate Control Description](image-url)
**Intel C/C++ Compiler Intrinsic Equivalent**

VFIXUPIMMPS __m512 __mm512_fixupimm_ps( __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_fixupimm_round_ps( __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_round_ps( __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask8 k, __m512 b, __m128i c, int imm8);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_round_ps( __mmask8 k, __m512 b, __m128i c, int imm8);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask8 k, __m512 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 __mm_mask_fixupimm_ps(__m128 k, __m128 a, __m128i c, int imm8);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i c, int imm8);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_round_ps( __mmask8 k, __m128 a, __m128i c, int imm8);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i c, int imm8);

**SIMD Floating-Point Exceptions**

Zero, Invalid.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”
VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

| Opcode/ | Op/  | 64/32 | CPUID | Description |
| Instruction | En | Bit Mode | Feature | |
| EVEX.LLIG.66.0F3A.W1 55 / r ib | V/V | VV | AVX512F | Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1. |

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Perform a fix-up of the low quadword element encoded in double precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64-bit memory location.

The two-level look-up table perform a fix-up of each double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source operand in order to match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

class TOKEN_TYPE {
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}

5-188 Vol. 2C
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8[7:0])

\[
\text{tsrc}[63:0] := \begin{cases}
    0 & \text{if } (\text{src1}[62:52] = 0) \text{ AND } (\text{MXCSR.DAZ} = 1) \\
    \text{src1}[63:0] & \text{otherwise}
\end{cases}
\]

\[
\text{CASE}(\text{tsrc}[63:0]) \text{ of TOKEN_TYPE} \{
\begin{align*}
    \text{QNAN_TOKEN} & : j := 0; \\
    \text{SNAN_TOKEN} & : j := 1; \\
    \text{ZERO_VALUE_TOKEN} & : j := 2; \\
    \text{POS_ONE_VALUE_TOKEN} & : j := 3; \\
    \text{NEG_INF_TOKEN} & : j := 4; \\
    \text{POS_INF_TOKEN} & : j := 5; \\
    \text{NEG_VALUE_TOKEN} & : j := 6; \\
    \text{POS_VALUE_TOKEN} & : j := 7;
\end{align*}
\}
\]

\text{; The required response from src3 table is extracted}

\[
\text{token_response}[3:0] = tbl3[3+4*j:4*j];
\]

\[
\text{CASE}(\text{token_response}[3:0]) \{
\begin{align*}
    0000 & : \text{dest}[63:0] := \text{dest}[63:0]; \\
    0001 & : \text{dest}[63:0] := \text{tsrc}[63:0]; \\
    0010 & : \text{dest}[63:0] := \text{QNaN}(\text{tsrc}[63:0]); \\
    0011 & : \text{dest}[63:0] := \text{QNAN_Indefinite}; \\
    0100 & : \text{dest}[63:0] := -\infty; \\
    0101 & : \text{dest}[63:0] := +\infty; \\
    0110 & : \text{dest}[63:0] := \text{tsrc}.sign? -\infty : +\infty; \\
    0111 & : \text{dest}[63:0] := -0; \\
    1000 & : \text{dest}[63:0] := +0; \\
    1001 & : \text{dest}[63:0] := -1; \\
    1010 & : \text{dest}[63:0] := +1; \\
    1011 & : \text{dest}[63:0] := \frac{1}{2}; \\
    1100 & : \text{dest}[63:0] := 90.0; \\
    1101 & : \text{dest}[63:0] := \pi/2; \\
    1110 & : \text{dest}[63:0] := \text{MAX_FLOAT}; \\
    1111 & : \text{dest}[63:0] := -\text{MAX_FLOAT};
\end{align*}
\}
\]

\text{; The required fault reporting from imm8 is extracted}

\text{TOKENs are mutually exclusive and TOKENs priority defines the order.}

\text{Multiple faults related to a single token can occur simultaneously.}

\text{IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;}
\text{IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;}
\]
\text{; end fault reporting}
\text{return dest}[63:0];
\}
\text{; end of FIXUPIMM_DP()}

**VFIXUPIMMSD (EVEX encoded version)**

IF k1[0] OR *no writemask*
    THEN DEST[63:0] := FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE DEST[63:0] := 0 ; zeroing-masking
    FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Immediate Control Description:

![Figure 5-11. VFIXUPIMMSD Immediate Control Description](image)

**Intel C/C++ Compiler Intrinsic Equivalent**

VFIXUPIMMSD __m128d __m128d _mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __m128d _mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d __m128d _mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d __m128d _mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Table 2-47, “Type E3 Class Exception Conditions.”
VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W0 55 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Perform a fix-up of the low doubleword element encoded in single precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each single precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e., zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```c
enum TOKEN_TYPE
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}
```
FIXUPIMM_SP (dest[31:0], src1[31:0], tbl3[31:0], imm8[7:0]){
    tsrc[31:0] := ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }   ; end source special CASE(tsrc...)
    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];
    CASE(token_response[3:0]) {
        0000: dest[31:0] := dest[31:0];   ; preserve content of DEST
        0001: dest[31:0] := tsrc[31:0];   ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] := QNaN(tsrc[31:0]);
        0011: dest[31:0] := QNaN_Indefinite;
        0100: dest[31:0] := -INF;
        0101: dest[31:0] := +INF;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011: dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    } ; end of token_response CASE
    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
    ; end fault reporting
    return dest[31:0];
} ; end of FIXUPIMM_SP()
VFIXUPIMMSS (EVEX encoded version)

IF k1[0] OR *no writemask*
    THEN DEST[31:0] := FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE DEST[31:0] := 0 ; zeroing-masking
    FI;
DEST[MAXVL-1:128] := 0

Immediate Control Description:

![Figure 5-12. VFIXUPIMMSS Immediate Control Description]

**Intel C/C++ Compiler Intrinsic Equivalent**

VFIXUPIMMSS _m128 _mm_fixupimm_ss( _m128 a, _m128i tbl, int imm);
VFIXUPIMMSS _m128 _mm_mask_fixupimm_ss( _m128 s, _mmask8 k, _m128 a, _m128i tbl, int imm);
VFIXUPIMMSS _m128 _mm_maskz_fixupimm_ss( _mmask8 k, _m128 a, _m128i tbl, int imm);
VFIXUPIMMSS _m128 _mm_fixupimm_round_ss( _m128 a, _m128i tbl, int imm, int sae);
VFIXUPIMMSS _m128 _mm_mask_fixupimm_round_ss( _m128 s, _mmask8 k, _m128 a, _m128i tbl, int imm, int sae);
VFIXUPIMMSS _m128 _mm_maskz_fixupimm_round_ss( _mmask8 k, _m128 a, _m128i tbl, int imm, int sae);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions."
## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst(ier)</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst(ier)</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst(ier)</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
**Description**

Performs a set of SIMD multiply-add computation on packed double precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PD:** Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

**VFMADD213PD:** Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

**VFMADD231PD:** Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---
INSTRUCTION SET REFERENCE, V

Operation
In the operations below, “*" and “+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI
VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                        DEST[i+63:i] :=
                            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] :=
      RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
     ELSE ; zeroing-masking
       DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
      ELSE
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
     ELSE ; zeroing-masking
       DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
                FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADDxxxPD _m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD _m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD _m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD _m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD _m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD _m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD _m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD _m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD _m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);
VFMADDxxxPD _m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD _m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD _m256d _mm256_maskz_fmadd_round_pd(__mmask8 k, __m256d a, __m256d b, __m256d c, int r);
VFMADDxxxPD _m256d _mm256_mask3_fmadd_round_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD _m128d _mm_mask_fmadd_pd(__m128d a, __m128d b, __m128d c);
VFMADDxxxPD _m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD _m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD _m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD _m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
## VF[N]MADD[132,213,231]PH—Fused Multiply-Add of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP6.w0 98 /r VFMADD132PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add to xmm2, and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.w0 98 /r VFMADD132PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add to ymm2, and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.w0 98 /r VFMADD132PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add to zmm2, and store the result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.w0 A8 /r VFMADD213PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, add to xmm3/m128/m16bcst, and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.w0 A8 /r VFMADD213PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, add to ymm3/m256/m16bcst, and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.w0 A8 /r VFMADD213PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, add to zmm3/m512/m16bcst, and store the result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.w0 B8 /r VFMADD231PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, add to xmm3/m128/m16bcst, and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.w0 B8 /r VFMADD231PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, add to ymm3/m256/m16bcst, and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.w0 B8 /r VFMADD231PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, add to zmm3/m512/m16bcst, and store the result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.w0 9C /r VFNMADD132PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm2, and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.w0 9C /r VFNMADD132PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm2, and store the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.w0 9C /r VFNMADD132PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm2, and store the result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.w0 AC /r VFNMADD213PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m128/m16bcst, and store the result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.w0 AC /r VFNMADD213PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, and negate the value. Add this value to ymm3/m256/m16bcst, and store the result in ymm1.</td>
<td></td>
</tr>
</tbody>
</table>
This instruction performs a packed multiply-add or negated multiply-add computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation "132", "213" and "231" indicate the use of the operands in $\pm A \times B + C$, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-2. The destination elements are updated according to the writemask.

Table 5-2. VF[,N]MADD[132,213,231]PH Notation for Operands

<table>
<thead>
<tr>
<th>Notation</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>dest = $\pm$ dest*src3+src2</td>
</tr>
<tr>
<td>231</td>
<td>dest = $\pm$ src2*src3+dest</td>
</tr>
<tr>
<td>213</td>
<td>dest = $\pm$ src2*dest+src3</td>
</tr>
</tbody>
</table>
**Operation**

*VF[N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register*

VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

*VF[N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source*

VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 + SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
VF[N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
        ELSE IF *zeroing*:
            DEST.fp16[j] := 0
        // else dest.fp16[j] remains unchanged
    DEST[MAXVL-1:VL] := 0

VF[N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE IF *zeroing*:
            DEST.fp16[j] := 0
        // else dest.fp16[j] remains unchanged
    DEST[MAXVL-1:VL] := 0
VF[N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *negative form:
         DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] + DEST.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

VF[N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 + DEST.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132PH, VFMADD213PH, and VFMADD231PH:

__m128h _mm_fmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFNMADD132PH, VFNMADD213PH, and VFNMADD231PH:

__m128h _mm_fnmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fnmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fnmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fnmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fnmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fnmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fnmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fnmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fnmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fnmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fnmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VFMADDRND231PD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values with rounding control

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F3A.W1 B8 /r /ib V/V FMA</td>
<td></td>
<td></td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0.</td>
</tr>
<tr>
<td>VFMADDRND231PD xmm0, xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F3A.W1 B8 /r /ib V/V FMA</td>
<td></td>
<td></td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0.</td>
</tr>
<tr>
<td>VFMADDRND231PD ymm0, ymm1, ymm2/m256, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

Multiplies the two or four packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two or four packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

The immediate byte defines several bit fields that control rounding, DAZ, FTZ, and exception suppression (SeeTable 5-3). The rounding mode specified in MXCSR.RC may be bypassed if the immediate bit called MS1 (MXCSR.RC Override) is set. Likewise, the MXCSR.FTZ and MXCSR.DAZ may also be bypassed if the immediate bit called MS2 (MXCSR.FTZ/DAZ Override) is set. In case SAE (Suppress All Exceptions) bit is set (i.e. imm8[3] = 1), the status flags in MXCSR are not updated and no SIMD floating-point exceptions are raised. When SAE bit is not set (i.e. imm8[3] = 0) then SIMD floating-point exceptions are signaled according to the MXCSR. If any result operand is an SNaN then it will be converted to a QNaN.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Compiler tools may optionally support the complementary mnemonic VMADDRND321PD. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column. See also Section 2.3.1, "FMA Instruction Operand Order and Arithmetic Behavior".

**Operation**

In the operations below, “+” and “*” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

### VFMADDRND231PD DEST, SRC2, SRC3, imm8

IF (VEX.128) THEN
  MAXVL = 2
ELSEIF (VEX.256)
  MAXVL = 4
FI
IF (imm8[3] = 1) THEN
  Suppress_SIMD_Exception_Signaling();
FI
For i = 0 to MAXVL-1 {
  n = 64*i;
}
IF (VEX.128) THEN
  DEST[255:128] ← 0
FI
IF (imm8[3] = 1) THEN
    Resume_SIMD_Exception_Signaling_Reporting();
FI

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDRND231PD __m128d __mm_fmaddround_pd (__m128d a, __m128d b, __m128d c, const int ctrl);
VFMADDRND231PD __m256d __mm256_fmaddround_pd (__m256d a, __m256d b, __m256d c, const int ctrl);

SIMD Floating-Point Exceptions
IF imm[3] = 1 Then
    None
Else
    Overflow, Underflow, Invalid, Precision, Denormal
FI

Other Exceptions
See Exceptions Type 2
### VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 98 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 A8 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 B8 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 98 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 A8 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 B8 /r</td>
<td>A/V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 98 /r</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 A8 /r</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 B8 /r</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 98 /r</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 A8 /r</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 B8 /r</td>
<td>B/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.512.66.0F38.W0 98 /r</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.512.66.0F38.W0 A8 /r</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.512.66.0F38.W0 B8 /r</td>
<td>B/V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values

**Description**

Performs a set of SIMD multiply-add computation on packed single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:**

The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:**

The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version:**

The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 4
ELSEIF (VEX.256) THEN
   MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
   n := 32*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMADD213PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMADD231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 4
ELSEIF (VEX.256)
    MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        ELSE
            IF *merging-masking*; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values

INSTRUCTION SET REFERENCE, V

Vol. 2C 5-213

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
          ELSE
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        FI;
      ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0\) TO \(KL-1\)
\(i := j \times 32\)
IF \(k1[j]\) OR *no writemask*
THEN
IF (EVEX.b = 1)
THEN
\(\text{DEST}[i+31:i] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[i+31:i] \times \text{DEST}[i+31:i] + \text{SRC3}[i+31:i])\)
ELSE
\(\text{DEST}[i+31:i] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[i+31:i] \times \text{DEST}[i+31:i] + \text{SRC3}[i+31:i])\)
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
\(\text{DEST}[i+31:i] := 0\)
FI
FI;
ENDFOR
\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)
 IF (VL = 512) AND (EVEX.b = 1)
 THEN
 SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
 ELSE
 SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
 FI;
 FOR \(j := 0\) TO \(KL-1\)
 \(i := j \times 32\)
 IF \(k1[j]\) OR *no writemask*
 THEN \(\text{DEST}[i+31:i] := \text{RoundFPControl}(\text{SRC2}[i+31:i] \times \text{SRC3}[i+31:i] + \text{DEST}[i+31:i])\)
 ELSE
 IF *merging-masking* ; merging-masking
 THEN *DEST[i+31:i] remains unchanged*
 ELSE ; zeroing-masking
 \(\text{DEST}[i+31:i] := 0\)
 FI
 FI;
ENDFOR
\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0\) TO \(KL-1\)
\(i := j \times 32\)
IF \(k1[j]\) OR *no writemask*
THEN
  IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] :=
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
    ELSE
      DEST[i+31:i] :=
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged* 
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDxxxPS __m512 _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m256 _mm256_fmadd_ps(__m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps(__mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm128_mask_fmadd_ps(__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm128_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm128_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps (__mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps (__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
**VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 [k1][z], xmm2, xmm3/m64{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 [k1][z], xmm2, xmm3/m64{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 [k1][z], xmm2, xmm3/m64{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD multiply-add computation on the low double precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

VFMADD132SD: Multiplies the low double precision floating-point value from the first source operand to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point values in the second source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low double precision floating-point value from the second source operand to the low double precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the third source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low double precision floating-point value from the second source to the low double precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double precision floating-point value in the first source operand, performs rounding and stores the resulting double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.
**Operation**

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

IF k1[0] or *no writemask*
    THEN
        DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

**VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

IF k1[0] or *no writemask*
    THEN
        DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
INSTRUCTION SET REFERENCE, V

VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN  DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] := 0
            FI;
    FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := MAXVL-1:128RoundFPControl_MXCSR(Dest[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[MAXVL-1:128] := 0

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[MAXVL-1:128] := 0

VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VF[N]MADD[132,213,231]SH—Fused Multiply-Add of Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 99 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm3/m16, add to xmm2, and store the result in xmm1.</td>
</tr>
<tr>
<td>VFMADD132SH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 A9 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm2, add to xmm3/m16, and store the result in xmm1.</td>
</tr>
<tr>
<td>VFMADD213SH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 B9 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm2 and xmm3/m16, add to xmm1, and store the result in xmm1.</td>
</tr>
<tr>
<td>VFMADD231SH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 9D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Add this value to xmm2, and store the result in xmm1.</td>
</tr>
<tr>
<td>VFNMADD132SH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 AD /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm2, and negate the value. Add this value to xmm3/m16, and store the result in xmm1.</td>
</tr>
<tr>
<td>VFNMADD213SH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 BD /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Add this value to xmm1, and store the result in xmm1.</td>
</tr>
<tr>
<td>VFNMADD231SH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a scalar multiply-add or negated multiply-add computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation “132”, “213” and “231” indicate the use of the operands in ±A * B + C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-4.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 5-4. VF[N]MADD[132,213,231]SH Notation for Operands

<table>
<thead>
<tr>
<th>Notation</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>dest = ± dest*src3+src2</td>
</tr>
<tr>
<td>231</td>
<td>dest = ± src2*src3+dest</td>
</tr>
<tr>
<td>213</td>
<td>dest = ± src2*dest+src3</td>
</tr>
</tbody>
</table>
**Operation**

**VF\[N\]MADD132SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
   IF *negative form*:
      DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
   ELSE:
      DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] + SRC2.fp16[0])
ELSE IF *zeroing*:
   DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged
DEST[127:16] := 0
DEST[MAXVL-1:128] := 0

**VF\[N\]MADD213SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
   IF *negative form*:
      DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
   ELSE:
      DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] + SRC3.fp16[0])
ELSE IF *zeroing*:
   DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged
DEST[127:16] := 0
DEST[MAXVL-1:128] := 0

**VF\[N\]MADD231SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:
   IF *negative form*:
      DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
   ELSE:
      DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] + DEST.fp16[0])
ELSE IF *zeroing*:
   DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged
DEST[127:16] := 0
DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADD132SH, VFMADD213SH, and VFMADD231SH:

__m128h_mm_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h_mm_mask_fmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h_mm_mask3_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h_mm_maskz_fmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h_mm_fmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h_mm_mask_fmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h_mm_mask3_fmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h_mm_maskz_fmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

VFNMADD132SH, VFNMADD213SH, and VFNMADD231SH:

__m128h_mm_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h_mm_mask_fnmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h_mm_mask3_fnmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, __m128h a, __m128h b, __m128h c);
__m128h_mm_maskz_fnmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h_mm_fnmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h_mm_mask_fnmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h_mm_mask3_fnmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h_mm_maskz_fnmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
**VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W0 99 /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMADD132SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMADD213SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMADD231SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD multiply-add computation on single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

**VFMADD132SS**:

Multiplies the low single precision floating-point value from the first source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the second source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

**VFMADD213SS**:

Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the third source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

**VFMADD231SS**:

Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the first source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version**:

The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation
In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or *no writemask*

THEN

    DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])

ELSE

    IF *merging-masking* ; merging-masking

    THEN *DEST[31:0] remains unchanged*

    ELSE ; zeroing-masking

    THEN DEST[31:0] := 0

FI;

FI;


DEST[MAXVL-1:128] := 0

**VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or *no writemask*

THEN

    DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])

ELSE

    IF *merging-masking* ; merging-masking

    THEN *DEST[31:0] remains unchanged*

    ELSE ; zeroing-masking

    THEN DEST[31:0] := 0

FI;

FI;


DEST[MAXVL-1:128] := 0
VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

Fi;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] := 0
            FI;
        FI;
    FI;
DEST[MAXVL-1:128] := 0

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[MAXVL-1:128] := 0

VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[MAXVL-1:128] := 0

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
### VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V FMA</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V FMA</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A6 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 B6 /r VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 (k1)[x], zmm2, zmm3/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 (k1)[x], zmm2, zmm3/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 (k1)[x], zmm2, zmm3/m512/m64bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

VFMADDSUB132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.
VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3
IF (VEX.128) THEN
DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
FI

VFMADDSUB213PD DEST, SRC2, SRC3
IF (VEX.128) THEN
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
FI

VFMADDSUB231PD DEST, SRC2, SRC3
IF (VEX.128) THEN
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
FI

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
   THEN
      IF i is even*
         THEN
            DEST[i+63:i] :=
                RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
         ELSE
            DEST[i+63:i] :=
                RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
         FI
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
            DEST[i+63:i] := 0
      FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
   THEN
      IF i is even*
         THEN
            IF (EVEX.b = 1)
               THEN
                   DEST[i+63:i] :=
                       RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
               ELSE
                   DEST[i+63:i] :=
                       RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
               FI;
         ELSE
            IF (EVEX.b = 1)
               THEN
                   DEST[i+63:i] :=
                       RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
               ELSE
                   DEST[i+63:i] :=
                       RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
               FI
         FI
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF j *is even*
    THEN
      IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
      ELSE
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
      FI;
    ELSE
      IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
      ELSE
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
      FI;
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN
      *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF j *is even*
    THEN
      IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
      ELSE
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
      FI;
    ELSE
      IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
      ELSE
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
      FI;
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN
      *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision
THEN
DEST[i+63;i] :=
RoundFPControl_MXCSR(SRC2[i+63;i]*DEST[i+63;i] + SRC3[i+63;0])
ELSE
DEST[i+63;i] :=
RoundFPControl_MXCSR(SRC2[i+63;i]*DEST[i+63;i] + SRC3[i+63;i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63;i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63;i] := 0
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN
IF j *is even*
THEN DEST[i+63;i] :=
RoundFPControl(SRC2[i+63;i]*SRC3[i+63;i] - DEST[i+63;i])
ELSE DEST[i+63;i] :=
RoundFPControl(SRC2[i+63;i]*SRC3[i+63;i] + DEST[i+63;i])
FI
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63;i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63;i] := 0
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN
IF j *is even*
THEN
IF (EVEX.b = 1)
THEN
   DEST[i+63:i] :=
   RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
ELSE
   DEST[i+63:i] :=
   RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
FI;
ELSE
IF (EVEX.b = 1)
THEN
   DEST[i+63:i] :=
   RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
ELSE
   DEST[i+63:i] :=
   RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
FI;
ELSE
   IF *merging-masking* ; merging-masking
   THEN *DEST[i+63:i] remains unchanged*
   ELSE ; zeroing-masking
      DEST[i+63:i] := 0
   FI
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPD __m512d _mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d _mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m256d _mm256_mask_fmaddsub_round_pd(__m256d a, __mmask8 k, __m256d b, __m256d c, int r);
VFMADDSUBxxxPD __m256d _mm256_maskz_fmaddsub_round_pd(__mmask8 k, __m256d a, __m256d b, __m256d c, int r);
VFMADDSUBxxxPD __m256d _mm256_mask3_fmaddsub_round_pd(__m256d a, __m256d b, __m256d c, __mmask8 k, int r);
SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values

### Description

This instruction performs a packed multiply-add (odd elements) or multiply-subtract (even elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation “132”, “213” and “231” indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-8.

The destination elements are updated according to the writemask.

---

### Instruction Set Encoding

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP6.W0 96 /r VFMADDSUB132PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, add/subtract elements in xmm2, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 96 /r VFMADDSUB132PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, add/subtract elements in ymm2, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 96 /r VFMADDSUB132PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, add/subtract elements in zmm2, and store the result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.W0 A6 /r VFMADDSUB213PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 A6 /r VFMADDSUB213PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 A6 /r VFMADDSUB213PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.W0 B6 /r VFMADDSUB231PH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, add/subtract elements in xmm1, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 B6 /r VFMADDSUB231PH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, add/subtract elements in ymm1, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 B6 /r VFMADDSUB231PH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, add/subtract elements in zmm1, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

The destination elements are updated according to the writemask.
Table 5-5. VFMADDSUB[132,213,231]PH Notation for Odd and Even Elements

<table>
<thead>
<tr>
<th>Notation</th>
<th>Odd Elements</th>
<th>Even Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>dest = dest*src3+src2</td>
<td>dest = dest*src3-src2</td>
</tr>
<tr>
<td>231</td>
<td>dest = src2*src3+dest</td>
<td>dest = src2*src3-dest</td>
</tr>
<tr>
<td>213</td>
<td>dest = src2*dest+src3</td>
<td>dest = src2*dest-src3</td>
</tr>
</tbody>
</table>

Operation

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
  SET_RM(EVEX.RC)
ELSE
  SET_RM(MXCSR.RC)
FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF *j is even*:
      DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] - SRC2.fp16[j])
    ELSE:
      DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC3.fp16[j] + SRC2.fp16[j])
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
  // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0

VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      t3 := SRC3.fp16[0]
    ELSE:
      t3 := SRC3.fp16[j]
    IF *j is even*:
      DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
    ELSE:
      DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
  // else dest.fp16[j] remains unchanged
DEST[MAXVL-1:VL] := 0
VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
      ELSE
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] + SRC3.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
         // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3)
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3)
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
         // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR "no writemask":
      IF *j is even:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] - DEST.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] + DEST.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR "no writemask":
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C++ Compiler Intrinsic Equivalent

VFMADDSUB132PH, VFMADDSUB213PH, and VFMADDSUB231PH:

__m128h _mm_fmaddsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmaddsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmaddsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmaddsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmaddsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmaddsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmaddsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmaddsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmaddsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmaddsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmaddsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmaddsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmaddsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmaddsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
### VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 96 / r VFMADDSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 A6 / r VFMADDSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 B6 / r VFMADDSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 96 / r VFMADDSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 A6 / r VFMADDSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 B6 / r VFMADDSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A6 / r VFMADDSUB213PS xmm1 (k1)z1, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 B6 / r VFMADDSUB231PS xmm1 (k1)z1, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A6 / r VFMADDSUB213PS ymm1 (k1)z1, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 B6 / r VFMADDSUB231PS ymm1 (k1)z1, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A6 / r VFMADDSUB213PS zmm1 (k1)z1, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 B6 / r VFMADDSUB231PS zmm1 (k1)z1, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 96 / r VFMADDSUB213PS zmm1 (k1)z1, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Set Reference, V**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
  n := 64*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFMADDSUB213PS DEST, SRC2, SRC3
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
  n := 64*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFMADDSUB231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
  n := 64*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN DEST[i+31:i] :=
          RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
  ENDIF;
ENDIF
ELSE DEST[i+31:i] :=
    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
FI
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMAADDPS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] :=
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] :=
                                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[i+31:i] :=
                                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                            ELSE
                                DEST[i+31:i] :=
                                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                            FI;
                    FI
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] := 0
                FI
        FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF j *is even*
    THEN DEST[i+31:i] :=
      RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
    ELSE DEST[i+31:i] :=
      RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    FI;
ELSE
  IF *merging-masking* ; merging-masking
  THEN *DEST[i+31:i] remains unchanged*
  ELSE ; zeroing-masking
    DEST[i+31:i] := 0
  FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] :=
        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
    ELSE
      DEST[i+31:i] :=
        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    FI;
  ELSE
    IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] :=
        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
    ELSE
      DEST[i+31:i] :=
        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    FI;
  FI;
IF
ELSE
    IF *merging-masking*
    THEN *DEST[i+31:i] remains unchanged*
    ELSE
        zeroing-masking
        DEST[i+31:i] := 0
    FI
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
        THEN
            IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] :=
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
                DEST[i+31:i] :=
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
            ELSE
                zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
        THEN
            IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] :=
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
                DEST[i+31:i] :=
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
            ELSE
                zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
FI;
ELSE
IF (EVEX.b = 1)
THEN
DEST[i+31:i] :=
RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
ELSE
DEST[i+31:i] :=
RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
FI;
ELSE
IF *merging-masking*
THEN *DEST[i+31:i] remains unchanged*
ELSE
DEST[i+31:i] := 0
FI
ENDIF
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDSUBxxPS __m512 _mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxPS __m512 _mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxPS __m512 _mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxPS __m512 _mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxPS __m512 _mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxPS __m512 _mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxPS __m512 _mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxPS __m512 _mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxPS __m256 _mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxPS __m256 _mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxPS __m256 _mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxPS __m256 _mm256_maskz_fmaddsub_round_ps(__mmask8 k, __m256 a, __m256 b, __m256 c, int r);
VFMADDSUBxxPS __m256 _mm256_mask3_fmaddsub_round_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
### VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 AA /r VFMSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 BA /r VFMSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1[k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 AA /r VFMSUB213PD xmm1[k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 BA /r VFMSUB231PD xmm1[k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 9A /r VFMSUB132PD ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 AA /r VFMSUB213PD ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 BA /r VFMSUB231PD ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 9A /r VFMSUB132PD zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm3/mem, subtract zmm2 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 AA /r VFMSUB213PD zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm2, subtract zmm3/mem and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 BA /r VFMSUB231PD zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm2 and zmm3/mem, subtract zmm1 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
Description
Performs a set of SIMD multiply-subtract computation on packed double precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.
VFMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).
VFMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).
VFMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).
EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.
VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.
VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation
In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).
VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] :=
          RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] :=
          RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
      FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] :=
    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
        THEN
            DEST[i+63:i] :=
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        ELSE
            DEST[i+63:i] :=
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
    FI;
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
  ELSE ; zeroing-masking
    DEST[i+63:i] := 0
  FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] :=
      RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] :=
          RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
        ELSE
          DEST[i+63:i] :=
          RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_maskfmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_maskfmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_fmsub_round_pd(__m256d a, __m256d b, __m256d c, int r);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m256d _mm256_maskfmsub_round_pd(__m256d a, __m256d b, __m256d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m256d _mm256_maskfmsub_round_pd(__m256d a, __m256d b, __m256d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_round_pd(__m256d a, __m256d b, __m256d c, int r);
VFMSUBxxxPD __m128d _mm128_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm128_fmsub_round_pd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxPD __m128d _mm128_mask_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm128_maskfmsub_round_pd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxPD __m128d _mm128_maskz_fmsub_pd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm128_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm128_maskfmsub_round_pd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxPD __m128d _mm128_maskz_fmsub_round_pd(__m128d a, __m128d b, __m128d c, int r);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VF[N]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMSUB132PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract xmm2, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB132PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract ymm2, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB132PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract zmm2, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB213PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB213PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB213PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

VFNMSUB[132,213,231]PH

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFNMSUB132PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract xmm2, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFNMSUB132PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract ymm2, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFNMSUB132PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract zmm2, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

VFMSUB213PH

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMSUB213PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB213PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB213PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

VFNMSUB213PH

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFNMSUB213PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFNMSUB213PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFNMSUB213PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

VFMSUB231PH

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMSUB231PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB231PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB231PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

VFMSUB231PH

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMSUB231PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, subtract xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB231PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, subtract ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB231PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, subtract zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
VF\[,N\]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values

### Description

This instruction performs a packed multiply-subtract or a negated multiply-subtract computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation "132", "213" and "231" indicate the use of the operands in \(\pm A \times B - C\), where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-6.

The destination elements are updated according to the writemask.

### Table 5-6. VF\[,N\]MSUB[132,213,231]PH Notation for Operands

<table>
<thead>
<tr>
<th>Notation</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>dest = (\pm dest\times src3 - src2)</td>
</tr>
<tr>
<td>231</td>
<td>dest = (\pm src2 \times src3 - dest)</td>
</tr>
<tr>
<td>213</td>
<td>dest = (\pm src2 \times dest - src3)</td>
</tr>
</tbody>
</table>
**Operation**

VF[\(N\)]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

VL = 128, 256 or 512

\(KL := \frac{VL}{16}\)

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR \(j := 0\) TO \(KL - 1\):
   IF \(k1[j]\) OR *no writemask*:
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * SRC2.fp16[j] - SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * SRC2.fp16[j] - SRC2.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged
   DEST[MAXVL-1:VL] := 0

VF[\(N\)]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

VL = 128, 256 or 512

\(KL := \frac{VL}{16}\)

FOR \(j := 0\) TO \(KL - 1\):
   IF \(k1[j]\) OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] * t3 - SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged
   DEST[MAXVL-1:VL] := 0
VF[\text{N}]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register

VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
      ELSE
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*DEST.fp16[j] - SRC3.fp16[j])
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged
   DEST[MAXVL-1:VL] := 0

VF[\text{N}]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source

VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *negative form*:
         DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * DEST.fp16[j] - t3)
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3)
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged
   DEST[MAXVL-1:VL] := 0
VF[N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *negative form:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]*SRC3.fp16[j] - DEST.fp16[j])
        ELSE IF *zeroing*:
            DEST.fp16[j] := 0
            // else dest.fp16[j] remains unchanged
    ELSE:
        DEST[MAXVL-1:VL] := 0

VF[N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *negative form*:
            DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] * t3 - DEST.fp16[j])
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j])
        ELSE IF *zeroing*:
            DEST.fp16[j] := 0
            // else dest.fp16[j] remains unchanged
    ELSE:
        DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMSUB132PH, VFMSUB213PH, and VFMSUB231PH:

\[
\begin{align*}
&\text{__m128h } \text{_mm_fmsub_ph}(\text{__m128h } a, \text{__m128h } b, \text{__m128h } c); \\
&\text{__m128h } \text{_mm_mask_fmsub_ph}(\text{__m128h } a, \text{__mmask8 } k, \text{__m128h } b, \text{__m128h } c); \\
&\text{__m128h } \text{_mm_mask3_fmsub_ph}(\text{__m128h } a, \text{__m128h } b, \text{__m128h } c, \text{__mmask8 } k); \\
&\text{__m128h } \text{_mm_maskz_fmsub_ph}(\text{__mmask8 } k, \text{__m128h } a, \text{__m128h } b, \text{__m128h } c); \\
&\text{__m256h } \text{_mm256_fmsub_ph}(\text{__m256h } a, \text{__m256h } b, \text{__m256h } c); \\
&\text{__m256h } \text{_mm256_mask_fmsub_ph}(\text{__m256h } a, \text{__mmask16 } k, \text{__m256h } b, \text{__m256h } c); \\
&\text{__m256h } \text{_mm256_mask3_fmsub_ph}(\text{__m256h } a, \text{__m256h } b, \text{__m256h } c, \text{__mmask16 } k); \\
&\text{__m256h } \text{_mm256_maskz_fmsub_ph}(\text{__mmask16 } k, \text{__m256h } a, \text{__m256h } b, \text{__m256h } c); \\
&\text{__m512h } \text{_mm512_fmsub_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c); \\
&\text{__m512h } \text{_mm512_mask_fmsub_ph}(\text{__m512h } a, \text{__mmask32 } k, \text{__m512h } b, \text{__m512h } c); \\
&\text{__m512h } \text{_mm512_mask3_fmsub_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{__mmask32 } k); \\
&\text{__m512h } \text{_mm512_maskz_fmsub_ph}(\text{__mmask32 } k, \text{__m512h } a, \text{__m512h } b, \text{__m512h } c); \\
&\text{__m512h } \text{_mm512_fmsub_round_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{const int rounding}); \\
&\text{__m512h } \text{_mm512_mask_fmsub_round_ph}(\text{__m512h } a, \text{__mmask32 } k, \text{__m512h } b, \text{__m512h } c, \text{const int rounding}); \\
&\text{__m512h } \text{_mm512_mask3_fmsub_round_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{__mmask32 } k, \text{const int rounding}); \\
&\text{__m512h } \text{_mm512_maskz_fmsub_round_ph}(\text{__mmask32 } k, \text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{const int rounding}); \\
\end{align*}
\]

VFNMSUB132PH, VFNMSUB213PH, and VFNMSUB231PH:

\[
\begin{align*}
&\text{__m128h } \text{_mm_fnmsub_ph}(\text{__m128h } a, \text{__m128h } b, \text{__m128h } c); \\
&\text{__m128h } \text{_mm_mask_fnmsub_ph}(\text{__m128h } a, \text{__mmask8 } k, \text{__m128h } b, \text{__m128h } c); \\
&\text{__m128h } \text{_mm_mask3_fnmsub_ph}(\text{__m128h } a, \text{__m128h } b, \text{__m128h } c, \text{__mmask8 } k); \\
&\text{__m128h } \text{_mm_maskz_fnmsub_ph}(\text{__mmask8 } k, \text{__m128h } a, \text{__m128h } b, \text{__m128h } c); \\
&\text{__m256h } \text{_mm256_fnmsub_ph}(\text{__m256h } a, \text{__m256h } b, \text{__m256h } c); \\
&\text{__m256h } \text{_mm256_mask_fnmsub_ph}(\text{__m256h } a, \text{__mmask16 } k, \text{__m256h } b, \text{__m256h } c); \\
&\text{__m256h } \text{_mm256_mask3_fnmsub_ph}(\text{__m256h } a, \text{__m256h } b, \text{__m256h } c, \text{__mmask16 } k); \\
&\text{__m256h } \text{_mm256_maskz_fnmsub_ph}(\text{__mmask16 } k, \text{__m256h } a, \text{__m256h } b, \text{__m256h } c); \\
&\text{__m512h } \text{_mm512_fnmsub_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c); \\
&\text{__m512h } \text{_mm512_mask_fnmsub_ph}(\text{__m512h } a, \text{__mmask32 } k, \text{__m512h } b, \text{__m512h } c); \\
&\text{__m512h } \text{_mm512_mask3_fnmsub_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{__mmask32 } k); \\
&\text{__m512h } \text{_mm512_maskz_fnmsub_ph}(\text{__mmask32 } k, \text{__m512h } a, \text{__m512h } b, \text{__m512h } c); \\
&\text{__m512h } \text{_mm512_fmsub_round_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{const int rounding}); \\
&\text{__m512h } \text{_mm512_mask_fnmsub_round_ph}(\text{__m512h } a, \text{__mmask32 } k, \text{__m512h } b, \text{__m512h } c, \text{const int rounding}); \\
&\text{__m512h } \text{_mm512_mask3_fnmsub_round_ph}(\text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{__mmask32 } k, \text{const int rounding}); \\
&\text{__m512h } \text{_mm512_maskz_fnmsub_round_ph}(\text{__mmask32 } k, \text{__m512h } a, \text{__m512h } b, \text{__m512h } c, \text{const int rounding}); \\
\end{align*}
\]

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
### VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 [k1][z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 [k1][z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 [k1][z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single Precision Floating-Point Values

**Description**

Performs a set of SIMD multiply-subtract computation on packed single precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, "+" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
```

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version)

IF (VL = 512) AND (EVEX.b = 1) THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0

VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI

For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI

For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(Dest[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(Dest[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN Dest[i+31:i] :=
      RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
        ELSE
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
THEN
  IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] :=
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
    ELSE
      DEST[i+31:i] :=
      RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDIF
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxPS __m512 _mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_maskz_fmsub_ps (__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask3_fmsub_ps (__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_fmsub_ps(__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m512 _mm512_fmsub_ps (__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double Precision Floating-Point Values

**Description**

Performs a SIMD multiply-subtract computation on the low packed double precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

VFMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Opcode/Description**

- **VEX.LIG.66.0F38.W1 9B /r**: VFMSUB132SD xmm1, xmm2, xmm3/m64
  - Description: Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
- **VEX.LIG.66.0F38.W1 AB /r**: VFMSUB213SD xmm1, xmm2, xmm3/m64
  - Description: Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
- **VEX.LIG.66.0F38.W1 BB /r**: VFMSUB231SD xmm1, xmm2, xmm3/m64
  - Description: Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
- **EVEX.LLIG.66.0F38.W1 9B /r**: VFMSUB132SD xmm1 (k1)[z], xmm2, xmm3/m64[er]
  - Description: Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
- **EVEX.LLIG.66.0F38.W1 AB /r**: VFMSUB213SD xmm1 (k1)[z], xmm2, xmm3/m64[er]
  - Description: Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
- **EVEX.LLIG.66.0F38.W1 BB /r**: VFMSUB231SD xmm1 (k1)[z], xmm2, xmm3/m64[er]
  - Description: Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI;

IF k1[0] or *no writemask*
   THEN DEST[63:0] := RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
   ELSE
       IF *merging-masking* ; merging-masking
          THEN *DEST[63:0] remains unchanged*
          ELSE ; zeroing-masking
              THEN DEST[63:0] := 0
          FI;
   FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

**VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI;

IF k1[0] or *no writemask*
   THEN DEST[63:0] := RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
   ELSE
       IF *merging-masking* ; merging-masking
          THEN *DEST[63:0] remains unchanged*
          ELSE ; zeroing-masking
              THEN DEST[63:0] := 0
          FI;
   FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN  DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMSUBL132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMSUBL213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMSUBL231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VF[N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 9B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm3/m16, subtract xmm2, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 AB /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm2, subtract xmm3/m16, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 BB /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm2 and xmm3/m16, subtract xmm1, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 9F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm3/m16, and negate the value. Subtract xmm2 from this value, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 AF /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm1 and xmm2, and negate the value. Subtract xmm3/m16 from this value, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 BF /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply FP16 values from xmm2 and xmm3/m16, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Table 5-7. VF[N]MSUB[132,213,231]SH Notation for Operands

<table>
<thead>
<tr>
<th>Notation</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>dest = ± dest*src3-src2</td>
</tr>
<tr>
<td>231</td>
<td>dest = ± src2*src3-dest</td>
</tr>
<tr>
<td>213</td>
<td>dest = ± src2*dest-src3</td>
</tr>
</tbody>
</table>

Description

This instruction performs a scalar multiply-subtract or negated multiply-subtract computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation "132", "213" and "231" indicate the use of the operands in ±A * B − C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-7.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.
**Operation**

**VF[.N]MSUB132SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:
  SET_Rm(EVEX.RC)
ELSE
  SET_Rm(MXCSR.RC)

IF k1[0] OR *no writemask*:
  IF *negative form*:
    DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
  ELSE:
    DEST.fp16[0] := RoundFPControl(DEST.fp16[0]*SRC3.fp16[0] - SRC2.fp16[0])
ELSE IF *zeroing*:
  DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0

**VF[.N]MSUB213SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:
  SET_Rm(EVEX.RC)
ELSE
  SET_Rm(MXCSR.RC)

IF k1[0] OR *no writemask*:
  IF *negative form*:
    DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
  ELSE:
    DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*DEST.fp16[0] - SRC3.fp16[0])
ELSE IF *zeroing*:
  DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0

**VF[.N]MSUB231SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:
  SET_Rm(EVEX.RC)
ELSE
  SET_Rm(MXCSR.RC)

IF k1[0] OR *no writemask*:
  IF *negative form*:
    DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
  ELSE:
    DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]*SRC3.fp16[0] - DEST.fp16[0])
ELSE IF *zeroing*:
  DEST.fp16[0] := 0
// else DEST.fp16[0] remains unchanged

DEST[127:16] remains unchanged
DEST[MAXVL-1:128] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUB132SH, VFMSUB213SH, and VFMSUB231SH:

- `__m128h _mm_fmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_mask_fmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_mask3_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_maskz_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_fmsub_sh (__m128h a, __m128h b, __m128h c);`
- `__m128h _mm_mask_fmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);`
- `__m128h _mm_mask3_fmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);`
- `__m128h _mm_maskz_fmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);`

VFNMSUB132SH, VFNMSUB213SH, and VFNMSUB231SH:

- `__m128h _mm_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_mask_fnmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_mask3_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_maskz_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);`
- `__m128h _mm_fnmsub_sh (__m128h a, __m128h b, __m128h c);`
- `__m128h _mm_mask_fnmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);`
- `__m128h _mm_mask3_fnmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);`
- `__m128h _mm_maskz_fnmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);`

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W0 9B /r VFMFB132SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 AB /r VFMFB213SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 BB /r VFMFB231SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 9B /r VFMFB132SS xmm1 (k1)[z], xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 AB /r VFMFB213SS xmm1 (k1)[z], xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 BB /r VFMFB231SS xmm1 (k1)[z], xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD multiply-subtract computation on the low packed single precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single precision floating-point value from the second source to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point value in the first source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).
VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] := 0

FI;

FI;


DEST[MAXVL-1:128] := 0

**VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] := RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] := 0

FI;

FI;


DEST[MAXVL-1:128] := 0
VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;

IF k1[0] or *no writemask*
  THEN
    DEST[31:0] := RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] := 0
    FI;
  FI;

DEST[MAXVL-1:128] := 0

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[MAXVL-1:128] := 0

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[MAXVL-1:128] := 0

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_fmsub_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __m128 d, __mmask8 k);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __m128 d, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __m128 d, __mmask8 k);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
VFMSUBADD132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double precision floating-point values from the second source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double precision floating-point elements and adds the even double precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM/reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM/reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "\*" and "\+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

\[
\text{DEST}[63:0] := \text{RoundFPControl_MXCSR(DEST}[63:0]\ast \text{SRC3}[63:0] + \text{SRC2}[63:0])
\]
\[
\text{DEST}[127:64] := \text{RoundFPControl_MXCSR(DEST}[127:64]\ast \text{SRC3}[127:64] - \text{SRC2}[127:64])
\]
\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
ELSEIF (VEX.256)

\[
\text{DEST}[63:0] := \text{RoundFPControl_MXCSR(DEST}[63:0]\ast \text{SRC3}[63:0] + \text{SRC2}[63:0])
\]
\[
\text{DEST}[127:64] := \text{RoundFPControl_MXCSR(DEST}[127:64]\ast \text{SRC3}[127:64] - \text{SRC2}[127:64])
\]
\[
\]
\[
\]
FI

**VFMSUBADD213PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

\[
\text{DEST}[63:0] := \text{RoundFPControl_MXCSR(SRC2}[63:0]\ast \text{DEST}[63:0] + \text{SRC3}[63:0])
\]
\[
\text{DEST}[127:64] := \text{RoundFPControl_MXCSR(SRC2}[127:64]\ast \text{DEST}[127:64] - \text{SRC3}[127:64])
\]
\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
ELSEIF (VEX.256)

\[
\text{DEST}[63:0] := \text{RoundFPControl_MXCSR(SRC2}[63:0]\ast \text{DEST}[63:0] + \text{SRC3}[63:0])
\]
\[
\text{DEST}[127:64] := \text{RoundFPControl_MXCSR(SRC2}[127:64]\ast \text{DEST}[127:64] - \text{SRC3}[127:64])
\]
\[
\]
\[
\]
FI

**VFMSUBADD231PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

\[
\text{DEST}[63:0] := \text{RoundFPControl_MXCSR(SRC2}[63:0]\ast \text{SRC3}[63:0] + \text{DEST}[63:0])
\]
\[
\text{DEST}[127:64] := \text{RoundFPControl_MXCSR(SRC2}[127:64]\ast \text{SRC3}[127:64] - \text{DEST}[127:64])
\]
\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
ELSEIF (VEX.256)

\[
\text{DEST}[63:0] := \text{RoundFPControl_MXCSR(SRC2}[63:0]\ast \text{SRC3}[63:0] + \text{DEST}[63:0])
\]
\[
\text{DEST}[127:64] := \text{RoundFPControl_MXCSR(SRC2}[127:64]\ast \text{SRC3}[127:64] - \text{DEST}[127:64])
\]
\[
\]
\[
\]
FI

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

IF (VL = 512) AND (EVEX.b = 1)

THEN

\[\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC)};\]

ELSE

\[\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC)};\]

FI

FOR \(j := 0\) TO \(KL-1\)

\[i := j \ast 64\]

IF \(k1[j]\) OR *no writemask*
THEN
  IF j *is even*  
    THEN DEST[i+63:i] :=
        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
    ELSE DEST[i+63:i] :=
        RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
  FI
ELSE
  IF *merging-masking* ; merging-masking  
    THEN *DEST[i+63:i] remains unchanged*  
    ELSE ; zeroing-masking  
        DEST[i+63:i] := 0
  FI
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*  
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF (EVEX.b = 1)
              THEN
                  DEST[i+63:i] :=
                      RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
              ELSE
                  DEST[i+63:i] :=
                      RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
              FI;
        ELSE
          IF *merging-masking* ; merging-masking  
            THEN *DEST[i+63:i] remains unchanged*  
            ELSE ; zeroing-masking  
                DEST[i+63:i] := 0
            FI
          FI
      FI
  ENDIF
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
            THEN DEST[i+63:i] :=
                RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        FI
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
            THEN IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                    ELSE
                        DEST[i+63:i] :=
                            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                FI;
            ELSE
                IF (EVEX.b = 1)
                    THEN
                        DEST[i+63:i] :=
                            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                    ELSE
                        DEST[i+63:i] :=
                            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                FI;
        ELSE
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                    ELSE
                        DEST[i+63:i] :=
                            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                FI;
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply–Alternating Subtract/Add of Packed Double Precision

INSTRUCTION SET REFERENCE, V

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN
IF j *is even*
THEN
DEST[i+63:i] :=
RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
ELSE
DEST[i+63:i] :=
RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
FI
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN
IF j *is even*
THEN
IF (EVEX.b = 1)
THEN
DEST[i+63:i] :=
RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
ELSE
DEST[i+63:i] :=
RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
FI
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
FI;
ELSE
IF (EVEX.b = 1)
THEN
DEST[i+63:i] :=
RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
ELSE
DEST[i+63:i] :=
RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
FI;
ELSE
IF *merging-masking*
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] := 0
FI
ENDFOR
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUBADDxxxPD __m512d _mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm128_mask_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm128_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm128_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m256d __mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP6.W0 97 /r VFMSUBADD132PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract/add elements in xmm2, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 97 /r VFMSUBADD132PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract/add elements in ymm2, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 97 /r VFMSUBADD132PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract/add elements in zmm2, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.W0 A7 /r VFMSUBADD213PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 A7 /r VFMSUBADD213PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 A7 /r VFMSUBADD213PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP6.W0 B7 /r VFMSUBADD231PH xmm1[k1][z], xmm2, xmm3/m128/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract/add elements in xmm1, and store the result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 B7 /r VFMSUBADD231PH ymm1[k1][z], ymm2, ymm3/m256/m16bcst</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract/add elements in ymm1, and store the result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 B7 /r VFMSUBADD231PH zmm1[k1][z], zmm2, zmm3/m512/m16bcst {er}</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract/add elements in zmm1, and store the result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a packed multiply-add (even elements) or multiply-subtract (odd elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation "132", "213" and "231" indicate the use of the operands in A * B ± C, where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-8. The destination elements are updated according to the writemask.
**Table 5-8. VFMSUBADD[132,213,231]PH Notation for Odd and Even Elements**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Odd Elements</th>
<th>Even Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>dest = dest*src3-src2</td>
<td>dest = dest*src3+src2</td>
</tr>
<tr>
<td>231</td>
<td>dest = src2*src3-dest</td>
<td>dest = src2*src3+dest</td>
</tr>
<tr>
<td>213</td>
<td>dest = src2*dest-src3</td>
<td>dest = src2*dest+src3</td>
</tr>
</tbody>
</table>

**Operation**

**VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] + SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j]*SRC3.fp16[j] - SRC2.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 + SRC2.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(DEST.fp16[j] * t3 - SRC2.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0:
VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + SRC3.fp16[j])
        ELSE
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - SRC3.fp16[j])
        ELSE IF *zeroing*:
            DEST.fp16[j] := 0
        // else dest.fp16[j] remains unchanged
    DEST[MAXVL-1:VL] := 0

VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        IF EVEX.b = 1:
            t3 := SRC3.fp16[0]
        ELSE:
            t3 := SRC3.fp16[j]
        IF *j is even*:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] + t3 )
        ELSE:
            DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * DEST.fp16[j] - t3 )
        ELSE IF *zeroing*:
            DEST.fp16[j] := 0
        // else dest.fp16[j] remains unchanged
    DEST[MAXVL-1:VL] := 0:
VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register
VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
   SET_RM(EVEX.RC)
ELSE
   SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] + DEST.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * SRC3.fp16[j] - DEST.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         t3 := SRC3.fp16[0]
      ELSE:
         t3 := SRC3.fp16[j]
      IF *j is even*:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 + DEST.fp16[j])
      ELSE:
         DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] * t3 - DEST.fp16[j])
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADD132PH, VFMSUBADD213PH, and VFMSUBADD231PH:

│ __m128h _mm_fmsubadd_ph (__m128h a, __m128h b, __m128h c); │
│ __m128h _mm_mask_fmsubadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c); │
│ __m128h _mm_mask3_fmsubadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k); │
│ __m128h _mm_maskz_fmsubadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c); │
│ __m256h _mm256_fmsubadd_ph (__m256h a, __m256h b, __m256h c); │
│ __m256h _mm256_mask_fmsubadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c); │
│ __m256h _mm256_mask3_fmsubadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k); │
│ __m256h _mm256_maskz_fmsubadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c); │
│ __m512h _mm512_fmsubadd_ph (__m512h a, __m512h b, __m512h c); │
│ __m512h _mm512_mask_fmsubadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c); │
│ __m512h _mm512_mask3_fmsubadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k); │
│ __m512h _mm512_maskz_fmsubadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c); │
│ __m512h _mm512_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding); │
│ __m512h _mm512_mask_fmsubadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding); │
│ __m512h _mm512_mask3_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding); │
│ __m512h _mm512_maskz_fmsubadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding); │

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
### VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values in the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values in the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values in the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
Operation
In the operations below, ‘*’ and ‘+’ symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFMSUBADD213PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFMSUBADD231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

IF \((VL = 512)\) AND \((\text{EVEX}.b = 1)\)

THEN

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION}(\text{EVEX}.\text{RC});
\]

ELSE

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION}(\text{MXCSR}.\text{RC});
\]

FI;

FOR \(j := 0\) TO \(KL-1\)

\(i := j \times 32\)

IF \(k1[j] \) OR *no writemask*

THEN

IF \(j \) *is even*

THEN

\[
\text{DEST}[i+31:i] := \text{RoundFPControl}(\text{DEST}[i+31:i] \times \text{SRC3}[i+31:i] + \text{SRC2}[i+31:i])
\]

ELSE

\[
\text{DEST}[i+31:i] := \text{RoundFPControl}(\text{DEST}[i+31:i] \times \text{SRC3}[i+31:i] - \text{SRC2}[i+31:i])
\]

FI

ELSE

IF *merging-masking* ; merging-masking

THEN *\(\text{DEST}[i+31:i]\) remains unchanged*

ELSE ; zeroing-masking

\[
\text{DEST}[i+31:i] := 0
\]

FI

ENDFOR

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\(i := j \times 32\)

IF \(k1[j] \) OR *no writemask*

THEN

IF \(j \) *is even*

THEN

IF \((\text{EVEX}.b = 1)\)

THEN

\[
\text{DEST}[i+31:i] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[i+31:i] \times \text{SRC3}[31:0] + \text{SRC2}[i+31:i])
\]

ELSE

\[
\text{DEST}[i+31:i] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[i+31:i] \times \text{SRC3}[i+31:i] + \text{SRC2}[i+31:i])
\]

FI

ELSE

IF \((\text{EVEX}.b = 1)\)

THEN

\[
\text{DEST}[i+31:i] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[i+31:i] \times \text{SRC3}[31:0] - \text{SRC2}[i+31:i])
\]

ELSE

\[
\text{DEST}[i+31:i] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[i+31:i] \times \text{SRC3}[i+31:i] - \text{SRC2}[i+31:i])
\]

\]
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single Precision

INSTRUCTION SET REFERENCE, V

Vol. 2C 5-287

**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
ELSE
   DEST[i+31:i] :=
   RoundFPControl_MXCSR(SRC2[i+31:i] * DEST[i+31:i] + SRC3[i+31:i])
   FI;
ELSE
IF (EVEX.b = 1)
   THEN
   DEST[i+31:i] :=
   RoundFPControl_MXCSR(SRC2[i+31:i] * DEST[i+31:i] - SRC3[i+31:i])
   ELSE
   DEST[i+31:i] :=
   RoundFPControl_MXCSR(SRC2[i+31:i] * DEST[i+31:i] - SRC3[31:0])
   FI;
ELSE
IF *merging-masking* ; merging-masking
   THEN *DEST[i+31:i] remains unchanged*
   ELSE ; zeroing-masking
   DEST[i+31:i] := 0
   FI
FI
ENDIF
ELSE
DEST[MAXVL-1:VL] := 0

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI;
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN
      IF j *is even*
         THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i] * SRC3[i+31:i] + DEST[i+31:i])
         ELSE DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i] * SRC3[i+31:i] - DEST[i+31:i])
         FI
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+31:i] := 0
         FI
      FI
   ENDIF
DEST[MAXVL-1:VL] := 0

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] :=
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
            ELSE
              DEST[i+31:i] :=
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] :=
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
              DEST[i+31:i] :=
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
          FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI
  FI
ENDDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADDxxxPS __m512 _mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps(__m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_mask_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m128 _mm128_fmsubadd_ps(__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm128_mask_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm128_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps(__m256 a, __m256 b, __m256 c);
SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
**VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V FMA</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V FMA</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 AC /r VFNMADD213PD xmm0 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 BC /r VFNMADD231PD xmm0 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 9C /r VFNMADD132PD ymm0 {k1}{z}, ymm1, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 AC /r VFNMADD213PD ymm0 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 BC /r VFNMADD231PD ymm0 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
</tbody>
</table>
VFNMADD132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation
In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).
VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
    FOR j := 0 TO KL-1
        i := j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+63:i] :=
                RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0
VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
    ENDFOR
    DEST[MAXVL-1:VL] := 0
VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] :=
    RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
  ELSE*
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
    DEST[i+63:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+63:i] :=
        RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
    ELSE
      DEST[i+63:i] :=
        RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
    FI;
  ELSE*
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
    DEST[i+63:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd (__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd (__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd (__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_round_pd (__m256d a, __mmask8 k, __m256d b, __m256d c, int r);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_round_pd (__mmask8 k, __m256d a, __m256d b, __m256d c, int r);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_round_pd (__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m128d _mm128_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, "Type 2 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."
VFNMADD132PS/VFNADD213PS/VFNADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 AC /r VFNADD213PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 BC /r VFNADD231PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 9C /r VFNADD132PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 AC /r VFNADD213PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 BC /r VFNADD231PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 9C /r VFNADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 AC /r VFNADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 BC /r VFNADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 9C /r VFNADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX52F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 AC /r VFNADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 BC /r VFNADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 9C /r VFNADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 AC /r VFNADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 BC /r VFNADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values

### Instruction Set Reference, V

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM&lt;reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM&lt;r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM&lt;reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM&lt;r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

VFNMADD132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.128 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)**

```plaintext
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI
```

---

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single Precision Floating-Point Values
VFNADD132PS DEST, SRC2, SRC3 (EVEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFNADD213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:256] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFNADD231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFNADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] :=
    RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
          FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI;
  ENDFOR
  DEST[MAXVL-1:VL] := 0

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
    RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
  ENDFOR
  DEST[MAXVL-1:VL] := 0
VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR "no writemask"
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])
        ELSE
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
      FI;
    ELSE
      IF "merging-masking" ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
  ENDFOR
  DEST[MAXVL-1:VL] := 0

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR "no writemask"
    THEN DEST[i+31:i] :=
    RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
  ELSE
    IF "merging-masking" ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] :=
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
    ELSE
      DEST[i+31:i] :=
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
    FI;
  ELSE
    IF *merging-masking*
    THEN *DEST[i+31:i] remains unchanged*
    ELSE
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps(__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_round_ps(__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask16 k, __m256 b, __m256 c, __mmask16 k);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask16 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_round_ps(__m256 a, __mmask16 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_round_ps(__mmask16 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m128 _mm128_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm128_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm128_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm128_mask_fnmadd_round_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm128_maskz_fnmadd_round_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm128_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m64 _mm64_fnmadd_ps (__m64 a, __m64 b, __m64 c);
VFNMADDxxxPS __m64 _mm64_mask_fnmadd_ps(__m64 a, __mmask4 k, __m64 b, __m64 c, __mmask4 k);
VFNMADDxxxPS __m64 _mm64_maskz_fnmadd_ps(__mmask4 k, __m64 a, __m64 b, __m64 c);
VFNMADDxxxPS __m64 _mm64_mask_fnmadd_round_ps(__m64 a, __mmask4 k, __m64 b, __m64 c);
VFNMADDxxxPS __m64 _mm64_maskz_fnmadd_round_ps(__mmask4 k, __m64 a, __m64 b, __m64 c);
VFNMADDxxxPS __m64 _mm64_fnmadd_ps (__m64 a, __m64 b, __m64 c);
VFNMADDxxxPS __m32 _mm32_fnmadd_ps (__m32 a, __m32 b, __m32 c);
VFNMADDxxxPS __m32 _mm32_mask_fnmadd_ps(__m32 a, __mmask2 k, __m32 b, __m32 c, __mmask2 k);
VFNMADDxxxPS __m32 _mm32_maskz_fnmadd_ps(__mmask2 k, __m32 a, __m32 b, __m32 c);
VFNMADDxxxPS __m32 _mm32_mask_fnmadd_round_ps(__m32 a, __mmask2 k, __m32 b, __m32 c);
VFNMADDxxxPS __m32 _mm32_maskz_fnmadd_round_ps(__mmask2 k, __m32 a, __m32 b, __m32 c);
VFNMADDxxxPS __m32 _mm32_fnmadd_ps (__m32 a, __m32 b, __m32 c);
VFNMADDxxxPS __m16 _mm16_fnmadd_ps (__m16 a, __m16 b, __m16 c);
VFNMADDxxxPS __m16 _mm16_mask_fnmadd_ps(__m16 a, __mmask1 k, __m16 b, __m16 c, __mmask1 k);
VFNMADDxxxPS __m16 _mm16_maskz_fnmadd_ps(__mmask1 k, __m16 a, __m16 b, __m16 c);
VFNMADDxxxPS __m16 _mm16_mask_fnmadd_round_ps(__m16 a, __mmask1 k, __m16 b, __m16 c);
VFNMADDxxxPS __m16 _mm16_maskz_fnmadd_round_ps(__mmask1 k, __m16 a, __m16 b, __m16 c);
VFNMADDxxxPS __m16 _mm16_fnmadd_ps (__m16 a, __m16 b, __m16 c);
VFNMADDxxxPS __m8 _mm8_fnmadd_ps (__m8 a, __m8 b, __m8 c);
VFNMADDxxxPS __m8 _mm8_mask_fnmadd_ps(__m8 a, __mmask0 k, __m8 b, __m8 c, __mmask0 k);
VFNMADDxxxPS __m8 _mm8_maskz_fnmadd_ps(__mmask0 k, __m8 a, __m8 b, __m8 c);
VFNMADDxxxPS __m8 _mm8_mask_fnmadd_round_ps(__m8 a, __mmask0 k, __m8 b, __m8 c);
VFNMADDxxxPS __m8 _mm8_maskz_fnmadd_round_ps(__mmask0 k, __m8 a, __m8 b, __m8 c);
VFNMADDxxxPS __m8 _mm8_fnmadd_ps (__m8 a, __m8 b, __m8 c);
VFNMADDxxxPS __m4 _mm4_fnmadd_ps (__m4 a, __m4 b, __m4 c);
VFNMADDxxxPS __m4 _mm4_mask_fnmadd_ps(__m4 a, __mmask0 k, __m4 b, __m4 c, __mmask0 k);
VFNMADDxxxPS __m4 _mm4_maskz_fnmadd_ps(__mmask0 k, __m4 a, __m4 b, __m4 c);
VFNMADDxxxPS __m4 _mm4_mask_fnmadd_round_ps(__m4 a, __mmask0 k, __m4 b, __m4 c);
VFNMADDxxxPS __m4 _mm4_maskz_fnmadd_round_ps(__mmask0 k, __m4 a, __m4 b, __m4 c);
VFNMADDxxxPS __m4 _mm4_fnmadd_ps (__m4 a, __m4 b, __m4 c);
VFNMADDxxxPS __m2 _mm2_fnmadd_ps (__m2 a, __m2 b, __m2 c);
VFNMADDxxxPS __m2 _mm2_mask_fnmadd_ps(__m2 a, __mmask0 k, __m2 b, __m2 c, __mmask0 k);
VFNMADDxxxPS __m2 _mm2_maskz_fnmadd_ps(__mmask0 k, __m2 a, __m2 b, __m2 c);
VFNMADDxxxPS __m2 _mm2_mask_fnmadd_round_ps(__m2 a, __mmask0 k, __m2 b, __m2 c);
VFNMADDxxxPS __m2 _mm2_maskz_fnmadd_round_ps(__mmask0 k, __m2 a, __m2 b, __m2 c);
VFNMADDxxxPS __m2 _mm2_fnmadd_ps (__m2 a, __m2 b, __m2 c);
VFNMADDxxxPS __m1 _mm1_fnmadd_ps (__m1 a, __m1 b, __m1 c);
VFNMADDxxxPS __m1 _mm1_mask_fnmadd_ps(__m1 a, __mmask0 k, __m1 b, __m1 c, __mmask0 k);
VFNMADDxxxPS __m1 _mm1_maskz_fnmadd_ps(__mmask0 k, __m1 a, __m1 b, __m1 c);
VFNMADDxxxPS __m1 _mm1_mask_fnmadd_round_ps(__m1 a, __mmask0 k, __m1 b, __m1 c);
VFNMADDxxxPS __m1 _mm1_maskz_fnmadd_round_ps(__mmask0 k, __m1 a, __m1 b, __m1 c);
VFNMADDxxxPS __m1 _mm1_fnmadd_ps (__m1 a, __m1 b, __m1 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Negative Multiply-Add of Scalar Double Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 BitMode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W1 9D /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 AD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 BD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 9D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 AD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 BD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

VFMADD132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
    ELSE    IF *merging-masking* ; merging-masking
               THEN *DEST[63:0] remains unchanged*
               ELSE ; zeroing-masking
                     THEN DEST[63:0] := 0
                 FI;
    FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

**VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
    ELSE    IF *merging-masking* ; merging-masking
               THEN *DEST[63:0] remains unchanged*
               ELSE ; zeroing-masking
                     THEN DEST[63:0] := 0
                 FI;
    FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_maskfnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W0 9D /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 AD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 BD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 9D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 AD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 BD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**
In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
THEN
    DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
    THEN DEST[31:0] := 0
    FI;
FI;
DEST[MAXVL-1:128] := 0

**VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
THEN
    DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
    THEN DEST[31:0] := 0
    FI;
FI;
DEST[MAXVL-1:128] := 0
VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] := RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
           THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] := 0
        FI;
    FI;
DEST[MAXVL-1:128] := 0

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR( (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[MAXVL-1:128] := 0

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR( (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[MAXVL-1:128] := 0

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR( (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
## VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
</tbody>
</table>
### Description

VFNMSUB132PD: Multiplies the two, four or eight packed double precision floating-point values from the first source operand to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double precision floating-point values from the second source operand to the two, four or eight packed double precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double precision floating-point values from the second source to the two, four or eight packed double precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

### Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 9E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 AE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 BE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMsr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
  DEST[n+63:n] := RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN  
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256) 
  DEST[MAXVL-1:256] := 0
FI

VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
  DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN  
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256) 
  DEST[MAXVL-1:256] := 0
FI

VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 64*i;
  DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN  
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256) 
  DEST[MAXVL-1:256] := 0
FI
VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMSUBxxxPD __m512d __m512d_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __m512d_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __m512d_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __m512d_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __m512d_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d __m512d_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __m512d_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __m512d_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d __m256d_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __m256d_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __m256d_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d __m128d_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __m128d_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __m128d_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d __m128d_fnmsub_pd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d __m256d_fnmsub_pd(__m256d a, __m256d b, __m256d c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 9E /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VFNMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 AE /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VFNMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 BE /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VFNMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 9E /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VFNMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 AE /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VFNMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.0 BE /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>VFNMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 9E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VFNMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 AE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>VFNMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 BE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VFNMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 9E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VFNMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 AE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>VFNMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.0 BE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>VFNMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 9E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>VFNMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 AE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>VFNMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.0 BE /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.</td>
</tr>
<tr>
<td>VFNMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Description**

VFNMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

MAXNUM := 2
ELSEIF (VEX.256)

MAXNUM := 4
FI

For i = 0 to MAXNUM-1 {

n := 32*i;

DEST[n+31:n] := RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}

IF (VEX.128) THEN

DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)

DEST[MAXVL-1:256] := 0
FI
VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
MAXNUM := 2
ELSEIF (VEX.256)
MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
DEST[MAXVL-1:256] := 0
FI

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
MAXNUM := 2
ELSEIF (VEX.256)
MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
DEST[MAXVL-1:256] := 0
FI

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] :=
    RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
          ELSE
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
            FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] := 0
          FI
        FI;
  ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
    RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI;
  ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(VL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
                    RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
                    ELSE
                        DEST[i+31:i] :=
                        RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(VL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0
VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
      THEN
        DEST[i+31:i] :=
        RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
      ELSE
        DEST[i+31:i] :=
        RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxPS __m512 __m512 __m512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __m512 __m512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __m512 __m512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __m512 __m512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __m512 __m512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 __m512 __m512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __m512 __m512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m256 __m256 __m256_mask_fnmsub_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __m256 __m256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __m256 __m256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 __m128 __m128_mask_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 __m128 __m128_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __m128 __m128_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m256 __m256 __m256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W1 9F /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 AF /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 BF /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 9F /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 AF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 BF /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

VFNMSUB132SD: Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, ”*“ and ”-“ symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

FI;

IF k1[0] or *no writemask*

THEN  

```
DEST[63:0] := RoundFPControl(-(DEST[63:0] * SRC3[63:0]) - SRC2[63:0])
```

ELSE

```
IF *merging-masking*

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[63:0] := 0
```

FI;

FI;

```
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

**VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

FI;

IF k1[0] or *no writemask*

THEN  

```
DEST[63:0] := RoundFPControl(-(SRC2[63:0] * DEST[63:0]) - SRC3[63:0])
```

ELSE

```
IF *merging-masking*

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[63:0] := 0
```

FI;

FI;

```
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```
VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI;

IF k1[0] or *no writemask*
   THEN
      DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
   ELSE
     IF *merging-masking* ; merging-masking
       THEN *DEST[63:0] remains unchanged*
     ELSE ; zeroing-masking
       THEN DEST[63:0] := 0
     FI;
   FI;

DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

DEST[63:0] := RoundFPControl_MXCSR(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

DEST[63:0] := RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.w0 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.w0 AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.w0 BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.w0 9F /r VFNMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.w0 AF /r VFNMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.w0 BF /r VFNMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations
involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

IF k1[0] or *no writemask*
THEN
   DEST[31:0] := RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
ELSE
   IF *merging-masking* ; merging-masking
   THEN *DEST[31:0] remains unchanged*
   ELSE ; zeroing-masking
   THEN DEST[31:0] := 0
   FI;
FI;

DEST[MAXVL-1:128] := 0

**VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
   SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

IF k1[0] or *no writemask*
THEN
   DEST[31:0] := RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
ELSE
   IF *merging-masking* ; merging-masking
   THEN *DEST[31:0] remains unchanged*
   ELSE ; zeroing-masking
   THEN DEST[31:0] := 0
   FI;
FI;

DEST[MAXVL-1:128] := 0
VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] := RoundFPControl(- (SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] := 0
        FI;
    FI;
DEST[MAXVL-1:128] := 0

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[MAXVL-1:128] := 0

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[MAXVL-1:128] := 0

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VFPCLASSPD—Tests Types of Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 / r ib VFPCLASSPD k2 [k1], xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 / r ib VFPCLASSPD k2 [k1], ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 / r ib VFPCLASSPD k2 [k1], zmm2/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The FPCODE instruction checks the packed double precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-14.

![Figure 5-13. Imm8 Byte Specifier of Special Case Floating-Point Values for VFPCLASSPD/SD/PS/SS](image)

Table 5-14. Classifier Operations for VFPCLASSPD/SD/PS/SS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>QNaN</td>
<td>PosZero</td>
<td>NegZero</td>
<td>PosINF</td>
<td>NegINF</td>
<td>Denormal</td>
<td>Negative</td>
<td>SNaN</td>
</tr>
<tr>
<td>Classifier</td>
<td>Checks for QNaN</td>
<td>Checks for +0</td>
<td>Checks for -0</td>
<td>Checks for INF</td>
<td>Checks for -INF</td>
<td>Checks for Denormal</td>
<td>Checks for Negative finite</td>
<td>Checks for SNaN</td>
</tr>
</tbody>
</table>

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.
EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

CheckFPClassDP \{(tsrc[63:0], imm8[7:0])\}

```plaintext
//* Start checking the source operand for special type */
NegNum := tsrc[63];
IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
  MantAllZeros := 1;
ELSIF (tsrc[51:0]=0h) Then
  MantAllZeros := 1;
FI;
ZeroNumber := ExpAllZeros AND MantAllZeros
SignalingBit := tsrc[51];
sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); //sNaN
qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; //qNaN
Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; //+0
Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; //0
Pinf_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; //+Inf
Ninf_res := NegNum AND ExpAllZeros AND MantAllZeros; //-Inf
Denorm_res := ExpAllZeros AND NOT(MantAllZeros); //denorm
FinNeg_res := NegNum AND NOT(ExpAllZeros) AND NOT(ZeroNumber); //--finite

bResult = (imm8[0] AND qNaN_res) OR (imm8[1] AND Pzero_res) OR

Return bResult;
} //* end of CheckFPClassDP() */
```

**VFCLASSPD (EVEX Encoded versions)**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)
  \(i := j * 64\)
  IF \(k1[j]\) OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC *is memory*)
        THEN
          DEST[j] := CheckFPClassDP(SRC1[63:0], imm8[7:0]);
        ELSE
          DEST[j] := CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
        FI;
      ELSE DEST[j] := 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFPCLASSPD __mmask8 __mm512_fpclass_pd_mask(__m512d a, int c);
VFPCLASSPD __mmask8 __mm512_mask_fpclass_pd_mask(__mmask8 m, __m512d a, int c);
VFPCLASSPD __mmask8 __mm256_fpclass_pd_mask(__m256d a, int c);
VFPCLASSPD __mmask8 __mm256_mask_fpclass_pd_mask(__mmask8 m, __m256d a, int c);
VFPCLASSPD __mmask8 __mm128_fpclass_pd_mask(__m128d a, int c);
VFPCLASSPD __mmask8 __mm_mask_fpclass_pd_mask(__mmask8 m, __m128d a, int c);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.
VFPCLASSPH—Test Types of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{z2}, xmm1/m128/ m16bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.256.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{z2}, ymm1/m256/ m16bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.512.NP.0F3A.W0 66 /r /ib VFPCLASSPH k1{z2}, zmm1/m512/ m16bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction checks the packed FP16 values in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 5-9 for the categories. The classified results of all specified categories of an input value are OR-ed together to form the final boolean result for the input element. The result is written to the corresponding bits in the destination mask register according to the writemask.

Table 5-9. Classifier Operations for VFPCLASSPH/VFPCLASSSH

<table>
<thead>
<tr>
<th>Bits</th>
<th>Category</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm8[0]</td>
<td>QNAN</td>
<td>Checks for QNAN</td>
</tr>
<tr>
<td>imm8[1]</td>
<td>PosZero</td>
<td>Checks +0</td>
</tr>
<tr>
<td>imm8[2]</td>
<td>NegZero</td>
<td>Checks for -0</td>
</tr>
<tr>
<td>imm8[3]</td>
<td>PosINF</td>
<td>Checks for +∞</td>
</tr>
<tr>
<td>imm8[4]</td>
<td>NegINF</td>
<td>Checks for -∞</td>
</tr>
<tr>
<td>imm8[5]</td>
<td>Denormal</td>
<td>Checks for Denormal</td>
</tr>
<tr>
<td>imm8[6]</td>
<td>Negative</td>
<td>Checks for Negative finite</td>
</tr>
<tr>
<td>imm8[7]</td>
<td>SNAN</td>
<td>Checks for SNAN</td>
</tr>
</tbody>
</table>
**Operation**

def check_fp_class_fp16(tsrc, imm8):
    negative := tsrc[15]
    exponent_all_ones := (tsrc[14:10] == 0x1F)
    exponent_all_zeros := (tsrc[14:10] == 0)
    mantissa_all_zeros := (tsrc[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := tsrc[9]
    snan := exponent_all_ones and not(mantissa_all_zeros) and not(signaling_bit)
    qnan := exponent_all_ones and not(mantissa_all_zeros) and signaling_bit
    positive_zero := not(negative) and zero
    negative_zero := negative and zero
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    negative_infinity := negative and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)
    finite_negative := negative and not(exponent_all_ones) and not(zero)
    return (imm8[0] and qnan) OR
           (imm8[1] and positive_zero) OR
           (imm8[2] and negative_zero) OR
           (imm8[3] and positive_infinity) OR
           (imm8[4] and negative_infinity) OR
           (imm8[5] and denormal) OR
           (imm8[6] and finite_negative) OR
           (imm8[7] and snan)

VFPCLASSPH dest{k2}, src, imm8

VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k2[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := SRC.fp16[0]
        ELSE:
            tsrc := SRC.fp16[i]
        DEST.bit[i] := check_fp_class_fp16(tsrc, imm8)
    ELSE:
        DEST.bit[i] := 0

DEST[MAXKL-1:kl] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFPCLASSPH __mmask8 __mm_fpclass_ph_mask (__m128h a, int imm8);
VFPCLASSPH __mmask8 __mm_mask_fpclass_ph_mask (__mmask8 k1, __m128h a, int imm8);
VFPCLASSPH __mmask16 __mm256_fpclass_ph_mask (__m256h a, int imm8);
VFPCLASSPH __mmask16 __mm256_mask_fpclass_ph_mask (__mmask16 k1, __m256h a, int imm8);
VFPCLASSPH __mmask32 __mm512_fpclass_ph_mask (__m512h a, int imm8);
VFPCLASSPH __mmask32 __mm512_mask_fpclass_ph_mask (__mmask32 k1, __m512h a, int imm8);

**SIMD Floating-Point Exceptions**

None.

VFPCLASSPH—Test Types of Packed FP16 Values

Vol. 2C  5-333
Other Exceptions
EVEX-encoded instructions, see Table 2-49, "Type E4 Class Exception Conditions."
VFPCLASSPS—Tests Types of Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
</tbody>
</table>

Description

The FPCLASSPS instruction checks the packed single-precision floating-point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-14.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0])

```c
// * Start checking the source operand for special type */
NegNum := tsrc[31];
IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; Fi;
IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then MantAllZeros := 1;
ELSIF (tsrc[22:0]=0h) Then MantAllZeros := 1;
FI;
ZeroNumber := ExpAllZeros AND MantAllZeros
SignalingBit := tsrc[22];

sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```
INSTRUCTION SET REFERENCE, V

VFPCLASSPS—Tests Types of Packed Float32 Values

Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res := ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = (imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

Return bResult;
} */ end of CheckSPClassSP() */

VFPCCLASSPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b == 1) AND (SRC *is memory*)
            THEN
                DEST[j] := CheckFPClassDP(SRC1[31:0], imm8[7:0]);
            ELSE
                DEST[j] := CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
            FI;
        ELSE  DEST[j] := 0;  zeroing-masking only
        FI;
ENDIF
DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFCLASSPS __m128 __mmask16 _mm512_fpclass_ps_mask(__m512 a, int c);
VFCLASSPS __m16 __mmask16 _mm512_mask_fpclass_ps_mask(__mmask16 m, __m512 a, int c)
VFCLASSPS __m128 __mmask8 _mm256_fpclass_ps_mask(__m256 a, int c)
VFCLASSPS __m128 __mmask8 _mm256_mask_fpclass_ps_mask(__mmask8 m, __m256 a, int c)
VFCLASSPS __m256 __mmask8 __mm128_fpclass_ps_mask(__mmask8 m, __m128 a, int c)
VFCLASSPS __m256 __mmask8 __mm128_mask_fpclass_ps_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, “Type E4 Class Exception Conditions.”
Additionally:
#UD  If EVEX.vvvv != 1111B.
VFPCLASSSD—Tests Type of a Scalar Float64 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W1 67 /r ib</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The FPCLASSSD instruction checks the low double precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-14.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

  NegNum := tsrc[63];
  IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
  IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
  IF (ExpAllZeros AND MXCSR.DAZ) Then
    MantAllZeros := 1;
  ELSIF (tsrc[51:0]=0h) Then
    MantAllZeros := 1;
  FI;
  ZeroNumber := ExpAllZeros AND MantAllZeros
  SignalingBit := tsrc[51];

  sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
  qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
  Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
  Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
  Pinf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
  Ninf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
  Denorm_res := ExpAllZeros AND NOT(MantAllZeros); // denorm
  FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

  bResult = (imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

  Return bResult;
} //*/ end of CheckFPClassDP() */

VFPCLASSSD—Tests Type of a Scalar Float64 Value
VFPCLASSSD (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
        CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE DEST[0] := 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFPCLASSSD __mmask8 _mm_fpclass_sd_mask(__m128d a, int c)
VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-53, “Type E6 Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B.
**VFCLASSSH—Test Types of Scalar FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP.0F3A.W0 67 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction checks the low FP16 value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against; see Table 5-9 for the categories. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in the destination mask register according to the writemask. The other bits in the destination mask register are zeroed.

**Operation**

**VFCLASSSH dest[k2], src, imm8**

IF k2[0] or *no writemask*:

```
DEST.bit[0] := check_fp_class_fp16(src.fp16[0], imm8) // see VFCLASSPH
```

ELSE:

```
DEST.bit[0] := 0
```

```
DEST[MAXKL-1:1] := 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VFCLASSSH __mmask8 _mm_fpclass_sh_mask (__m128h a, int imm8);
VFCLASSSH __mmask8 __mm_mask_fpclass_sh_mask (__mmask8 k1, __m128h a, int imm8);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”
VFPCLASSSS—Tests Type of a Scalar Float32 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W0 67</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The FPCLASSSS instruction checks the low single-precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1:1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-14.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0])

```c
// Start checking the source operand for special type */
NegNum := tsrc[31];
IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
  MantAllZeros := 1;
ELSIF (tsrc[22:0]=0h) Then
  MantAllZeros := 1;
FI;
ZeroNumber= ExpAllZeros AND MantAllZeros
SignalingBit= tsrc[22];

sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
Pinf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
Ninf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res := ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND sNaN_res ) OR (imm8[1] AND Pzero_res ) OR
Return bResult;
```
VFCLASSSS (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
        CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE DEST[0] := 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFCLASSSS __mmask8 __mm_fpclass_ss_mask(__m128 a, int c)
VFCLASSSS __mmask8 __mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-53, "Type E6 Class Exception Conditions."
Additionally:
#UD If EVEX.vvvv != 1111B.
**VGATHERDPD/VGATHERQPD—Gather Packed Double Precision Floating-Point Values Using Signed Dword/Qword Indices**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 92 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather double precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 93 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather double precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 92 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather double precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 93 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather double precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The instruction conditionally loads up to 2 or 4 double precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.
VEX.256 version: The instruction will gather four double precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VGATHERDPD (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFF_FFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  k := j * 32;
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
VGATHERQPD (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
  FI;
  MASK[i +63: i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;

VGATHERQPD (VEX.256 version)
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63: i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;

VGATHERDPD (VEX.256 version)
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  k := j * 32;
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
    MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);
VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);
VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);
VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);
VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);
VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);
VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);
VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-27, ”Type 12 Class Exception Conditions.”
**VGATHERDPS/VGATHERQPS—Gather Packed Single Precision Floating-Point Values Using Signed Dword/Qword Indices**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 92 /r VGATHERDPS xmm1, vm32x, xmm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather single-precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64x, xmm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather single-precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 92 /r VGATHERDPS ymm1, vm32y, ymm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather single-precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64y, xmm2</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather single-precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:ind</td>
<td>VEX.vvvv (r, w)</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.
VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• This instruction will cause a #UD if the address size attribute is 16-bit.
• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation
DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VGATHERDPS (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 3
i := j * 32;
IF MASK[31+i] THEN
MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
ELSE
MASK[i +31:i] := 0;
FI;
ENDFOR
FOR j := 0 to 3
i := j * 32;
DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
IF MASK[31+i] THEN
DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +31:i] := 0;
ENDFOR

VGATHERDPS/VGATHERQPS—Gather Packed Single Precision Floating-Point Values Using Signed Dword/Qword Indices
DEST[MAXVL-1:128] := 0;

**VGATHERQPS (VEX.128 version)**

MASK[MAXVL-1:64] := 0;
FOR j := 0 to 3
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR

FOR j := 0 to 1
  k := j * 64;
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR

DEST[MAXVL-1:64] := 0;

**VGATHERDPS (VEX.256 version)**

MASK[MAXVL-1:256] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR

FOR j := 0 to 7
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR

DEST[MAXVL-1:256] := 0;

**VGATHERQPS (VEX.256 version)**

MASK[MAXVL-1:128] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR

FOR j := 0 to 3
  i := j * 32;
  IF MASK[i+32] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
\[
k := j \times 64;
\]
\[
i := j \times 32;
\]
\[
\text{DATA_ADDR} := \text{BASE_ADDR} + (\text{SignExtend(VINDEX1[k+63:k])} \times \text{SCALE}) + \text{DISP};
\]
\[
\text{IF MASK[31+i]} \text{ THEN}
\]
\[
\text{DEST[i+31:i]} := \text{FETCH_32BITS(DATA_ADDR)}; // a fault exits the instruction
\]
\[
\text{FI};
\]
\[
\text{MASK[i+31:i]} := 0;
\]
\[
\text{ENDIF}
\]
\[
\text{DEST[MAXVL-1:128]} := 0;
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\text{VGATHERDPS: } \_\text{m}128\_\text{mm}_i32\text{gather}_\text{ps} (\text{float const } \ast \text{base}, \_\text{m}128\text{i index, const int scale});
\]
\[
\text{VGATHERDPS: } \_\text{m}128\_\text{mm}_\text{mask}_i32\text{gather}_\text{ps} (\_\text{m}128 \text{ src, float const } \ast \text{base, } \_\text{m}128\text{i index, } \_\text{m}128 \text{ mask, const int scale});
\]
\[
\text{VGATHERDPS: } \_\text{m}256\_\text{mm}_256_i32\text{gather}_\text{ps} (\text{float const } \ast \text{base, } \_\text{m}256\text{i index, const int scale});
\]
\[
\text{VGATHERDPS: } \_\text{m}256\_\text{mm}_256\_\text{mask}_i32\text{gather}_\text{ps} (\_\text{m}256 \text{ src, float const } \ast \text{base, } \_\text{m}256\text{i index, } \_\text{m}256 \text{ mask, const int scale});
\]
\[
\text{VGATHERQPS: } \_\text{m}128\_\text{mm}_i64\text{gather}_\text{ps} (\text{float const } \ast \text{base, } \_\text{m}128\text{i index, const int scale});
\]
\[
\text{VGATHERQPS: } \_\text{m}128\_\text{mm}_\text{mask}_i64\text{gather}_\text{ps} (\_\text{m}128 \text{ src, float const } \ast \text{base, } \_\text{m}128\text{i index, } \_\text{m}128 \text{ mask, const int scale});
\]
\[
\text{VGATHERQPS: } \_\text{m}256\_\text{mm}_256_i64\text{gather}_\text{ps} (\text{float const } \ast \text{base, } \_\text{m}256\text{i index, const int scale});
\]
\[
\text{VGATHERQPS: } \_\text{m}256\_\text{mm}_256\_\text{mask}_i64\text{gather}_\text{ps} (\_\text{m}256 \text{ src, float const } \ast \text{base, } \_\text{m}256\text{i index, } \_\text{m}256 \text{ mask, const int scale});
\]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-27, "Type 12 Class Exception Conditions."
VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword Indices

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R): VSIBbase, VectorReg(R): VSIBindex</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

A set of single-precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

### Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this
instruction is repeatable - given the same input values and architectural state, the same set of elements to the
left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if
ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.
The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit
mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are
ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction
will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j]
    THEN DEST[i+31:i] :=
        MEM[BASE_ADDR +
            SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
        k1[j] := 0
    ELSE *DEST[i+31:i] := remains unchanged*
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0

VGATHERDPD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j]
    THEN DEST[i+63:i] := MEM[BASE_ADDR +
        SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
        k1[j] := 0
    ELSE *DEST[i+63:i] := remains unchanged*
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD __m512d __mm512_i32gather_pd( __m256i vdx, void * base, int scale);
VGATHERDPD __m512d __mm512_mask_i32gather_pd( __m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d __mm256_mmask_i32gather_pd( __m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d __mm128_mmask_i32gather_pd( __m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_i32gather_ps( __m512i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_mask_i32gather_ps( __m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 __mm256_mmask_i32gather_ps( __m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPS __m128 __mm128_mmask_i32gather_ps( __m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, "Type E12 Class Exception Conditions."

INSTRUCTION SET REFERENCE, V

5-352  Vol. 2C  VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword Indices
VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

### Description

A set of 8 single-precision/double precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable—given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

This instruction does not perform AC checks, and so will never deliver an AC fault.

Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VGATHERQPS (EVEX encoded version)

(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

FOR \(j := 0 \) TO \(KL-1\)

\[i := j \times 32\]
\[k := j \times 64\]

IF \(k1[j] \) OR *no writemask*

THEN \(DEST[i+31:i] := MEM[BASE_ADDR + (VINDEX[k+63:k]) \times SCALE + DISP]\)

\(k1[j] := 0\)

ELSE *DEST[i+31:i] := remains unchanged*

FI;

ENDFOR

\(k1[MAX_KL-1:KL] := 0\)

\(DEST[MAXVL-1:VL/2] := 0\)

VGATHERQPD (EVEX encoded version)

(\(KL, VL\) = (2, 128), (4, 256), (8, 512))

FOR \(j := 0 \) TO \(KL-1\)

\[i := j \times 64\]

IF \(k1[j] \) OR *no writemask*

THEN \(DEST[i+63:i] := MEM[BASE_ADDR + (VINDEX[i+63:i]) \times SCALE + DISP]\)

\(k1[j] := 0\)

ELSE *DEST[i+63:i] := remains unchanged*

FI;

ENDFOR

\(k1[MAX_KL-1:KL] := 0\)

\(DEST[MAXVL-1:VL] := 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERQPD __m512d __mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d __mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d __mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d __mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 __mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 __mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, "Type E12 Class Exception Conditions."
VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMcr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponents from the normalized double precision floating-point representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double precision floating-point value and written to the corresponding qword elements of the destination operand (the first operand) as double precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

$\text{GETEXP}(x) = \lfloor \log_2(|x|) \rfloor$

Notation $\lfloor x \rfloor$ stands for the greatest integer not exceeding real number $x$.

### Table 5-15. VGETEXPPD/SD Special Cases

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>QNaN(src1)</td>
<td>If (SRC = SNaN) then #IE</td>
</tr>
<tr>
<td>0 &lt;</td>
<td>src1</td>
<td>&lt; INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= +INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= 0</td>
</tr>
</tbody>
</table>
Operation

NormalizeExpTinyDPFP(SRC[63:0])
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[51]; // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1; // One bit shift left
        Dst.exp--; // Decrement the exponent
    }
    Dst.fraction := 0; // zero out fraction bits
    Dst.sign := 1; // Return negative sign
    TMP[63:0] := MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    Return (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign := 0; // Zero out sign bit
    Src.exp := SRC[62:52];
    Src.fraction := SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);
    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[63:0] := NormalizeExpTinyDPFP(SRC[63:0]) ; // Get Normalized Exponent
            Set #DE
        }
        ELSE // exponent value is correct
        {
            TMP[63:0] := (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
        }
        TMP := SAR(TMP, 52); // Shift Arithmetic Right
        TMP := TMP – 1023; // Subtract Bias
        Return CvtI2D(TMP); // Convert INT to double precision floating-point number
    }
}
VGETEXPPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN
          DEST[i+63:i] := ConvertExpDPFP(SRC[63:0])
        ELSE
          DEST[i+63:i] := ConvertExpDPFP(SRC[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
See Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B.
VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values

VGETEXPPH xmm1{k1}{z}, xmm2/m128/m16bcst
VGETEXPPH ymm1{k1}{z}, ymm2/m256/m16bcst
VGETEXPPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}

Opcode/Description
EVEX.128.66.MAP6.W0 42 /r
VGETEXPPH xmm1{k1}{z}, xmm2/m128/m16bcst
Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.

EVEX.256.66.MAP6.W0 42 /r
VGETEXPPH ymm1{k1}{z}, ymm2/m256/m16bcst
Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.

EVEX.512.66.MAP6.W0 42 /r
VGETEXPPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}
Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM(reg) (w)</td>
<td>ModRM(r/m) (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction extracts the biased exponent from the normalized FP16 representation of each word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to an FP16 value and written to the corresponding word elements of the destination operand (the first operand) as FP16 numbers.

The destination elements are updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPH). Thus, the VGETEXPPH instruction does not require software to handle SIMD floating-point exceptions.

Table 5-16. VGETEXPH/VGETEXPSh Special Cases

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>QNaN(src1)</td>
<td>If (SRC = SNaN), then #IE.</td>
</tr>
<tr>
<td>0 &lt;</td>
<td>src1</td>
<td>&lt; INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= +INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= 0</td>
</tr>
</tbody>
</table>
Operation

def normalize_exponent_tiny_fp16(src):
    jbit := 0
    // src & dst are FP16 numbers with sign(1b), exp(5b) and fraction (10b) fields
    dst.exp := 1  // write bits 14:10
    dst.fraction := src.fraction // copy bits 9:0
    while jbit == 0:
        jbit := dst.fraction[9] // msb of the fraction
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    dst.fraction := 0
    return dst

def getexp_fp16(src):
    src.sign := 0 // make positive
    exponent_all_ones := (src[14:10] == 0x1F)
    exponent_all_zeros := (src[14:10] == 0)
    mantissa_all_zeros := (src[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := src[9]
    nan := exponent_all_ones and not(mantissa_all_zeros)
    snan := nan and not(signaling_bit)
    qnan := nan and signaling_bit
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)

    if nan:
        if snan:
            MXCSR.IE := 1
            return qnan(src) // convert snan to a qnan
        if positive_infinity:
            return src
        if zero:
            return -INF
        if denormal:
            tmp := normalize_exponent_tiny_fp16(src)
            MXCSR.DE := 1
        else:
            tmp := src
            tmp := SAR(tmp, 10) // shift arithmetic right
            tmp := tmp - 15 // subtract bias
            return convert_integer_to_fp16(tmp)
VGETEXPPH dest\(k1\), src
VL = 128, 256 or 512
KL := VL/16

FOR \(i := 0\) to KL-1:
  IF \(k1[i]\) or *no writemask*:
    IF SRC is memory and (EVEX.b = 1):
      \(tsc := src.fp16[0]\)
    ELSE:
      \(tsc := src.fp16[i]\)
    \(DEST.fp16[i] := \text{getexp}_\text{fp16}(tsrc)\)
  ELSE IF *zeroing*:
    \(DEST.fp16[i] := 0\)
  //else \(DEST.fp16[i]\) remains unchanged

\(DEST[\text{MAXVL}-1:VL] := 0\)

\textbf{Intel C/C++ Compiler Intrinsic Equivalent}
\vspace{0.1cm}
\begin{itemize}
  \item VGETEXPPH _m128h _mm_getexp_ph (_m128h a);
  \item VGETEXPPH _m128h _mm_mask_getexp_ph (_m128h src, __mmask8 k, __m128h a);
  \item VGETEXPPH _m128h _mm_maskz_getexp_ph (__mmask8 k, __m128h a);
  \item VGETEXPPH _m256h _mm256_getexp_ph (_m256h a);
  \item VGETEXPPH _m256h _mm256_mask_getexp_ph (_m256h src, __mmask16 k, __m256h a);
  \item VGETEXPPH _m256h _mm256_maskz_getexp_ph (__mmask16 k, __m256h a);
  \item VGETEXPPH _m512h _mm512_getexp_ph (_m512h a);
  \item VGETEXPPH _m512h _mm512_mask_getexp_ph (_m512h src, __mmask32 k, __m512h a);
  \item VGETEXPPH _m512h _mm512_maskz_getexp_ph (__mmask32 k, __m512h a);
  \item VGETEXPPH _m512h _mm512_mask_getexp_round_ph (_mm512h a, const int sae);
  \item VGETEXPPH _m512h _mm512_maskz_getexp_round_ph (__mmask32 k, __m512h a, const int sae);
\end{itemize}

\textbf{SIMD Floating-Point Exceptions}
Invalid, Denormal.

\textbf{Other Exceptions}
EVEX-encoded instructions, see Table 2-46, ”Type E2 Class Exception Conditions.”
VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values

### Description
Extracts the biased exponents from the normalized single-precision floating-point representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision floating-point value and written to the corresponding dword elements of the destination operand (the first operand) as single-precision floating-point numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-17.

The formula is:

\[ \text{GETEXP}(x) = \lfloor \log_2(|x|) \rfloor \]

Notation \( \lfloor x \rfloor \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Special Cases

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>QNaN(src1)</td>
<td>If (SRC = SNaN) then #IE</td>
</tr>
<tr>
<td>0 &lt;</td>
<td>src1</td>
<td>&lt; INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= +INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= 0</td>
</tr>
</tbody>
</table>
Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

```
Figure 5-14. VGETEXPPS Functionality On Normal Input values
```

### Operation

**NormalizeExpTinySPFP**(SRC[31:0])

```c
{  
  // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.  
  Src.Jbit := 0;  
  Dst.exp := 1;  
  Dst.fraction := SRC[22:0];  
  WHILE(Src.Jbit = 0)  
  {  
    Src.Jbit := Dst.fraction[22]; // Get the fraction MSB  
    Dst.fraction := Dst.fraction << 1; // One bit shift left  
    Dst.exp-- ; // Decrement the exponent  
  }  
  Dst.fraction := 0; // zero out fraction bits  
  Dst.sign := 1; // Return negative sign  
  TMP[31:0] := MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);  
  Return (TMP[31:0]);  
}
```

**ConvertExpSPFP**(SRC[31:0])

```c
{  
  Src.sign := 0; // Zero out sign bit  
  Src.exp := SRC[30:23];  
  Src.fraction := SRC[22:0];  
  // Check for NaN  
  IF (SRC = NaN)  
  {  
    IF ( SRC = SNAN ) SET IE;  
    Return QNAN(SRC);  
  }  
  // Check for +INF  
  IF (Src = +INF) RETURN (Src);  
  // check if zero operand  
  IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);  
  ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)  
  {  
```
IF (Src.exp = 0) AND (Src.fraction != 0) 
{ 
    TMP[31:0] := NormalizeExpTinySPFP(SRC[31:0]) ; // Get Normalized Exponent 
    Set #DE 
} 
ELSE // exponent value is correct 
{ 
    TMP[31:0] := (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction) ; 
} 
TMP := SAR(TMP, 23); // Shift Arithmetic Right 
TMP := TMP - 127; // Subtract Bias 
Return CvtI2S(TMP); // Convert INT to single precision floating-point number 
}

VGETEXPPS (EVEX encoded versions) 
(KL, VL) = (4, 128), (8, 256), (16, 512) 
FOR j := 0 TO KL-1 
i := j * 32 
IF k1[j] OR *no writemask* 
THEN 
IF (EVEX.b = 1) AND (SRC *is memory*) 
THEN 
    DEST[i+31:i] := 
    ConvertExpSPFP(SRC[31:0]) 
ELSE 
    DEST[i+31:i] := 
    ConvertExpSPFP(SRC[i+31:i]) 
Fi; 
ELSE 
IF *merging-masking* ; merging-masking 
THEN *DEST[i+31:i] remains unchanged* 
ELSE ; zeroing-masking 
    DEST[i+31:i] := 0 
Fi 
Fi; 
ENDFOR 
DEST[MAXVL-1:VL] := 0 

Intel C/C++ Compiler Intrinsic Equivalent 
VGETEXPPS __m512 _mm512_getexp_ps(__m512 a); 
VGETEXPPS __m512 __mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a); 
VGETEXPPS __m512 __mm512_maskz_getexp_ps( __mmask16 k, __m512 a); 
VGETEXPPS __m512 __mm512_getexp_round_ps( __m512 a, int sae); 
VGETEXPPS __m512 __mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae); 
VGETEXPPS __m512 __mm512_maskz_getexp_round_ps( __mmask16 k, __m512 a, int sae); 
VGETEXPPS __m256 __mm256_getexp_ps(__m256 a); 
VGETEXPPS __m256 __mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a); 
VGETEXPPS __m256 __mm256_maskz_getexp_ps( __mmask8 k, __m256 a); 
VGETEXPPS __m128 __mm128_getexp_ps(__m128 a); 
VGETEXPPS __m128 __mm128_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a); 
VGETEXPPS __m128 __mm128_maskz_getexp_ps( __mmask8 k, __m128 a);
SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
See Table 2-46, “Type E2 Class Exception Conditions.”
Additionally:
#UD If EVEX.vvvv != 1111B.
VGETEXPSD—Convert Exponents of Scalar Double Precision Floating-Point Value to Double Precision Floating-Point Value

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W1 43 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the biased exponent (bits 62:52) of the low double precision floating-point value in xmm3/m64 to a double precision floating-point value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Extracts the biased exponent from the normalized double precision floating-point representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double precision floating-point value and written to the destination operand (the first operand) as double precision floating-point numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

\[
\text{GETEXP}(x) = \text{floor}(\log_2(|x|))
\]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

### Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

**VGETEXPSD (EVEX encoded version)**

If \( \text{k1}[0] \) OR *no writemask*

THEN \( \text{DEST}[63:0] := \) ConvertExpDPFP(SRC2[63:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged* 

ELSE ; zeroing-masking

DEST[63:0] := 0

FI

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSD __m128d _mm_getexp_sd( __m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd( __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 43 /r VGETEXPSH xmm1{k1}{z}, xmm2, xmm3/m16 {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Convert the exponent of FP16 values in the low word of the source operand to FP16 results representing unbiased integer exponents, and stores the results in the low word of the destination register subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction extracts the biased exponents from the normalized FP16 representation of the low word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to an unbiased negative integer value. The integer value of the unbiased exponent is converted to an FP16 value and written to the low word element of the destination operand (the first operand) as an FP16 number.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-16.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTSH). Thus, the VGETEXPSH instruction does not require software to handle SIMD floating-point exceptions.

**Operation**

VGETEXPSH dest{k1}, src1, src2

IF k1[0] or *no writemask*:

\[ \text{DEST.fp16[0]} := \text{getexp}_\text{fp16}(\text{src2.fp16[0]}) \] // see VGETEXPPH

ELSE IF *zeroing*:

\[ \text{DEST.fp16[0]} := 0 \]

//else DEST.fp16[0] remains unchanged

\[ \text{DEST[127:16]} := \text{src1[127:16]} \]

\[ \text{DEST[MAXVL-1:128]} := 0 \]

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPSH _m128h _mm_getexp_round_sh (__m128h a, __m128h b, const int sae);
VGETEXPSH _m128h _mm_mask_getexp_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH _m128h _mm_maskz_getexp_round_sh (__mmask8 k, __m128h a, __m128h b, const int sae);
VGETEXPSH _m128h _mm_getexp_sh (__m128h a, __m128h b);
VGETEXPSH _m128h _mm_mask_getexp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VGETEXPSH _m128h _mm_maskz_getexp_sh (__mmask8 k, __m128h a, __m128h b);
**SIMD Floating-Point Exceptions**
Invalid, Denormal

**Other Exceptions**
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 43 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a single-precision floating-point value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponent from the normalized single-precision floating-point representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision floating-point value and written to the destination operand (the first operand) as single-precision floating-point numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-17.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD floating-point exceptions.

Operation

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS
// ConvertExpDPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

VGETEXPSS (EVEX encoded version)

IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
        ConvertExpDPFP(SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions.”
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas From Float64 Vector

### Description

Convert double precision floating values in the source operand (the second operand) to double precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

For each input double precision floating-point value x, The conversion operation is:

\[ \text{GetMant}(x) = \pm 2^k |x.\text{significand}| \]

where:

\[ 1 \leq |x.\text{significand}| < 2 \]

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc.
and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted double precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b; otherwise instructions will #UD.

### Table 5-18. GetMant() Special Float Values Behavior

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
<th>Exceptions / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(SRC)</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>+∞</td>
<td>1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>+0</td>
<td>1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>-0</td>
<td>IF (SC[0]) THEN +1.0 ELSE -1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>-∞</td>
<td>IF (SC[1]) THEN QNaN_Indefinite ELSE (if (SC[0]) THEN +1.0 ELSE -1.0)</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>negative</td>
<td>SC[1] ? QNaN_Indefinite : Getmant(SRC)¹</td>
<td>If (SC[1]) then #IE</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

**Operation**

```python
def getmant_fp64(src, sign_control, normalization_interval):
    bias := 1023
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0x7FF) and (dst.fraction = 0)
    nan := (dst.exp = 0x7FF) and (dst.fraction != 0)
    src_signaling := src.fraction[51]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        MXCSR.IE := 1
        return qnan(src)
    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
```
if infinity:
    if sign_control[1]:
        MXCSR.IE := 1
        return QNaN_Indefinite
    return signed_one
if denormal:
    jbit := 0
    dst.exp := bias
    while jbit = 0:
        jbit := dst.fraction[51]
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    MXCSR.DE := 1

unbiased_exp := dst.exp - bias
odd_exp := unbiased_exp[0]
signaling_bit := dst.fraction[51]
if normalization_interval = 0b00:
    dst.exp := bias
else if normalization_interval = 0b01:
    dst.exp := odd_exp ? bias-1 : bias
else if normalization_interval = 0b10:
    dst.exp := bias-1
else if normalization_interval = 0b11:
    dst.exp := signaling_bit ? bias-1 : bias
return dst

VGETMANTPD (EVEX Encoded Versions)
VGETMANTPD dest{k1}, src, imm8
VL := 128, 256, or 512
KL := VL / 64
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or /*no writemask*/:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.double[0]
        ELSE:
            tsrc := src.double[i]
        DEST.double[i] := getmant_fp64(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.double[i] := 0
    //else DEST.double[i] remains unchanged

DEST[MAX_VL-1:VL-1] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTPD __m512d __mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d __mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d __mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d __mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d __mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d __mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d __mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm128_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm128_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d __mm128_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.
VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector

**Opcode/Instruction** | Op/En | 64/32 Bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
EVEX.128.NP.0F3A.W0 26 /r /ib VGETMANTPH xmm1{k1}{z}, xmm2/m128/m16bcst, imm8 | A | V/V | AVX512-FP16 AVX512VL | Get normalized mantissa from FP16 vector xmm2/m128/m16bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.

EVEX.256.NP.0F3A.W0 26 /r /ib VGETMANTPH ymm1{k1}{z}, ymm2/m256/m16bcst, imm8 | A | V/V | AVX512-FP16 AVX512VL | Get normalized mantissa from FP16 vector ymm2/m256/m16bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.

EVEX.512.NP.0F3A.W0 26 /r /ib VGETMANTPH zmm1{k1}{z}, zmm2/m512/m16bcst {sae}, imm8 | A | V/V | AVX512-FP16 | Get normalized mantissa from FP16 vector zmm2/m512/m16bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, subject to writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts the FP16 values in the source operand (the second operand) to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-19. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

The destination elements are updated according to the writemask.

### Table 5-19. imm8 Controls for VGETMANTPH/VGETMANTSH

<table>
<thead>
<tr>
<th>imm8 Bits</th>
<th>Definition</th>
</tr>
</thead>
</table>
| imm8[3:2] | Sign Control (SC)  
0b00: Sign(SRC)  
0b01: 0  
0b1x: QNaN_Indefinite if sign(SRC)!=0 |
| imm8[1:0] | Interv  
0b00: Interval is [1, 2)  
0b01: Interval is [1/2, 2)  
0b10: Interval is [1/2, 1)  
0b11: Interval is [3/4, 3/2) |

For each input FP16 value x, The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[1 \leq |x.\text{significand}| < 2\]
Unbiased exponent $k$ depends on the interval range defined by $\text{interv}$ and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of $\text{imm8}[1:0]$ and sign control are shown in Table 5-19.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent $k$ (adding bias) and a mantissa normalized to the range specified by $\text{interv}$.

The GetMant() function follows Table 5-20 when dealing with floating-point special numbers.

### Operation

```python
def getmant_fp16(src, sign_control, normalization_interval):
    bias := 15
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and (dst.fraction = 0)
    denormal := (dst.exp = 0) and (dst.fraction != 0)
    infinity := (dst.exp = 0x1F) and (dst.fraction = 0)
    nan := (dst.exp = 0x1F) and (dst.fraction != 0)
    src_signaling := src.fraction[9]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
            return qnan(src)
        if positive and (zero or infinity):
            return 1.0
        if negative:
            if zero:
                return signed_one
            if infinity:
                # Implementation
```
if sign_control[1]:
    MXCSR.IE := 1
return QNaN_Indefinite
return signed_one
if sign_control[1]:
    MXCSR.IE := 1
return QNaN_Indefinite
if denormal:
    jbit := 0
    dst.exp := bias // set exponent to bias value
while jbit = 0:
    jbit := dst.fraction[9]
    dst.fraction := dst.fraction << 1
    dst.exp := dst.exp - 1
    MXCSR.DE := 1
unbiased_exp := dst.exp - bias
odd_exp := unbiased_exp[0]
signaling_bit := dst.fraction[9]
if normalization_interval = 0b00:
    dst.exp := bias
else if normalization_interval = 0b01:
    dst.exp := odd_exp ? bias - 1 : bias
else if normalization_interval = 0b10:
    dst.exp := bias - 1
else if normalization_interval = 0b11:
    dst.exp := signaling_bit ? bias - 1 : bias
return dst

VGETMANTPH dest[k1], src, imm8
VL = 128, 256 or 512
KL := VL/16

sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := getmant_fp16(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPH __m128h _mm_getmant_ph (__m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_mask_getmant_ph (__m128h src, __mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_maskz_getmant_ph (__mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_getmant_ph (__m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_mask_getmant_ph (__m256h src, __mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_maskz_getmant_ph (__mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_mask_getmant_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_maskz_getmant_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_round_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_mask_getmant_round_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_maskz_getmant_round_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector

**Description**

Convert single-precision floating values in the source operand (the second operand) to single-precision floating-point values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input single-precision floating-point value $x$, The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k|x.\text{significand}|$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent $k$ can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted single-precision floating-point result is encoded according to the sign control, the unbiased exponent $k$ (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.
Operation

def getmant_fp32(src, sign_control, normalization_interval):
    bias := 127
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and (0 is dst.fraction) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0xFF) and (dst.fraction = 0)
    nan := (dst.exp = 0xFF) and (dst.fraction != 0)
    src_signaling := src.fraction[22]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)
    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
            return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite
    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[22]
            dst.fraction := dst.fraction << 1
            dst.exp := dst.exp - 1
        MXCSR.DE := 1
        unbiased_exp := dst.exp - bias
        odd_exp := unbiased_exp[0]
        signaling_bit := dst.fraction[22]
        if normalization_interval = 0b00:
            dst.exp := bias
        else if normalization_interval = 0b01:
            dst.exp := odd_exp ? bias-1 : bias
        else if normalization_interval = 0b10:
            dst.exp := bias-1
        else if normalization_interval = 0b11:
            dst.exp := signaling_bit ? bias-1 : bias

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector

INSTRUCTION SET REFERENCE, V
return dst

**VGETMANTPS (EVEX encoded versions)**

VGETMANTPS dest[k1], src, imm8

VL = 128, 256, or 512

KL := VL / 32

sign_control := imm8[3:2]

normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF SRC is memory and (EVEX.b = 1):
      tsrc := src.float[0]
    ELSE:
      tsrc := src.float[i]
    DEST.float[i] := getmant_fp32(tsrc, sign_control, normalization_interval)
  ELSE IF *zeroing*:
    DEST.float[i] := 0
  //else DEST.float[i] remains unchanged

DEST[MAX_VL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTPS __m512 _mm512_getmant_ps(__m512 a, enum intv, enum sgn);

VGETMANTPS __m512 __mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn);

VGETMANTPS __m512 __mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);

VGETMANTPS __m512 __mm512_getmant_round_ps(__m512 a, enum intv, enum sgn, int r);

VGETMANTPS __m512 __mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);

VGETMANTPS __m512 __mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);

VGETMANTPS __m256 __mm256_getmant_ps(__m256 a, enum intv, enum sgn);

VGETMANTPS __m256 __mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);

VGETMANTPS __m256 __mm256_maskz_getmant_ps(__mmask8 k, __m256 a, enum intv, enum sgn);

VGETMANTPS __m128 __mm128_getmant_ps(__m128 a, enum intv, enum sgn);

VGETMANTPS __m128 __mm128_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);

VGETMANTPS __m128 __mm128_maskz_getmant_ps(__mmask8 k, __m128 a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.
VGETMANTSD—Extract Float64 of Normalized Mantissa From Float64 Scalar

**Opcode/ Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sacl}, imm8</td>
<td>A V/V AVX512F</td>
<td>Extract the normalized mantissa of the low float64 element in xmm3/m64 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Operation
// getmant_fp64(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPD

VGETMANTSD (EVEX encoded version)
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
  THEN DEST[63:0] :=
    getmant_fp64(src, sign_control, normalization_interval)
ELSE
  IF *merging-masking*; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63:0] := 0
  FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions
Denormal, Invalid

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar

**Opcode/ Instruction** | **Op/ En** | **64/32 Bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVA.XLLG.NP.0F3A.W0 27 /r /ib | A | V/V | AVX512-FP16 | Extract the normalized mantissa of the low FP16 element in xmm3/m16 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 subject to writemask k1 and merge with the other elements of xmm2. Bits 127:16 of xmm2 are copied to xmm1[127:16].

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.Vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

This instruction converts the FP16 value in the low element of the second source operand to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-19. The converted result is written to the low element of the destination operand using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (SC) is specified by bits 3:2 of the immediate byte.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For each input FP16 value x, The conversion operation is:

\[ \text{GetMant}(x) = \pm 2^k |x.\text{significand}| \]

where:

\[ 1 \leq |x.\text{significand}| < 2 \]

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 5-19.

Each converted FP16 result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-20 when dealing with floating-point special numbers.

**Operation**

\[
\text{VGETMANTSH dest[k1], src1, src2, imm8} \\
\text{sign\_control} := \text{imm8}[3:2] \\
\text{normalization\_interval} := \text{imm8}[1:0]
\]

IF k1[0] or *no writemask*:

\[
\text{dest.fp16[0]} := \text{getmant\_fp16(src2.fp16[0]),} \\
\text{sign\_control,} \\
\text{normalization\_interval})
\]

ELSE IF *zeroing*:

\[
\text{dest.fp16[0]} := 0 \\
//else dest.fp16[0] remains unchanged
\]

\[
\text{DEST[127:16]} := \text{src1[127:16]} \\
\text{DEST[MAXVL-1:128]} := 0
\]
Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSH __m128h _mm_getmant_round_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_mask_getmant_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_maskz_getmant_round_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);

VGETMANTSH __m128h _mm_getmant_sh (__m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

VGETMANTSH __m128h _mm_mask_getmant_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

VGETMANTSH __m128h _mm_maskz_getmant_sh (__mmask8 k, __m128h a, __m128h b, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 [k1]{[z], xmm2, xmm3/m32{sa}, imm8}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to single-precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[
1 \leq |x.\text{significand}| < 2
\]

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted single-precision floating-point result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-18 when dealing with floating-point special numbers.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.
Operation

// getmant_fp32(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPS

VGETMANTSS (EVEX encoded version)

SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
        getmant_fp32(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            DEST[31:0] := 0
        FI
Fl;
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."
VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 18 / r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 18 / r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 18 / r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 18 / r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 18 / r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1A / r</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1A / r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed double precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/x/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/x/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/x/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/x/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granularity offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either a YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.
Operation

**VINSERTF32x4 (EVEX encoded versions)**

\((KL, VL) = (8, 256), (16, 512)\)

\(\text{TEMP\_DEST}[VL-1:0] := \text{SRC1}[VL-1:0]\)

IF \(VL = 256\)

CASE (imm8[0]) OF
  0: \(\text{TMP\_DEST}[127:0] := \text{SRC2}[127:0]\)
  1: \(\text{TMP\_DEST}[255:128] := \text{SRC2}[127:0]\)
ESAC.

FI;

IF \(VL = 512\)

CASE (imm8[1:0]) OF
  00: \(\text{TMP\_DEST}[127:0] := \text{SRC2}[127:0]\)
  01: \(\text{TMP\_DEST}[255:128] := \text{SRC2}[127:0]\)
  10: \(\text{TMP\_DEST}[383:256] := \text{SRC2}[127:0]\)
  11: \(\text{TMP\_DEST}[511:384] := \text{SRC2}[127:0]\)
ESAC.

FI;

FOR \(j := 0\) TO \(KL-1\)
  \(i := j \times 32\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(\text{DEST}[i+31:i] := \text{TMP\_DEST}[i+31:i]\)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST*[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      \(\text{DEST}[i+31:i] := 0\)
    FI
  FI;
ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

**VINSERTF64x2 (EVEX encoded versions)**

\((KL, VL) = (4, 256), (8, 512)\)

\(\text{TEMP\_DEST}[VL-1:0] := \text{SRC1}[VL-1:0]\)

IF \(VL = 256\)

CASE (imm8[0]) OF
  0: \(\text{TMP\_DEST}[127:0] := \text{SRC2}[127:0]\)
  1: \(\text{TMP\_DEST}[255:128] := \text{SRC2}[127:0]\)
ESAC.

FI;

IF \(VL = 512\)

CASE (imm8[1:0]) OF
  00: \(\text{TMP\_DEST}[127:0] := \text{SRC2}[127:0]\)
  01: \(\text{TMP\_DEST}[255:128] := \text{SRC2}[127:0]\)
  10: \(\text{TMP\_DEST}[383:256] := \text{SRC2}[127:0]\)
  11: \(\text{TMP\_DEST}[511:384] := \text{SRC2}[127:0]\)
ESAC.

FI;

FOR \(j := 0\) TO \(KL-1\)
  \(i := j \times 64\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(\text{DEST}[i+63:i] := \text{TMP\_DEST}[i+63:i]\)
  ELSE
IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
ELSE; zeroing-masking
    DEST[i+63:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VINSERTF32x8 (EVEX.U1.512 encoded version)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TEMP_DEST[255:0] := SRC2[255:0]
  1: TEMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
  i := j * 32
  IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
  ELSE
      IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
      ELSE; zeroing-masking
          DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VINSERTF64x4 (EVEX.512 encoded version)
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TEMP_DEST[255:0] := SRC2[255:0]
  1: TEMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
  i := j * 64
  IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
  ELSE
      IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
      ELSE; zeroing-masking
          DEST[i+63:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VINSERTF128 (VEX encoded version)

```
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] := SRC2[127:0]
ESAC
DEST := TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VINSERTF32x4 __m512 _mm512_insertf32x4( __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_maskz_insertf32x4( __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_insertf32x4( __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_maskz_insertf32x4( __mmask16 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_maskz_insertf32x4( __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 __mm512_insertf32x8( __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_maskz_insertf32x8( __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d __mm512d_insertf64x2( __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d __mm512d_maskz_insertf64x2( __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d __mm512d_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x4 __m512d __mm512d_insertf64x4( __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d __mm512d_maskz_insertf64x4( __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d __mm512d_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 __mm256_insertf128_ps( __m256 a, __m128 b, int offset);
VINSERTF128 __m256d __mm256d_insertf128_pd( __m256d a, __m128d b, int offset);
VINSERTF128 __m256i __mm256i_insertf128_si256( __m256i a, __m128i b, int offset);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”
Additionally:

#UD If VEX.L = 0.

EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions.”
## VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 3B / r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 3B / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3B / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 3B / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3B / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3A / r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3A / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The
second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

**VINSERTI32x4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256
   CASE (imm8[0]) OF
      0: TEMP_DEST[127:0] := SRC2[127:0]
   ESAC.
   FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TEMP_DEST[127:0] := SRC2[127:0]
   ESAC.
   FI;

FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TEMP_DEST[i+31:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] := 0
         FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VINSERTI64x2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256
   CASE (imm8[0]) OF
      0: TEMP_DEST[127:0] := SRC2[127:0]
   ESAC.
   FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TEMP_DEST[127:0] := SRC2[127:0]
   ESAC.
   FI;

FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := TEMP_DEST[i+63:i]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+63:i] := 0
         FI
   FI;
ENDFOR
IF k1[j] OR *no writemask*
   THEN DEST[i+63:i] := TMP_DEST[i+63:i]
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged* 
      ELSE ; zeroing-masking
         DEST[i+63:i] := 0
   FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VINSERTI32x8 (EVEX.U1.512 encoded version)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] := SRC2[255:0]
   1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged* 
      ELSE ; zeroing-masking
         DEST[i+31:i] := 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VINSERTI64x4 (EVEX.512 encoded version)
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
   0: TMP_DEST[255:0] := SRC2[255:0]
   1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := TMP_DEST[i+63:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged* 
      ELSE ; zeroing-masking
         DEST[i+63:i] := 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
**INSTRUCTION SET REFERENCE, V**

**VINSERTI128**

\[
\text{TEMP}[255:0] := \text{SRC1}[255:0]
\]

CASE (imm8[0]) OF
  0: \(
  \text{TEMP}[127:0] := \text{SRC2}[127:0]
  \)
  1: \(
  \text{TEMP}[255:128] := \text{SRC2}[127:0]
  \)
ESAC

\[
\text{DEST} := \text{TEMP}
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
\text{VINSERTI32x4} & \quad \text{_mm512i}_{\text{insert}}32x4(\_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x4} & \quad \text{_mm512i}_{\text{mask}}_{\text{insert}}32x4(\_m512i s, \_mmask16 k, \_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x4} & \quad \text{_mm512i}_{\text{maskz}}_{\text{insert}}32x4(\_mmask16 k, \_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x4} & \quad \text{mm512i} _{\text{insert}}32x4(\_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x4} & \quad \_mm512i _{\text{mask}}_{\text{insert}}32x4(\_m512i s, \_mmask8 k, \_m256i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x4} & \quad \_mm512i _{\text{maskz}}_{\text{insert}}32x4(\_mmask8 k, \_m256i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x8} & \quad \_mm512i _{\text{insert}}32x8(\_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI32x8} & \quad \_mm512i _{\text{mask}}_{\text{insert}}32x8(\_m512i s, \_mmask16 k, \_m512i a, \_m256i b, \text{int imm}); \\
\text{VINSERTI32x8} & \quad \_mm512i _{\text{maskz}}_{\text{insert}}32x8(\_mmask16 k, \_m512i a, \_m256i b, \text{int imm}); \\
\text{VINSERTI64x2} & \quad \text{mm256i} _{\text{insert}}64x2(\_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI64x2} & \quad \text{mm256i} _{\text{mask}}_{\text{insert}}64x2(\_m512i s, \_mmask8 k, \_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI64x2} & \quad \text{mm256i} _{\text{maskz}}_{\text{insert}}64x2(\_mmask8 k, \_m512i a, \_m128i b, \text{int imm}); \\
\text{VINSERTI64x4} & \quad \text{mm512i} _{\text{insert}}64x4(\_m512i a, \_m256i b, \text{int imm}); \\
\text{VINSERTI64x4} & \quad \text{mm512i} _{\text{mask}}_{\text{insert}}64x4(\_m512i s, \_mmask8 k, \_m512i a, \_m256i b, \text{int imm}); \\
\text{VINSERTI64x4} & \quad \text{mm512i} _{\text{maskz}}_{\text{insert}}64x4(\_mmask8 k, \_m512i a, \_m256i b, \text{int imm}); \\
\text{VINSERTI128} & \quad \text{mm256i} _{\text{insert}}128_{\text{si256}}(\_m256i a, \_m128i b, \text{int offset}); \\
\end{align*}
\]

**SIMD Floating-PointExceptions**

None.

**Other Exceptions**

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

\[
\#UD \quad \text{If VEX.L} = 0.
\]

EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions.”
VMASKMOV—Conditional SIMD Packed Loads and Stores

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.6F38.W0 2C/r VMASKMOVPS xmm1, xmm2, m128</td>
<td>RV M</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed single-precision values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VEX.256.6F38.W0 2C/r VMASKMOVPS ymm1, ymm2, m256</td>
<td>RV M</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed single-precision values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.128.6F38.W0 2D/r VMASKMOVPD xmm1, xmm2, m128</td>
<td>RV M</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed double precision values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VEX.256.6F38.W0 2D/r VMASKMOVPD ymm1, ymm2, m256</td>
<td>RV M</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed double precision values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.128.6F38.W0 2E/r VMASKMOVPS m128, xmm1, xmm2</td>
<td>MV R</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed single-precision values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VEX.256.6F38.W0 2E/r VMASKMOVPS m256, ymm1, ymm2</td>
<td>MV R</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed single-precision values from ymm2 using mask in ymm1.</td>
</tr>
<tr>
<td>VEX.128.6F38.W0 2F/r VMASKMOVPD m128, xmm1, xmm2</td>
<td>MV R</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed double precision values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VEX.256.6F38.W0 2F/r VMASKMOVPD m256, ymm1, ymm2</td>
<td>MV R</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed double precision values from ymm2 using mask in ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.
In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

**Operation**

**VMASKMOVPVS - 128-bit load**

\[
\begin{align*}
\text{DEST}[31:0] &= \text{IF (SRC1[31]) Load}_32(\text{mem}) \text{ ELSE 0} \\
\text{DEST}[63:32] &= \text{IF (SRC1[63]) Load}_32(\text{mem + 4}) \text{ ELSE 0} \\
\text{DEST}[95:64] &= \text{IF (SRC1[95]) Load}_32(\text{mem + 8}) \text{ ELSE 0} \\
\text{DEST}[127:97] &= \text{IF (SRC1[127]) Load}_32(\text{mem + 12}) \text{ ELSE 0} \\text{DEST}[\text{MAXVL-1:128}] &= 0
\end{align*}
\]

**VMASKMOPVPS - 256-bit load**

\[
\begin{align*}
\text{DEST}[31:0] &= \text{IF (SRC1[31]) Load}_32(\text{mem}) \text{ ELSE 0} \\
\text{DEST}[63:32] &= \text{IF (SRC1[63]) Load}_32(\text{mem + 4}) \text{ ELSE 0} \\
\text{DEST}[95:64] &= \text{IF (SRC1[95]) Load}_32(\text{mem + 8}) \text{ ELSE 0} \\
\text{DEST}[127:96] &= \text{IF (SRC1[127]) Load}_32(\text{mem + 12}) \text{ ELSE 0} \\
\text{DEST}[159:128] &= \text{IF (SRC1[159]) Load}_32(\text{mem + 16}) \text{ ELSE 0} \\
\text{DEST}[191:160] &= \text{IF (SRC1[191]) Load}_32(\text{mem + 20}) \text{ ELSE 0} \\
\text{DEST}[223:192] &= \text{IF (SRC1[223]) Load}_32(\text{mem + 24}) \text{ ELSE 0} \\
\text{DEST}[255:224] &= \text{IF (SRC1[255]) Load}_32(\text{mem + 28}) \text{ ELSE 0}
\end{align*}
\]

**VMASKMOVPS - 128-bit store**

\[
\begin{align*}
\text{IF (SRC1[31]) DEST}[31:0] &= \text{SRC2}[31:0] \\
\text{IF (SRC1[63]) DEST}[63:32] &= \text{SRC2}[63:32] \\
\text{IF (SRC1[95]) DEST}[95:64] &= \text{SRC2}[95:64] \\
\text{IF (SRC1[127]) DEST}[127:96] &= \text{SRC2}[127:96]
\end{align*}
\]

**VMASKMOVPS - 256-bit store**

\[
\begin{align*}
\text{IF (SRC1[31]) DEST}[31:0] &= \text{SRC2}[31:0] \\
\text{IF (SRC1[63]) DEST}[63:32] &= \text{SRC2}[63:32] \\
\text{IF (SRC1[95]) DEST}[95:64] &= \text{SRC2}[95:64] \\
\text{IF (SRC1[127]) DEST}[127:96] &= \text{SRC2}[127:96] \\
\text{IF (SRC1[159]) DEST}[159:128] &= \text{SRC2}[159:128] \\
\text{IF (SRC1[191]) DEST}[191:160] &= \text{SRC2}[191:160] \\
\text{IF (SRC1[223]) DEST}[223:192] &= \text{SRC2}[223:192] \\
\text{IF (SRC1[255]) DEST}[255:224] &= \text{SRC2}[255:224]
\end{align*}
\]
VMASKMOVPD - 128-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

VMASKMOVPD - 256-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
__m256 _mm256_maskload_ps(float const *a, __m256i mask)
void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
__m128 _mm_maskload_ps(float const *a, __m128i mask)
void _mm_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm_maskload_pd(double const *a, __m128i mask);
void _mm_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-23, “Type 6 Class Exception Conditions” (No AC# reported for any mask bit combinations).
Additionally:
#UD If VEX.W = 1.
VMAXPH—Return Maximum of Packed FP16 Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 5F /r VMAXPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Return the maximum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 5F /r VMAXPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Return the maximum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 5F /r VMAXPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Return the maximum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

### Operation

```python
def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
```

5-400  Vol. 2C  VMAXPH—Return Maximum of Packed FP16 Values
**VMAXPH dest, src1, src2**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         tsrc2 := SRC2.fp16[0]
      ELSE:
         tsrc2 := SRC2.fp16[j]
      DEST.fp16[j] := MAX(SRC1.fp16[j], tsrc2)
   ELSE IF *zeroing*:
      DEST.fp16[j] := 0
   // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VMAXPH __m128h __mm_mask_max_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h __mm_maskz_max_ph (__mmask8 k, __m128h a, __m128h b);
VMAXPH __m256h __mm256_mask_max_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h __mm256_maskz_max_ph (__mmask16 k, __m256h a, __m256h b);
VMAXPH __m512h __mm512_mask_max_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h __mm512_maskz_max_ph (__mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h __mm512_mask_max_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h __mm512_maskz_max_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h __mm512_max_round_ph (__m512h a, __m512h b, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VMAXSH—Return Maximum of Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0.5F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Return the maximum low FP16 value between xmm3/m16 and xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the maximum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

def MAX(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 > SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2

VMAXSH dest, src1, src2

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MAX(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VMAXSH __m128h _mm_mask_max_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_maskz_max_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_max_round_sh (__m128h a, __m128h b, int sae);
VMAXSH __m128h _mm_mask_max_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_maskz_max_sh (__mmask8 k, __m128h a, __m128h b);
VMAXSH __m128h _mm_max_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VMINPH—Return Minimum of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP:MAPS.W0 5D /r VMINPH xmm1{k1}{z}, xmm2, xmm3/ m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Return the minimum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP:MAPS.W0 5D /r VMINPH ymm1{k1}{z}, ymm2, ymm3/ m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Return the minimum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP:MAPS.W0 5D /r VMINPH zmm1{k1}{z}, zmm2, zmm3/ m512/m16bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Return the minimum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

### Operation

```python
def MIN(SRC1, SRC2):
    if (SRC1 == 0.0) and (SRC2 == 0.0):
        DEST := SRC2
    else if (SRC1 == NaN):
        DEST := SRC2
    else if (SRC2 == NaN):
        DEST := SRC2
    else if (SRC1 < SRC2):
        DEST := SRC1
    else:
        DEST := SRC2
```
VMINPH dest, src1, src2
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      tsrc2 := SRC2.fp16[0]
    ELSE:
      tsrc2 := SRC2.fp16[j]
    DEST.fp16[j] := MIN(SRC1.fp16[j], tsrc2)
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged
  DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VMINPH __m128h _mm_mask_min_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_maskz_min_ph (__mmask8 k, __m128h a, __m128h b);
VMINPH __m128h _mm_min_ph (__m128h a, __m128h b);
VMINPH __m256h _mm256_mask_min_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_maskz_min_ph (__mmask16 k, __m256h a, __m256h b);
VMINPH __m256h _mm256_min_ph (__m256h a, __m256h b);
VMINPH __m512h _mm512_mask_min_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_maskz_min_ph (__mmask32 k, __m512h a, __m512h b);
VMINPH __m512h _mm512_min_ph (__m512h a, __m512h b);
VMINPH __m512h _mm512_mask_min_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_maskz_min_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h _mm512_min_round_ph (__m512h a, __m512h b, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VMINSH—Return Minimum Scalar FP16 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 5D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Return the minimum low FP16 value between xmm3/m16 and xmm2. Stores the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the minimum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Operation

def MIN(SRC1, SRC2):
    IF (SRC1 = 0.0) and (SRC2 = 0.0):
        DEST := SRC2
    ELSE IF (SRC1 = NaN):
        DEST := SRC2
    ELSE IF (SRC2 = NaN):
        DEST := SRC2
    ELSE IF (SRC1 < SRC2):
        DEST := SRC1
    ELSE:
        DEST := SRC2
VMINSH dest, src1, src2
IF k1[0] OR *no writemask*:
    DEST fp16[0] := MIN(SRC1 fp16[0], SRC2 fp16[0])
ELSE IF *zeroing*:
    DEST fp16[0] := 0
// else dest fp16[j] remains unchanged

DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VMINSH __m128h __mm_mask_min_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h __mm_maskz_min_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
VMINSH __m128h __mm_min_round_sh (__m128h a, __m128h b, int sae);
VMINSH __m128h __mm_mask_min_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINSH __m128h __mm_maskz_min_sh (__mmask8 k, __m128h a, __m128h b);
VMINSH __m128h __mm_min_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

**VMOVSH—Move Scalar FP16 Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAPS.W0 10 /r VMOVSH xmm1[k1]{z}, m16</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Move FP16 value from m16 to xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAPS.W0 11 /r VMOVSH m16[k1], xmm1</td>
<td>B</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Move low FP16 value from xmm1 to m16 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAPS.W0 10 /r VMOVSH xmm1[k1]{z}, xmm2, xmm3</td>
<td>C</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
<tr>
<td>EVEX.LLIG.F3.MAPS.W0 11 /r VMOVSH xmm1[k1]{z}, xmm2, xmm3</td>
<td>D</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>N/A</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction moves a FP16 value to a register or memory location. The two register-only forms are aliases and differ only in where their operands are encoded; this is a side effect of the encodings selected.

**Operation**

**VMOVSH dest, src (two operand load)**

IF k1[0] or no writemask:

- DEST.fp16[0] := SRC.fp16[0]
ELSE IF *zeroing*:

- DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[MAXVL:16] := 0

**VMOVSH dest, src (two operand store)**

IF k1[0] or no writemask:

- DEST.fp16[0] := SRC.fp16[0]
// ELSE DEST.fp16[0] remains unchanged
VMOVSH dest, src1, src2 (three operand copy)
IF k1[0] or no writemask:
    DEST.fp16[0] := SRC2.fp16[0]
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// ELSE DEST.fp16[0] remains unchanged

DEST[MAXVL:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VMOVSH __m128h _mm_load_sh (void const* mem_addr);
VMOVSH __m128h _mm_mask_load_sh (__m128h src, __mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_maskz_load_sh (__mmask8 k, void const* mem_addr);
VMOVSH __m128h _mm_mask_move_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_maskz_move_sh (__mmask8 k, __m128h a, __m128h b);
VMOVSH __m128h _mm_move_sh (__m128h a, __m128h b);
VMOVSH void _mm_mask_store_sh (void * mem_addr, __mmask8 k, __m128h a);
VMOVSH void _mm_store_sh (void * mem_addr, __m128h a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-51, "Type E5 Class Exception Conditions."
VMOVW—Move Word

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP5.WiG 6E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Copy word from reg/m16 to xmm1.</td>
</tr>
<tr>
<td>VMOVW xmm1, reg/m16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.MAP5.WiG 7E /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Copy word from xmm1 to reg/m16.</td>
</tr>
<tr>
<td>VMOVW reg/m16, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description
This instruction either (a) copies one word element from an XMM register to a general-purpose register or memory location or (b) copies one word element from a general-purpose register or memory location to an XMM register. When writing a general-purpose register, the lower 16-bits of the register will contain the word value. The upper bits of the general-purpose register are written with zeros.

Operation

VMOVW dest, src (two operand load)
DEST.word[0] := SRC.word[0]
DEST[MAXVL-16] := 0

VMOVW dest, src (two operand store)
DEST.word[0] := SRC.word[0]
// upper bits of GPR DEST are zeroed

Intel C/C++ Compiler Intrinsic Equivalent
VMOVW short __m128i __mm_cvtsi128_si16 (__m128i a);
VMOVW __m128i __mm_cvtsi16_si128 (short a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instructions, see Table 2-57, “Type E9NF Class Exception Conditions.”
**VMULPH—Multiply Packed FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 59  /r VMULPH xmm1[k1]{z}, xmm2, xmm3/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from xmm3/m128/m16bcst to xmm2 and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 59  /r VMULPH ymm1[k1]{z}, ymm2, ymm3/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Multiply packed FP16 values from ymm3/m256/m16bcst to ymm2 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 59  /r VMULPH zmm1[k1]{z}, zmm2, zmm3/m512/m16bcst {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply packed FP16 values in zmm3/m512/m16bcst with zmm2 and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction multiplies packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

### Operation

**VMULPH (EVEX encoded versions) when src2 operand is a register**

VL = 128, 256 or 512

KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
  SET_RM(EVEX.RC)
ELSE
  SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
VMULPH (EVEX encoded versions) when src2 operand is a memory source

VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
   IF k1[j] OR *no writemask*:
      IF EVEX.b = 1:
         DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[0]
      ELSE:
         DEST.fp16[j] := SRC1.fp16[j] * SRC2.fp16[j]
      ELSE IF *zeroing*:
         DEST.fp16[j] := 0
      // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VMULPH __m128h _mm_mask_mul_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_maskz_mul_ph (__mmask8 k, __m128h a, __m128h b);
VMULPH __m128h _mm_mul_ph (__m128h a, __m128h b);
VMULPH __m256h _mm256_mask_mul_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_maskz_mul_ph (__mmask16 k, __m256h a, __m256h b);
VMULPH __m256h _mm256_mul_ph (__m256h a, __m256h b);
VMULPH __m512h _mm512_mask_mul_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_maskz_mul_ph (__mmask32 k, __m512h a, __m512h b);
VMULPH __m512h _mm512_mul_ph (__m512h a, __m512h b);
VMULPH __m512h _mm512_mask_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_maskz_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VMULPH __m512h _mm512_mul_round_ph (__m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions.”
VMULSH—Multiply Scalar FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAPS.W0 59 /r VMULSH xmm1[k1]{z}, xmm2, xmm3/m16 {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Multiply the low FP16 value in xmm3/m16 by low FP16 value in xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

This instruction multiplies the low FP16 value from the source operands and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

**VMULSH (EVEX encoded versions)**

IF EVEX.b = 1 and SRC2 is a register:

- SET_RM(EVEX.RC)
ELSE
- SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:

- DEST.fp16[0] := SRC1.fp16[0] * SRC2.fp16[0]
ELSE IF *zeroing*:

- DEST.fp16[0] := 0

// else dest.fp16[0] remains unchanged


DEST[MAXVL-1:VL] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

```
VMULSH __m128h _mm_mask_mul_round_sh (__m128h src, __m128h k, __m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_maskz_mul_round_sh (__m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mul_round_sh (__m128h a, __m128h b, int rounding);
VMULSH __m128h _mm_mask_mul_sh (__m128h src, __m128h k, __m128h a, __m128h b);
VMULSH __m128h _mm_maskz_mul_sh (__m128h a, __m128h b);
VMULSH __m128h _mm_mul_sh (__m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions.”
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, xmm2, xmm3/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in xmm3/m128/m32bcst and xmm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, ymm2, ymm3/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in ymm3/m256/m32bcst and ymm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, zmm2, zmm3/m512/m32bcst</td>
<td>A V/V</td>
<td>AVX512F AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in zmm3/m512/m32bcst and zmm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, xmm2, xmm3/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in xmm3/m128/m64bcst and xmm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, ymm2, ymm3/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in ymm3/m256/m64bcst and ymm2.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, zmm2, zmm3/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512F AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in zmm3/m512/m64bcst and zmm2.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction writes an even/odd pair of mask registers. The mask register destination indicated in the MODRM.REG field is used to form the basis of the register pair. The low bit of that field is masked off (set to zero) to create the first register of the pair.

EVEX.aaa and EVEX.z must be zero.
**Operation**

**VP2INTERSECTD destmask, src1, src2**  
(KL, VL) = (4, 128), (8, 256), (16, 512)

// dest_mask_reg_id is the register id specified in the instruction for destmask  
dest_base := dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers  
maskregs[dest_base+0][MAX_KL-1:0] := 0  
maskregs[dest_base+1][MAX_KL-1:0] := 0

FOR i := 0 to KL-1:  
    FOR j := 0 to KL-1:  
        match := (src1.dword[i] == src2.dword[j])  
        maskregs[dest_base+0].bit[i] |= match  
        maskregs[dest_base+1].bit[j] |= match

**VP2INTERSECTQ destmask, src1, src2**  
(KL, VL) = (2, 128), (4, 256), (8, 512)

// dest_mask_reg_id is the register id specified in the instruction for destmask  
dest_base := dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers  
maskregs[dest_base+0][MAX_KL-1:0] := 0  
maskregs[dest_base+1][MAX_KL-1:0] := 0

FOR i = 0 to KL-1:  
    FOR j = 0 to KL-1:  
        match := (src1.qword[i] == src2.qword[j])  
        maskregs[dest_base+0].bit[i] |= match  
        maskregs[dest_base+1].bit[j] |= match

**Intel C/C++ Compiler Intrinsic Equivalent**

VP2INTERSECTD void _mm_2intersect_epi32(__m128i, __m128i, __mmask8 *, __mmask8 *);  
VP2INTERSECTD void _mm256_2intersect_epi32(__m256i, __m256i, __mmask8 *, __mmask8 *);  
VP2INTERSECTD void _mm512_2intersect_epi32(__m512i, __m512i, __mmask16 *, __mmask16 *);  
VP2INTERSECTQ void _mm_2intersect_epi64(__m128i, __m128i, __mmask8 *, __mmask8 *);  
VP2INTERSECTQ void _mm256_2intersect_epi64(__m256i, __m256i, __mmask8 *, __mmask8 *);  
VP2INTERSECTQ void _mm512_2intersect_epi64(__m512i, __m512i, __mmask8 *, __mmask8 *);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-50, "Type E4NF Class Exception Conditions."
VPBLENDDD—Blend Packed Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F3A.W0 02 /r ib VPBLENDDD xmm1, xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select dwords from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 02 /r ib VPBLENDDD ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select dwords from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding dword in the destination is copied from the source. If a bit in the mask, corresponding to a dword, is “1”, then the dword is copied, else the dword is unchanged.

**VEX.128 encoded version:** The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

**VPBLENDDD (VEX.256 encoded version)**

IF (imm8[0] == 1) THEN DEST[31:0] := SRC2[31:0]
ELSE DEST[31:0] := SRC1[31:0]
ELSE DEST[95:64] := SRC1[95:64]
ELSE DEST[127:96] := SRC1[127:96]
VPBLEND (VEX.128 encoded version)
IF (imm8[0] == 1) THEN DEST[31:0] := SRC2[31:0]
ELSE DEST[31:0] := SRC1[31:0]
ELSE DEST[95:64] := SRC1[95:64]
ELSE DEST[127:96] := SRC1[127:96]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPBLEND: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)
VPBLEND: __m256i _mm256_blend_epi32 ( __m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-21, "Type 4 Class Exception Conditions."
Additionally:
#UD If VEX.W = 1.
VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.</td>
<td></td>
</tr>
<tr>
<td>VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.</td>
<td></td>
</tr>
<tr>
<td>VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A/V/V</td>
<td>AVX512BW</td>
<td>Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.</td>
<td></td>
</tr>
<tr>
<td>VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.</td>
<td></td>
</tr>
<tr>
<td>VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.</td>
<td></td>
</tr>
<tr>
<td>VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A/V/V</td>
<td>AVX512BW</td>
<td>Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).
Operation

VPBLENDMB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SRC2[i+7:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN DEST[i+7:i] := SRC1[i+7:i]
    ELSE ; zeroing-masking
      DEST[i+7:i] := 0
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;

VPBLENDMW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC2[i+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN DEST[i+15:i] := SRC1[i+15:i]
    ELSE ; zeroing-masking
      DEST[i+15:i] := 0
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);

VPBLENDMW __m512i _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."
VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}[z], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}[z], zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}[z], ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}[z], zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.
Operation

**VPBLENDMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no controlmask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] := SRC2[31:0]
        ELSE
          DEST[i+31:i] := SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN DEST[i+31:i] := SRC1[i+31:i]
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI;
    FI;
  ENDFOR

DEST[MAXVL-1:VL] := 0;

**VPBLENDMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no controlmask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] := SRC2[31:0]
        ELSE
          DEST[i+31:i] := SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN DEST[i+31:i] := SRC1[i+31:i]
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI;
    FI;
  ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-49, “Type E4 Class Exception Conditions.”
VPBROADCASTB/W/D/Q—Load With Broadcast Integer Data From General Purpose Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all doublewords in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all doublewords in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all doublewords in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quadwords in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quadwords in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quadwords in the 512-bit destination subject to writemask k1.</td>
</tr>
</tbody>
</table>

NOTES:
1. EVEX.W in non-64 bit is ignored; the instruction behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPBROADCASTB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SRC[7:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged* ; zeroing-masking
    ELSE
      DEST[i+7:i] := 0
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPBROADCASTW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged* ; zeroing-masking
    ELSE
      DEST[i+15:i] := 0
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPBROADCASTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged* ; zeroing-masking
    ELSE
      DEST[i+31:i] := 0
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPBROADCASTQ (EVEX encoded versions)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

\[ \text{FOR } j := 0 \text{ TO } KL-1 \]
\[ i := j \times 64 \]
\[ \text{IF } k1[j] \text{ OR *no writemask*} \]
\[ \quad \text{THEN } \text{DEST}[i+63:i] := \text{SRC}[63:0] \]
\[ \quad \text{ELSE} \]
\[ \quad \text{IF *merging-masking*} \quad ; \text{merging-masking} \]
\[ \quad \quad \text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged*} \]
\[ \quad \quad \text{ELSE} \quad ; \text{zeroing-masking} \]
\[ \quad \quad \text{DEST}[i+63:i] := 0 \]
\[ \quad \FI \]
\[ \FI \]
\[ \text{ENDFOR} \]
\[ \text{DEST}[\text{MAXVL}-1:VL] := 0 \]

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB\_\_m512i\_\_mm512\_\_mask\_set1\_\_epi8\((\_\_m512i\_s,\_\_mmask64\_k,\_\text{int a});\)
VPBROADCASTB\_\_m512i\_\_mm512\_\_maskz\_set1\_\_epi8\((\_\_mmask64\_k,\_\text{int a});\)
VPBROADCASTB\_\_m256i\_\_mm256\_\_mask\_set1\_\_epi8\((\_\_m256i\_s,\_\_mmask32\_k,\_\text{int a});\)
VPBROADCASTB\_\_m256i\_\_mm256\_\_maskz\_set1\_\_epi8\((\_\_mmask32\_k,\_\text{int a});\)
VPBROADCASTB\_\_m128i\_\_mm\_\_mask\_set1\_\_epi8\((\_\_m128i\_s,\_\_mmask16\_k,\_\text{int a});\)
VPBROADCASTB\_\_m128i\_\_mm\_\_maskz\_set1\_\_epi8\((\_\_mmask16\_k,\_\text{int a});\)
VPBROADCASTD\_\_m512i\_\_mm512\_\_mask\_set1\_\_epi32\((\_\_m512i\_s,\_\_mmask64\_k,\_\text{int a});\)
VPBROADCASTD\_\_m512i\_\_mm512\_\_maskz\_set1\_\_epi32\((\_\_mmask64\_k,\_\text{int a});\)
VPBROADCASTD\_\_m256i\_\_mm256\_\_mask\_set1\_\_epi32\((\_\_m256i\_s,\_\_mmask32\_k,\_\text{int a});\)
VPBROADCASTD\_\_m256i\_\_mm256\_\_maskz\_set1\_\_epi32\((\_\_mmask32\_k,\_\text{int a});\)
VPBROADCASTD\_\_m128i\_\_mm\_\_mask\_set1\_\_epi32\((\_\_m128i\_s,\_\_mmask16\_k,\_\text{int a});\)
VPBROADCASTD\_\_m128i\_\_mm\_\_maskz\_set1\_\_epi32\((\_\_mmask16\_k,\_\text{int a});\)
VPBROADCASTQ\_\_m512i\_\_mm512\_\_mask\_set1\_\_epi64\((\_\_m512i\_s,\_\_mmask64\_k,\_\text{int64 a});\)
VPBROADCASTQ\_\_m512i\_\_mm512\_\_maskz\_set1\_\_epi64\((\_\_mmask64\_k,\_\text{int64 a});\)
VPBROADCASTQ\_\_m256i\_\_mm256\_\_mask\_set1\_\_epi64\((\_\_m256i\_s,\_\_mmask32\_k,\_\text{int64 a});\)
VPBROADCASTQ\_\_m256i\_\_mm256\_\_maskz\_set1\_\_epi64\((\_\_mmask32\_k,\_\text{int64 a});\)
VPBROADCASTQ\_\_m128i\_\_mm\_\_mask\_set1\_\_epi64\((\_\_m128i\_s,\_\_mmask16\_k,\_\text{int64 a});\)
VPBROADCASTQ\_\_m128i\_\_mm\_\_maskz\_set1\_\_epi64\((\_\_mmask16\_k,\_\text{int64 a});\)
VPBROADCASTD\_\_m512i\_\_mm512\_\_mask\_set1\_\_epi16\((\_\_m512i\_s,\_\_mmask32\_k,\_\text{int a});\)
VPBROADCASTD\_\_m512i\_\_mm512\_\_maskz\_set1\_\_epi16\((\_\_mmask32\_k,\_\text{int a});\)
VPBROADCASTD\_\_m256i\_\_mm256\_\_mask\_set1\_\_epi16\((\_\_m256i\_s,\_\_mmask16\_k,\_\text{int a});\)
VPBROADCASTD\_\_m256i\_\_mm256\_\_maskz\_set1\_\_epi16\((\_\_mmask16\_k,\_\text{int a});\)
VPBROADCASTD\_\_m128i\_\_mm\_\_mask\_set1\_\_epi16\((\_\_m128i\_s,\_\_mmask8\_k,\_\text{int a});\)
VPBROADCASTD\_\_m128i\_\_mm\_\_maskz\_set1\_\_epi16\((\_\_mmask8\_k,\_\text{int a});\)

Exceptions

EVEX-encoded instructions, see Table 2-55, "Type E7NM Class Exception Conditions."

Additionally:

#UD  \quad \text{If EVEX.vvvv} ! = 1111B.
### VPBROADCAST—Load Integer and Broadcast

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8</td>
<td>A V/V AVX2</td>
<td>Broadcast a byte integer in the source operand to sixteen locations in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8</td>
<td>A V/V AVX2</td>
<td>Broadcast a byte integer in the source operand to thirty-two locations in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 78 /r VPBROADCASTB ymm1<a href="z">k1</a>, xmm2/m8</td>
<td>B V/V AVX512VL AVX512BW</td>
<td>Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1<a href="z">k1</a>, xmm2/m8</td>
<td>B V/V AVX512VL AVX512BW</td>
<td>Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1<a href="z">k1</a>, xmm2/m8</td>
<td>B V/V AVX512BW</td>
<td>Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16</td>
<td>A V/V AVX2</td>
<td>Broadcast a word integer in the source operand to eight locations in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16</td>
<td>A V/V AVX2</td>
<td>Broadcast a word integer in the source operand to sixteen locations in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1<a href="z">k1</a>, xmm2/m16</td>
<td>B V/V AVX512VL AVX512BW</td>
<td>Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1<a href="z">k1</a>, xmm2/m16</td>
<td>B V/V AVX512VL AVX512BW</td>
<td>Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1<a href="z">k1</a>, xmm2/m16</td>
<td>B V/V AVX512BW</td>
<td>Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32</td>
<td>A V/V AVX2</td>
<td>Broadcast a dword integer in the source operand to four locations in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32</td>
<td>A V/V AVX2</td>
<td>Broadcast a dword integer in the source operand to eight locations in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1<a href="z">k1</a>, xmm2/m32</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1<a href="z">k1</a>, xmm2/m32</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1<a href="z">k1</a>, xmm2/m32</td>
<td>B V/V AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64</td>
<td>A V/V AVX2</td>
<td>Broadcast a qword element in source operand to two locations in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64</td>
<td>A V/V AVX2</td>
<td>Broadcast a qword element in source operand to four locations in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1<a href="z">k1</a>, xmm2/m64</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1<a href="z">k1</a>, xmm2/m64</td>
<td>B V/V AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1<a href="z">k1</a>, xmm2/m64</td>
<td>B V/V AVX512F</td>
<td>Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>E</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.
EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

---

**Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)**

**Figure 5-17. VPBROADCASTD Operation (128-bit version)**

**Figure 5-18. VPBROADCASTQ Operation (256-bit version)**
VPBROADCAST—Load Integer and Broadcast

Operation

VPBROADCASTB (EVEX encoded versions)

\[(KL, VL) = (16, 128), (32, 256), (64, 512)\]

FOR \(j \) := 0 TO \(KL-1\)

\[i := j * 8\]

IF \(k1[j] \text{ OR *no writemask*}\)

THEN \(\text{DEST}[i+7:i] := \text{SRC}[7:0]\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+7:i] remains unchanged*

ELSE ; zeroing-masking

\(\text{DEST}[i+7:i] := 0\)

FI

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

Figure 5-19. VBROADCASTI128 Operation (256-bit version)

Figure 5-20. VBROADCASTI256 Operation (512-bit version)
**VPBROADCAST (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] := 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPBROADCASTD (128 bit version)**

temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0

**VPBROADCASTD (VEX.256 encoded version)**

temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0

**VPBROADCASTD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] := 0
VPBROADCASTQ (VEX.256 encoded version)

\[
\text{temp} := \text{SRC}[63:0] \\
\text{DEST}[63:0] := \text{temp} \\
\text{DEST}[127:64] := \text{temp} \\
\text{DEST}[191:128] := \text{temp} \\
\text{DEST}[255:192] := \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] := 0
\]

VPBROADCASTQ (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
\qquad i := j \times 64 \\
\text{IF } k1[j] \text{ OR } \text{no writemask} \\
\qquad \text{THEN } \text{DEST}[i+63:i] := \text{SRC}[63:0] \\
\text{ELSE} \\
\qquad \text{IF merging-masking} \\
\qquad \text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged} \\
\text{ELSE} \\
\qquad \text{zeroing-masking} \\
\qquad \text{DEST}[i+63:i] := 0 \\
\text{FI}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VBROADCASTI32x2 (EVEX encoded versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
\qquad i := j \times 32 \\
\quad n := (j \mod 2) \times 32 \\
\text{IF } k1[j] \text{ OR } \text{no writemask} \\
\qquad \text{THEN } \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \\
\text{ELSE} \\
\qquad \text{IF merging-masking} \\
\qquad \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged} \\
\text{ELSE} \\
\qquad \text{zeroing-masking} \\
\qquad \text{DEST}[i+31:i] := 0 \\
\text{FI}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VBROADCASTI128 (VEX.256 encoded version)

\[
\text{temp} := \text{SRC}[127:0] \\
\text{DEST}[127:0] := \text{temp} \\
\text{DEST}[255:128] := \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] := 0
\]
**V_broadcastI32x4 (EVEX encoded versions)**

\((KL, VL) = (8, 256), (16, 512)\)

FOR \( j := 0 \) TO \( KL - 1 \)

\( i := j \times 32 \)
\( n := (j \mod 4) \times 32 \)

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \)

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+31:i] \text{ remains unchanged} \)

ELSE ; zeroing-masking

\( \text{DEST}[i+31:i] := 0 \)

FI

FI;
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)

**V_broadcastI64x2 (EVEX encoded versions)**

\((KL, VL) = (8, 256), (16, 512)\)

FOR \( j := 0 \) TO \( KL - 1 \)

\( i := j \times 64 \)
\( n := (j \mod 2) \times 64 \)

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+63:i] := \text{SRC}[n+63:n] \)

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+63:i] \text{ remains unchanged} \)

ELSE ; zeroing-masking

\( \text{DEST}[i+63:i] := 0 \)

FI

FI;
ENDFOR;

**V_broadcastI32x8 (EVEX.U1.512 encoded version)**

FOR \( j := 0 \) TO \( 15 \)

\( i := j \times 32 \)
\( n := (j \mod 8) \times 32 \)

IF \( k1[j] \) OR *no writemask*

THEN \( \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \)

ELSE

IF *merging-masking* ; merging-masking

THEN \( \text{DEST}[i+31:i] \text{ remains unchanged} \)

ELSE ; zeroing-masking

\( \text{DEST}[i+31:i] := 0 \)

FI

FI;
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)
**VBROADCAST16X4 (EVEX_512 encoded version)**

FOR j := 0 TO 7
  i := j * 64
  n := (j modulo 4) * 64
  IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] := SRC[n+63:n]
  ELSE
      IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
          DEST[i+63:i] := 0
      FI
  FI;
ENDIF
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBROADCASTB __m512i _mm512_broadcastb_epi8( __m128i a);
VPBROADCASTB __m512i __mm512_mask_broadcastb_epi8( __m512i s, __m512i mmask64 k, __m128i a);
VPBROADCASTB __m512i __mm512_maskz_broadcastb_epi8( __mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i __mm256_mask_broadcastb_epi8( __m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i __mm256_maskz_broadcastb_epi8( __mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8( __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8( __m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8( __mmask16 k, __m128i a);

VPBROADCASTQ __m512i _mm512_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m512i __mm512_mask_broadcastq_epi64( __m512i s, __mmask64 k, __m128i a);
VPBROADCASTQ __m512i __mm512_maskz_broadcastq_epi64( __mmask64 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i __mm256_mask_broadcastq_epi64( __m256i s, __mmask32 k, __m128i a);
VPBROADCASTQ __m256i __mm256_maskz_broadcastq_epi64( __mmask32 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64( __m128i s, __mmask16 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64( __mmask16 k, __m128i a);

VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i __mm512_mask_broadcastw_epi16( __m512i s, __mmask64 k, __m128i a);
VPBROADCASTW __m512i __mm512_maskz_broadcastw_epi16( __mmask64 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i __mm256_mask_broadcastw_epi16( __m256i s, __mmask32 k, __m128i a);
VPBROADCASTW __m256i __mm256_maskz_broadcastw_epi16( __mmask32 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16( __m128i a);
VPBROADCASTW __m128i _mm_mask_broadcastw_epi16( __m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16( __mmask8 k, __m128i a);

VPBROADCAST—I load Integer and Broadcast
INSTRUCTION SET REFERENCE, V

VBROADCASTI32x2 __m512i __mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i __mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i __mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i __mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i __mm_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m128i __mm_mask_broadcast_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i __mm_maskz_broadcast_i32x2(__mmask8 k, __m128i a);

VBROADCASTI32x4 __m512i __mm512_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m512i __mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBroadcastI32x4 __m512i __mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i __mm256_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBroadcastI32x4 __m256i __mm256_mask_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x4 __m256i __mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i __mm512_broadcast_i32x8(__m256i a);
VBROADCASTI32x8 __m512i __mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m128i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m256i s, __mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m256i s, __mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m256i s, __mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__m256i s, __mmask8 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions."
EVEX-encoded instructions, syntax with reg/mem operand, see Table 2-53, "Type E6 Class Exception Conditions."
Additionally:

#UD If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
If EVEX.L’L = 0 for VPBROADCASTI32x4/VPBROADCASTI64x2.
If EVEX.L’L < 10b for VPBROADCASTI32x8/VPBROADCASTI64x4.
VPBROADCASTM—Broadcast Mask to Vector Register

**Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128). EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**VPBROADCASTM2Q**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

  i := j*64

  DEST[i+63:i] := ZeroExtend(SRC[7:0])

ENDFOR

DEST[MAXVL-1:VL] := 0

**VPBROADCASTMw2D**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

  i := j*32

  DEST[i+31:i] := ZeroExtend(SRC[15:0])

ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32(__mmask16);
VPBROADCASTMB2Q __m256i _mm256_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m256i _mm256_broadcastmw_epi32(__mmask8);
VPBROADCASTMB2Q __m128i _mm_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m128i _mm_broadcastmw_epi32(__mmask8);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-54, "Type E6NF Class Exception Conditions."
VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 3F /r ib VPCMPB k1 [k2], xmm2, xmm3/m128, imm8</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 3F /r ib VPCMPB k1 [k2], ymm2, ymm3/m256, imm8</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3F /r ib VPCMPB k1 [k2], zmm2, zmm3/m512, imm8</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 3E /r ib VPCMPUB k1 [k2], xmm2, xmm3/m128, imm8</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 3E /r ib VPCMPUB k1 [k2], ymm2, ymm3/m256, imm8</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3E /r ib VPCMPUB k1 [k2], zmm2, zmm3/m512, imm8</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.
The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-21.

### Table 5-21. Pseudo-Op and VPCMP* Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>PCMPM Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCMPEQ* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VPCMPLT* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VPCMPLE* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VPCMPNEQ* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VPPCMPNLT* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VPCMPNLE* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 6</td>
</tr>
</tbody>
</table>

---

**Operation**

CASE (COMPARISON PREDICATE) OF
0: OP := EQ;
1: OP := LT;
2: OP := LE;
3: OP := FALSE;
4: OP := NEQ;
5: OP := NLT;
6: OP := NLE;
7: OP := TRUE;
ESAC;

**VPCMPB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k2[j] OR *no writemask*
    THEN
      CMP := SRC1[i+7:i] OP SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; Fl;
      ELSE
        DEST[j] = 0 ; zeroing-masking onlyFl;
    FI;
  ENDIF;
ENDFOR
DEST[MAX_KL-1:KL] := 0
**VPCMPUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE
            DEST[j] = 0; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPB __m512 __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __m512 __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __m512 __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __m512 __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __m512 __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __m512 __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-21.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>VPCMPD k1 [k2], xmm2, xmm3/m128/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>VPCMPD k1 [k2], ymm2, ymm3/m256/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
<tr>
<td>VPCMPD k1 [k2], zmm2, zmm3/m512/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>VPCMPUD k1 [k2], xmm2, xmm3/m128/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>VPCMPUD k1 [k2], ymm2, ymm3/m256/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
<tr>
<td>VPCMPUD k1 [k2], zmm2, zmm3/m512/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5-440 Vol. 2C VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP := EQ;
  1: OP := LT;
  2: OP := LE;
  3: OP := FALSE;
  4: OP := NEQ;
  5: OP := NLT;
  6: OP := NLE;
  7: OP := TRUE;
ESAC;

VPCMPD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
        ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
    ELSE DEST[j] := 0 ; zeroing-masking onlyFI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPCMPUD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
        ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
    ELSE DEST[j] := 0 ; zeroing-masking onlyFI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 __mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 __mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m512i a, __m512i b);
VPCMPD __mmask16 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 __mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 __mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m512i a, __m512i b);
VPCMPUD __mmask16 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/ En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description
Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-21.
**Operation**

CASE (COMPARISON PREDICATE) OF
   0: OP := EQ;
   1: OP := LT;
   2: OP := LE;
   3: OP := FALSE;
   4: OP := NEQ;
   5: OP := NLT;
   6: OP := NLE;
   7: OP := TRUE;
ESAC;

**VPCMPQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
        ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE DEST[j] := 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

**VPCMPUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
        ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE DEST[j] := 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPQ __mmask8 __mm_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 __mm_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 __mm_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 __mm cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPUQ __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask  

Vol. 2C  5-445
**VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 3F /r ib VPCMPW k1 (k2), xmm2, xmm3/m128, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 3F /r ib VPCMPW k1 (k2), ymm2, ymm3/m256, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3F /r ib VPCMPW k1 (k2), zmm2, zmm3/m512, imm8</td>
<td>A V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 3E /r ib VPCMPUW k1 (k2), xmm2, xmm3/m128, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 3E /r ib VPCMPUW k1 (k2), ymm2, ymm3/m256, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3E /r ib VPCMPUW k1 (k2), zmm2, zmm3/m512, imm8</td>
<td>A V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-21.
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP := EQ;
  1: OP := LT;
  2: OP := LE;
  3: OP := FALSE;
  4: OP := NEQ;
  5: OP := NLT;
  6: OP := NLE;
  7: OP := TRUE;
ESAC;

VPCMPw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k2[j] OR *no writemask*
    THEN
      ICMP := SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE
        DEST[j] = 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPCMPUw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k2[j] OR *no writemask*
    THEN
      CMP := SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE
        DEST[j] = 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPW __mmask32 __mm512_cmp_epi16_mask(__m512i a, __m512i b, int cmp);
VPCMPW __mmask32 __mm512_mask_cmp_epi16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 __mm256_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask16 __mm256_mask_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8  __mm_cmp_epi16_mask(__m128i a, __m128i b, int cmp);
VPCMPW __mmask8  __mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m512i a, __m512i b);
VPCMPW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask16 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8  __mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m128i a, __m128i b);
VPCMPW __mmask8  __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."
VPcompressB/VPcompressW—Store Sparse Packed Byte/Word Integer Values Into Dense Memory/Register

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 63 /r VPcompressB m128[k1], xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 63 /r VPcompressB xmm1[k1][z], xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 63 /r VPcompressB m256[k1], ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 63 /r VPcompressB ymm1[k1][z], ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 63 /r VPcompressB m512[k1], zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 63 /r VPcompressB zmm1[k1][z], zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 63 /r VPcompressW m128[k1], xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 63 /r VPcompressW xmm1[k1][z], xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 63 /r VPcompressW m256[k1], ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 63 /r VPcompressW ymm1[k1][z], ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2, AVX512VL</td>
<td>Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 63 /r VPcompressW m512[k1], zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 63 /r VPcompressW zmm1[k1][z], zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Description**

Compress (stores) up to 64 bytes or 32 words from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvv is reserved and must be 1111b otherwise instructions will #UD.

Moves up to 512 bits of packed byte values from the source operand (second operand) to the destination operand (first operand). This instruction is used to store partial contents of a vector register into a byte vector or single memory location using the active elements in operand writemask.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

This instruction supports memory fault suppression.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg/r/m (w)</td>
<td>ModRM:reg/r</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg/r/m (w)</td>
<td>ModRM:reg/r</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VPCOMPRESSB store form**

\[(KL, VL) = (16, 128), (32, 256), (64, 512)\]

\[k := 0\]

FOR \(j := 0 \) TO \(KL-1\):

IF \(k \bmod{8} \) OR *no writemask*:

\[\text{DEST}.byte[k] := \text{SRC}.byte[j]\]

\[k := k + 1\]

**VPCOMPRESSB reg-reg form**

\[(KL, VL) = (16, 128), (32, 256), (64, 512)\]

\[k := 0\]

FOR \(j := 0 \) TO \(KL-1\):

IF \(k \bmod{8} \) OR *no writemask*:

\[\text{DEST}.byte[k] := \text{SRC}.byte[j]\]

\[k := k + 1\]

IF *merging-masking*:

*\[\text{DEST}[VL-1:k*8] \text{ remains unchanged}\]*

ELSE \[\text{DEST}[VL-1:k*8] := 0\]

\[\text{DEST}[MAX\_VL-1:VL] := 0\]

**VPCOMPRESSW store form**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[k := 0\]

FOR \(j := 0 \) TO \(KL-1\):

IF \(k \bmod{16} \) OR *no writemask*:

\[\text{DEST}.word[k] := \text{SRC}.word[j]\]

\[k := k + 1\]

**VPCOMPRESSW reg-reg form**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[k := 0\]

FOR \(j := 0 \) TO \(KL-1\):

IF \(k \bmod{16} \) OR *no writemask*:

\[\text{DEST}.word[k] := \text{SRC}.word[j]\]

\[k := k + 1\]

IF *merging-masking*:

*\[\text{DEST}[VL-1:k*16] \text{ remains unchanged}\]*

ELSE \[\text{DEST}[VL-1:k*16] := 0\]

\[\text{DEST}[MAX\_VL-1:VL] := 0\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSB __m128i _mm_mask_compress_epi8(__m128i, __mmask16, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi8(__mmask16, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi8(__m256i, __mmask32, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi8(__mmask32, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi8(__m512i, __mmask64, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi8(__mmask64, __m512i);
VPCOMPRESSB void _mm_mask_compressstoreu_epi8(void*, __mmask16, __m128i);
VPCOMPRESSB void _mm256_mask_compressstoreu_epi8(void*, __mmask32, __m256i);
VPCOMPRESSB void _mm512_mask_compressstoreu_epi8(void*, __mmask64, __m512i);
VPCOMPRESSB __m128i _mm_mask_compress_epi16(__m128i, __mmask8, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi16(__mmask8, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi16(__m256i, __mmask16, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi16(__mmask16, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi16(__m512i, __mmask32, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi16(__mmask32, __m512i);
VPCOMPRESSB void _mm_mask_compressstoreu_epi16(void*, __mmask8, __m128i);
VPCOMPRESSB void _mm256_mask_compressstoreu_epi16(void*, __mmask16, __m256i);
VPCOMPRESSB void _mm512_mask_compressstoreu_epi16(void*, __mmask32, __m512i);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”
INSTRUCTION SET REFERENCE, V

VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed doubleword integer values from xmm2 to xmm1/m128 using control mask k1.</td>
</tr>
<tr>
<td>VPCOMPRESSD xmm1/m128 (k1)[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed doubleword integer values from ymm2 to ymm1/m256 using control mask k1.</td>
</tr>
<tr>
<td>VPCOMPRESSD ymm1/m256 (k1)[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed doubleword integer values from zmm2 to zmm1/m512 using control mask k1.</td>
</tr>
<tr>
<td>VPCOMPRESSD zmm1/m512 (k1)[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

VPCOMPRESSD (EVEX encoded versions) store form

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE := 32
k := 0
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no controlmask*
  THEN
    DEST[k+SIZE-1:k] := SRC[i+31:j]
    k := k + SIZE
  FI;
ENDFOR;
VPCOMPRESSD (EVEX encoded versions) reg-reg form

(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+31:i]
            k := k + SIZE
        FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] := 0
    FI
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32( __mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32( __mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32( __mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register

**Opcode/Instruction** | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
---|---|---|---|---
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 (k1)[z], xmm2 | A | V/V | AVX512VL AVX512F | Compress packed quadword integer values from xmm2 to xmm1/m128 using control mask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 (k1)[z], ymm2 | A | V/V | AVX512VL AVX512F | Compress packed quadword integer values from ymm2 to ymm1/m256 using control mask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 (k1)[z], zmm2 | A | V/V | AVX512F | Compress packed quadword integer values from zmm2 to zmm1/m512 using control mask k1.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

VPCOMPRESSQ (EVEX encoded versions) store form

(KL, VL) = (2, 128), (4, 256), (8, 512)

\[ \text{SIZE} := 64 \]

\[ k := 0 \]

FOR \( j := 0 \) TO \( KL-1 \)

\[ i := j \ast 64 \]

IF \( k1[j] \) OR \( *\text{no controlmask}^* \)

THEN

\[ \text{DEST}[k+\text{SIZE}-1:k] := \text{SRC}[i+63:i] \]

\[ k := k + \text{SIZE} \]

FI;

ENFOR
VPCOMPRESSQ (EVEX encoded versions) reg-reg form

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

\[\text{SIZE} := 64\]

\[k := 0\]

\[\text{FOR } j := 0 \text{ TO } KL-1\]

\[i := j \times 64\]

\[\text{IF } k1[j] \text{ OR } \text{no controlmask}\]

\[\text{THEN}\]

\[\text{DEST}[k+\text{SIZE}-1:k] := \text{SRC}[i+63:i]\]

\[k := k + \text{SIZE}\]

\[\text{FI;}\]

\[\text{ENDFOR}\]

\[\text{IF } \text{merging-masking}\]

\[\text{THEN } \text{DEST}[VL-1:k] \text{ remains unchanged}\]

\[\text{ELSE } \text{DEST}[VL-1:k] := 0\]

\[\text{FI}\]

\[\text{DEST}[\text{MAXVL}-1:VL] := 0\]

**Intel C/C++ Compiler Intrinsic Equivalent**

- \[\text{VPCOMPRESSQ } \_\_m512i \_\_mm512\_mask\_compress\_epi64(\_\_m512i s, \_\_mmask8 c, \_\_m512i a);\]
- \[\text{VPCOMPRESSQ } \_\_m512i \_\_mm512\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m512i a);\]
- \[\text{VPCOMPRESSQ void } \_\_mm512\_mask\_compress\_store\_epi64(\_\_mmask8 c, \_\_m512i a);\]
- \[\text{VPCOMPRESSQ } \_\_m256i \_\_mm256\_mask\_compress\_epi64(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);\]
- \[\text{VPCOMPRESSQ } \_\_m256i \_\_mm256\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m256i a);\]
- \[\text{VPCOMPRESSQ void } \_\_mm256\_mask\_compress\_store\_epi64(\_\_mmask8 c, \_\_m256i a);\]
- \[\text{VPCOMPRESSQ } \_\_m128i \_\_mm128\_mask\_compress\_epi64(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);\]
- \[\text{VPCOMPRESSQ } \_\_m128i \_\_mm128\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m128i a);\]
- \[\text{VPCOMPRESSQ void } \_\_mm128\_mask\_compress\_store\_epi64(\_\_mmask8 c, \_\_m128i a);\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values Into Dense Memory/ Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

**EVEX.512 encoded version:** The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

**EVEX.256 encoded version:** The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

**EVEX.128 encoded version:** The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VPCONFLICTD**
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j*32
  IF MaskBit(j) OR *no writemask* THEN
    FOR k := 0 TO j-1
      m := k*32
      IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
        DEST[i+k] := 1
      ELSE
        DEST[i+k] := 0
      FI
    ENDFOR
    DEST[i+31:i+j] := 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+31:i] remains unchanged*
    ELSE
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPCONFLICTQ**
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j*64
  IF MaskBit(j) OR *no writemask* THEN
    FOR k := 0 TO j-1
      m := k*64
      IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
        DEST[i+k] := 1
      ELSE
        DEST[i+k] := 0
      FI
    ENDFOR
    DEST[i+63:i+j] := 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+63:i] remains unchanged*
    ELSE
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCONFLICTD __m512i _mm512_conflict_epi32( __m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32( __m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32( __mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64( __m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64( __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64( __mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32( __m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32( __m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32( __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64( __m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64( __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64( __mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32( __m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32( __m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32( __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64( __m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64( __m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64( __mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
## VPDPBUSD—Multiply and Add Unsigned and Signed Bytes

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1<a href="z">k1</a>, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1<a href="z">k1</a>, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 50 /r VPDPBUSD zmm1<a href="z">k1</a>, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Multiples the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

### Operation

**VPDPBUSD dest, src1, src2 (VEX encoded versions)**

VL=(128, 256)

KL=VL/32

ORIGDEST := DEST

FOR i := 0 TO KL-1:

    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])

VPDPBUSD—Multiply and Add Unsigned and Signed Bytes
pword := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
p2word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
p3word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])

DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0

VPDPBUSD dest, src1, src2 (EVEX encoded versions)
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

ELSE IF *zeroing*:
    DEST.dword[i] := 0
ELSE:
    DEST.dword[i] := ORIGDEST.dword[i]

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSD __m128i _mm_dpbusd_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask16, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPDPBUSDS—Multiply and Add Unsigned and Signed Bytes With Saturation

### Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers or 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

### Operation

**VPDPBUSDS** dest, src1, src2 (VEX encoded versions)

\[ \text{VL} = (128, 256) \]

\[ \text{KL} = \text{VL} / 32 \]

\[ \text{ORIGDEST} := \text{DEST} \]

\[ \text{FOR } i := 0 \text{ TO } \text{KL} - 1:\]
INSTRUCTION SET REFERENCE, V

// Extending to 16b
// src1extend := ZERO_EXTEND
// src2extend := SIGN_EXTEND

p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0

VPDPBUSDS dest, src1, src2 (EVEX encoded versions)
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
  IF k1[i] or *no writemask*:
    // Byte elements of SRC1 are zero-extended to 16b and
    // byte elements of SRC2 are sign extended to 16b before multiplication.
    IF SRC2 is memory and EVEX.b == 1:
      t := SRC2.dword[0]
    ELSE:
      t := SRC2.dword[i]
    END IF
    p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) *SIGN_EXTEND(t.byte[3])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
  ELSE IF *zeroing*:
    DEST.dword[i] := 0
  ELSE: // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
  END IF
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPDPBUSDS __m128i _mm_dpbusds_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __m128i, __m128i, __m128i);
VPDPBUSDS __m128i __m128i_mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m256i __m256i_mm_dpbusds_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm_dpbusds_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm_mask_dpbusds_epi32(__m256i, __m256i, __m256i, __m256i);
VPDPBUSDS __m256i __m256i_mm_dpbusds_epi32(__m256i, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i __m512i_mm_dpbusds_avx_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i __m512i_mm_dpbusds_epi32(__m512i, __m512i, __m512i, __m512i);
VPDPBUSDS __m512i __m512i_mm_mask_dpbusds_epi32(__m512i, __m512i, __m512i, __m512i, __m512i);
VPDPBUSDS __m512i __m512i __m512i_mm_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i __m512i __m512i쨈m512i_mm_mask_dpbusds_epi32(__m512i, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPDPWSSD—Multiply and Add Signed Word Integers

Description
Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation
VPDPWSSD dest, src1, src2 (VEX encoded versions)
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
  p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0])
  p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1])
  DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
  DEST[MAX_VL-1:VL] := 0

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 52 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 52 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 52 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM1r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM1r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
VPDPWSSD dest, src1, src2 (EVEX encoded versions)

(KL,VL)=(4,128), (8,256), (16,512)

ORIGDEST := DEST

FOR i := 0 TO KL-1:
  IF k1[i] or *no writemask*:
    IF SRC2 is memory and EVEX.b == 1:
      t := SRC2.dword[0]
    ELSE:
      t := SRC2.dword[i]
    p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
    DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
  ELSE IF *zeroing*:
    DEST.dword[i] := 0
  ELSE:  // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
  DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSD __m128i _mm_dpwssd_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPDPWSSDS—Multiply and Add Signed Word Integers With Saturation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1<a href="z">k1</a>, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1<a href="z">k1</a>, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1<a href="z">k1</a>, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>ModRM:reg (r, w)</th>
<th>VEX.vvvv (r)</th>
<th>ModRM:r/m (r)</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.
INSTRUCTION SET REFERENCE, V

Operation

VPDPwSSDS dest, src1, src2 (VEX encoded versions)

VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:

  p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0])
  p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1])
  DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
DEST[MAX_VL-1:VL] := 0

VPDPwSSDS dest, src1, src2 (EVEX encoded versions)

(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:

  IF k1[i] or *no writemask*:
    IF SRC2 is memory and EVEX.b == 1:
      t := SRC2.dword[0]
    ELSE:
      t := SRC2.dword[i]
    p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
  ELSE IF *zeroing*:
    DEST.dword[i] := 0
  ELSE: // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPDPwSSDS __m128i _mm_dpwssds_avx_epi32(__m128i, __m128i, __m128i);
VPDPwSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPwSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPwSSDS __m128i _mm_maskz_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPwSSDS __m256i _mm256_dpwssds_avx_epi32(__m256i, __m256i, __m256i);
VPDPwSSDS __m256i _mm256_dpwssds_epi32(__m256i, __m256i, __m256i);
VPDPwSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPwSSDS __m256i _mm256_maskz_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPwSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPwSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPwSSDS __m512i _mm512_maskz_dpwssd_epi32(__m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPERM2F128—Permute Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 06 /r ib VPERM2F128 ymm1, ymm2, ymm3/m256, imm8</td>
<td>RV MI</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

![Diagram showing the permutation of fields](image)

**Figure 5-21. VPERM2F128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.
### Operation

**VPERM2F128**
CASE IMM8[1:0] of
0: DEST[127:0] := SRC1[127:0]
2: DEST[127:0] := SRC2[127:0]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] := SRC1[127:0]
ESAC

IF (imm8[3])
DEST[127:0] := 0
FI

IF (imm8[7])
DEST[MAXVL-1:128] := 0
FI

### Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128:  __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)
VPERM2F128:  __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)
VPERM2F128:  __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

#UD If VEX.L = 0
If VEX.W = 1.
VPERM2I128—Permutes Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 /r ib VPERM2I128 ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permutes 128-bit integer data in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Permutes 128-bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

![Figure 5-22. VPERM2I128 Operation](image)

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.
Operation

**VPERM2I128**

CASE IMM8[1:0] of
0: DEST[127:0] := SRC1[127:0]
2: DEST[127:0] := SRC2[127:0]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] := SRC1[127:0]
ESAC

IF (imm8[3])
DEST[127:0] := 0
FI

IF (imm8[7])
DEST[255:128] := 0
FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-23, “Type 6 Class Exception Conditions.”

Additionally:

#UD  
If VEX.L = 0,
If VEX.W = 1.
VPERMB—Permute Packed Bytes Elements

### Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

### Operation

#### VPERMB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

n := 3;

ELSE IF VL = 256:

n := 4;

ELSE IF VL = 512:

n := 5;

FI;

FOR j := 0 TO KL-1:

id := SRC1[j*8 + n : j*8] // location of the source byte

IF k1[j] OR *no writemask* THEN

    DEST[j*8 + 7 : j*8] := SRC2[id*8 + 7 : id*8];

ELSE IF zeroing-masking THEN

    DEST[j*8 + 7 : j*8] := 0;

*ELSE

    DEST[j*8 + 7 : j*8] remains unchanged*

FI

ENDFOR

DEST[MAX_VL-1:VL] := 0;

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

#### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8D /r</td>
<td>A/V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8D /r</td>
<td>A/V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8D /r</td>
<td>A/V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
VPERMB—Permutox packed bytes elements

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMB __m512i _mm512_permutexvar_epi8(__m512i idx, __m512i a);
VPERMB __m512i _mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i _mm512_maskz_permutexvar_epi8(__mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i _mm256_permutexvar_epi8(__m256i idx, __m256i a);
VPERMB __m256i _mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i _mm256_maskz_permutexvar_epi8(__mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i _mm_permutexvar_epi8(__m128i idx, __m128i a);
VPERMB __m128i _mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i _mm_maskz_permutexvar_epi8(__mmask16 k, __m128i idx, __m128i a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMD/VPERMW—Permuted Packed Doubleword/Word Elements

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPERMD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permutes doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permutes doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permutes doublewords in zmm3/m512/m32bcst using indexes in ymm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMD xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permutes word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permutes word integers in ymm3/m256 using indexes in xmm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>C</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permutes word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMrr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.
Operation

VPERMD (EVEX encoded versions)

(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n := 2; Fl;
IF VL = 512 THEN n := 3; Fl;
FOR j := 0 TO KL-1
  i := j * 32
  id := 32*SRC1[i+n:i]
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] := SRC2[31:0];
        ELSE DEST[i+31:i] := SRC2[id+31:id];
      FL;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
  END FOR
DEST[MAXVL-1:VL] := 0

VPERMD (VEX.256 encoded version)

DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0

VPERMW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n := 2; Fl;
IF VL = 256 THEN n := 3; Fl;
IF VL = 512 THEN n := 4; Fl;
FOR j := 0 TO KL-1
  i := j * 16
  id := 16*SRC1[i+n:i]
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC2[id+15:id]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] := 0
    FI
  FI;
END FOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i _mm512_permutexvar_epi32(__m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32(__m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i _mm512_permutexvar_epi16(__m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16(__m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16(__m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded VPERMD, see Table 2-50, "Type E4NF Class Exception Conditions."
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
Additionally:

```c
#UD If VEX.L = 0.
If EVEX.L’L = 0 for VPERMD.
```
### VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 75 /r VPERMI2B xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 75 /r VPERMI2B ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 75 /r VPERMI2B zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Permutates byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.
**Operation**

**VPERMI2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:
    id := 3;
ELSE IF VL = 256:
    id := 4;
ELSE IF VL = 512:
    id := 5;
FI;

TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
    off := 8*SRC1[j*8 + id: j*8];
    IF k1[j] OR *no writemask*:
    ELSE IF *zeroing-masking*
        DEST[j*8 + 7: j*8] := 0;
    ELSE
        DEST[j*8 + 7: j*8] remains unchanged*
    FI;
ENDFOR

DEST[MAX_VL-1:VL] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMI2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMI2B __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b);
VPERMI2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMI2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMI2B __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b);
VPERMI2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMI2B __m128i _mm128_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMI2B __m128i _mm128_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b);
VPERMI2B __m128i _mm128_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."

VPERMI2B—Full Permute of Bytes From Two Tables Overwriting the Index
### VPERM12W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 75 /r VPERM12W xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 75 /r VPERM12W ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 75 /r VPERM12W zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 76 /r VPERM12D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 76 /r VPERM12D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 76 /r VPERM12D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 76 /r VPERM12Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 76 /r VPERM12Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 76 /r VPERM12Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 77 /r VPERM12PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 77 /r VPERM12PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 77 /r VPERM12PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

### Description

Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands. If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0]/[0] selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (r,w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Opcode/Instruction

- **EVEX.128.66.0F38.W1 77 /r**
  - VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst
  - Description: Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.

- **EVEX.256.66.0F38.W1 77 /r**
  - VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst
  - Description: Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.

- **EVEX.512.66.0F38.W1 77 /r**
  - VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst
  - Description: Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
**Operation**

**VPERM2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
   id := 2
   Fl;
IF VL = 256
   id := 3
   Fl;
IF VL = 512
   id := 4
   Fl;
   TMP_DEST := DEST
   FOR j := 0 TO KL-1
      i := j * 16
      off := 16 * TMP_DEST[i:id]
      IF k1[j] OR *no writemask*
         THEN
            : SRC1[off+15:off]
         ELSE
            IF *merging-masking* ; merging-masking
               THEN *DEST[i+15:i] remains unchanged*
               ELSE ; zeroing-masking
                  DEST[i+15:i] := 0
         FI
      FI;
   ENDFOR
   DEST[MAXVL-1:VL] := 0

**VPERM2D/VPERM2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128
   id := 1
   Fl;
IF VL = 256
   id := 2
   Fl;
IF VL = 512
   id := 3
   Fl;
   TMP_DEST := DEST
   FOR j := 0 TO KL-1
      i := j * 32
      off := 32 * TMP_DEST[i:id]
      IF k1[j] OR *no writemask*
         THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN
                  DEST[i+31:i] := TMP_DEST[i+id+1] ? SRC2[31:0]
                  : SRC1[off+31:off]
               ELSE
                  : SRC1[off+31:off]
         ELSE
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
               THEN
                  DEST[i+31:i] := TMP_DEST[i+id+1] ? SRC2[31:0]
                  : SRC1[off+31:off]
               ELSE
                  : SRC1[off+31:off]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
  FI
ENDIF
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMI2Q/VPERMI2PD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
  id := 0
ELSE
  IF VL = 256
    id := 1
ELSE
    IF VL = 512
      id := 2
    FI
    TMP_DEST := DEST
    FOR j := 0 TO KL-1
      i := j * 64
      off := 64*TMP_DEST[i+id:i]
      IF k1[j] OR *no writemask*
        THEN
          IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                            : SRC1[off+63:off]
            ELSE
                            : SRC1[off+63:off]
          FI
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+63:i] := 0
          FI
        FI
    ENDFOR
    DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2W __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERM2I __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERM2I __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERM2I __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERM2I __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERM2I __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERM2I __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2I __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERM2I __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERM2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERM2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERM2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERM2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERM2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERM2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERM2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMI2D/Q/PS/PD: See Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMI2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute double precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute double precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 0D /r VPERMILPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 0D /r VPERMILPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 0D /r VPERMILPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 05 /i b VPERMILPD xmm1, xmm2/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute double precision floating-point values in xmm2/m128 using controls from imm8.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 05 /i b VPERMILPD ymm1, ymm2/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute double precision floating-point values in ymm2/m256 using controls from imm8.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 05 /i b VPERMILPD xmm1 [k1]{z}, xmm2/m128/m64bcst, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 05 /i b VPERMILPD ymm1 [k1]{z}, ymm2/m256/m64bcst, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 05 /i b VPERMILPD zmm1 [k1]{z}, zmm2/m512/m64bcst, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description
(variable control version)

Permute pairs of double precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permutated results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permutated results are written to the destination under the writemask.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

Immediate control version: Permute pairs of double precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permutated results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

VPERMILPD (EVEX immediate versions)

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF (EVEX.b = 1) AND (SRC1 *is memory*)

THEN TMP_SRC1[i+63:i] := SRC1[63:0];
ELSE TMP_SRC1[i+63:i] := SRC1[i+63:i];
FI;

ENDFOR;

IF (imm8[0] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] := TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] := TMP_SRC1[63:0]; FI;

IF VL >= 256


FI;

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] := TMP_DEST[i+63:i]
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VPERMILPD (256-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0] FI;
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64] FI;
DEST[MAXVL-1:256] := 0
VPERMILPD (128-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
DEST[MAXVL-1:128] := 0

VPERMILPD (EVEX variable versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
       THEN TMP_SRC2[i+63:i] := SRC2[63:0];
       ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] := SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] := SRC1[127:64]; FI;
IF VL >= 256
FI;
IF VL >= 512
    IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] := SRC1[319:256]; FI;
    IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] := SRC1[447:384]; FI;
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
       THEN DEST[i+63:i] := TMP_DEST[i+63:i]
       ELSE
           IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
           ELSE ; zeroing-masking
               DEST[i+63:i] := 0
           FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPERMILPD (256-bit variable version)
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:256] := 0

VPERMILPD (128-bit variable version)
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMILPD __m512d _mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd(__mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_mask_permute_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_maskz_permute_pd(__mmask8 k, __m256d a, int imm);
VPERMILPD __m128d _mm_mask_permute_pd(__m128d a, int imm);
VPERMILPD __m128d _mm_maskz_permute_pd(__mmask8 k, __m128d a, int imm);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Additionally:
#UD If VEX.W = 1.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
Additionally:
#UD If either (E)VEX.vvvv != 1111B and with imm8.
### VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 0C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.</td>
</tr>
<tr>
<td>VPERMILPS xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 04 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.</td>
</tr>
<tr>
<td>VPERMILPS xmm1, xmm2/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.</td>
</tr>
<tr>
<td>VPERMILPS ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 04 /r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.</td>
</tr>
<tr>
<td>VPERMILPS ymm1, ymm2/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 0C /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 0C /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 0C /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 04 /r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 04 /r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 04 /r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
<tr>
<td>D</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
**Description**

Variable control version:

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

(Immediate control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

\[
\text{Select4}(\text{SRC}, \text{control}) \{
\text{CASE (control}[1:0]) \text{ OF}
\]
\[
0: \quad \text{TMP} := \text{SRC}[31:0];
1: \quad \text{TMP} := \text{SRC}[63:32];
2: \quad \text{TMP} := \text{SRC}[95:64];
3: \quad \text{TMP} := \text{SRC}[127:96];
\text{ESAC;}
\]
\[
\text{RETURN TMP}
\}
\]

**VPERMILPS (EVEX immediate versions)**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

**FOR** \(j := 0 \text{ TO } KL-1\)

\[
i := j \times 32
\]
\[
\text{IF} (\text{EVEX}.b = 1 \text{ AND (SRC1 *is memory*)})
\]
\[
\text{THEN} \quad \text{TMP\_SRC1}[i+31:i] := \text{SRC1}[31:0];
\]
\[
\text{ELSE} \quad \text{TMP\_SRC1}[i+31:i] := \text{SRC1}[i+31:i];
\]
\[
\text{FI;}
\]
\[
\text{ENDIF;}
\]
\[
\text{TMP\_DEST}[31:0] := \text{Select4(TMP\_SRC1[127:0], imm8}[1:0]);
\]
\[
\text{TMP\_DEST}[63:32] := \text{Select4(TMP\_SRC1[127:0], imm8}[3:2]);
\]
\[
\text{TMP\_DEST}[95:64] := \text{Select4(TMP\_SRC1[127:0], imm8}[5:4]);
\]
\[
\text{TMP\_DEST}[127:96] := \text{Select4(TMP\_SRC1[127:0], imm8}[7:6]);
\]
\[
\text{FI;}
\]
\[
\text{IF} \quad VL \geq 256
\]
\[
\text{TMP\_DEST}[159:128] := \text{Select4(TMP\_SRC1[255:128], imm8}[1:0]);
\]
\[
\text{TMP\_DEST}[191:160] := \text{Select4(TMP\_SRC1[255:128], imm8}[3:2]);
\]
\[
\text{TMP\_DEST}[223:192] := \text{Select4(TMP\_SRC1[255:128], imm8}[5:4]);
\]
\[
\text{TMP\_DEST}[255:224] := \text{Select4(TMP\_SRC1[255:128], imm8}[7:6]);
\]
\[
\text{FI;}
\]
\[
\text{IF} \quad VL \geq 512
\]
\[
\text{TMP\_DEST}[287:256] := \text{Select4(TMP\_SRC1[383:256], imm8}[1:0]);
\]
\[
\text{TMP\_DEST}[319:288] := \text{Select4(TMP\_SRC1[383:256], imm8}[3:2]);
\]
\[
\text{TMP\_DEST}[351:320] := \text{Select4(TMP\_SRC1[383:256], imm8}[5:4]);
\]
\[
\text{TMP\_DEST}[383:352] := \text{Select4(TMP\_SRC1[383:256], imm8}[7:6]);
\]
\[
\text{TMP\_DEST}[415:384] := \text{Select4(TMP\_SRC1[511:384], imm8}[1:0]);
\]
\[
\text{TMP\_DEST}[447:416] := \text{Select4(TMP\_SRC1[511:384], imm8}[3:2]);
\]
\[
\text{TMP\_DEST}[479:448] := \text{Select4(TMP\_SRC1[511:384], imm8}[5:4]);
\]
\[
\text{TMP\_DEST}[511:480] := \text{Select4(TMP\_SRC1[511:384], imm8}[7:6]);
\]
\[
\text{FI;}
\]
\[
\text{FOR} \quad j := 0 \text{ TO } KL-1
\]
\[
i := j \times 32
\]
\[
\text{IF} \quad \text{k1}[j] \text{ OR *no writemask*}
\]
\[
\text{THEN} \quad \text{DEST}[i+31:i] := \text{TMP\_DEST}[i+31:i]
\]
\[
\text{ELSE}
\]
\[
\text{IF} \quad *\text{merging-masking*}
\]
\[
\text{THEN} \quad *\text{DEST}[i+31:i] \text{ remains unchanged*}
\]
\[
\text{ELSE} \quad \text{DEST}[i+31:i] := 0 \quad \text{;zeroing-masking}
\]
\[
\text{FI;}
\]
\[
\text{FI;}
\]
\[
\text{ENDIF}
\]
\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]
VPERMILPS (256-bit immediate version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);

VPERMILPS (128-bit immediate version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0

VPERMILPS (EVEX variable versions)
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0];
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
  FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] := Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] := Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
FI;
IF VL >= 512
  TMP_DEST[479:448] := Select4(SRC1[511:384], TMP_SRC2[449:448]);
  TMP_DEST[511:480] := Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] := 0 ;zeroing-masking
  FI;
ENDFOR;

VPERMILPS — Permute In-Lane of Quadruples of Single Precision Floating-Point Values

\begin{verbatim}
                        Fl;
                    ENDFOR
                    DEST[MAXVL-1:VL] := 0

VPERMILPS (256-bit variable version)

DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:256] := 0

VPERMILPS (128-bit variable version)

DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS __m512 __mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __mm512_mask_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __mm256_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_mask_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __mm128_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm128_mask_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm128_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm128_permutevar_ps( __m128 i, __m128 a);
VPERMILPS __m128 __mm128_mask_permutevar_ps( __m128 s, __mmask16 k, __m128 i, __m128 a);
VPERMILPS __m128 __mm128_maskz_permutevar_ps( __mmask16 k, __m128 i, __m128 a);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __mm256_mask_permutevar_ps( __m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 __mm128_permute_ps( __m128 a, int control);
VPERMILPS __m256 __mm256_permute_ps( __m256 a, int control);
VPERMILPS __m128 __mm128_permutevar_ps( __m128 a, __m128i control);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 a, __m256i control);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."
Additionally:

#UD If VEX.W = 1.

EVEX-encoded instruction, see Table 2-50, "Type E4NF Class Exception Conditions."
Additionally:

#UD If either (E)VEX.vvvv != 1111B and with imm8.

VPERMILPS—Permute In-Lane of Quadruples of Single Precision Floating-Point Values

Vol. 2C 5-493
VPERMPD—Permutes Double Precision Floating-Point Elements

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W1 01 / r ib VPERMPD ymm1, ymm2/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permutes double precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 01 / r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permutes double precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 01 / r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permutes double precision floating-point elements in zmm2/m512/m64bcst using indexes in imm8 and store the result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 16 / r VPERMPD ymm1 {k1}{z}, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permutes double precision floating-point elements in zmm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 16 / r VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permutes double precision floating-point elements in zmm3/m512/m64bcst using indexes in zmm2 and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (w)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The imm8 version: Copies quadword elements of double precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a quadword element in the source operand.

**VEX version:** The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a quadword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Operation

**VPERMPD (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
     THEN TMP_SRC[i+63:i] := SRC[63:0];
     ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
  FI;
ENDFOR;

TMP_DEST[63:0] := (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[255:192] := (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];

IF VL >= 512
  TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
FI;

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
     THEN DEST[i+63:i] := TMP_DEST[i+63:i];
     ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0 ;zeroing-masking
         FI;
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPERMPD (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
     THEN TMP_SRC2[i+63:i] := SRC2[63:0];
     ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;

IF VL = 512
  TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
VPERMPD—Permute Double Precision Floating-Point Elements

\[
\begin{align*}
\text{TMP\_DEST}[127:64] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[66:64] * 64))[63:0]; \\
\text{TMP\_DEST}[191:128] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[130:128] * 64))[63:0]; \\
\text{TMP\_DEST}[255:192] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[194:192] * 64))[63:0]; \\
\text{TMP\_DEST}[319:256] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[258:256] * 64))[63:0]; \\
\text{TMP\_DEST}[383:320] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[322:320] * 64))[63:0]; \\
\text{TMP\_DEST}[447:384] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[386:384] * 64))[63:0]; \\
\text{TMP\_DEST}[511:448] &:= (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[450:448] * 64))[63:0];
\end{align*}
\]

Fi;

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]
\[
i := j \times 64
\]
\[
\text{IF } k1[j] \text{ OR *no writemask*}
\]
\[
\text{THEN } \text{DEST}[i+63:i] := \text{TMP\_DEST}[i+63:i]
\]
\[
\text{ELSE}
\]
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged*}
\]
\[
\text{ELSE ; zeroing-masking}
\]
\[
\text{DEST}[i+63:i] := 0
\]
\[
\text{Fl;}
\]
\[
\text{Fl;}
\]
\[
\text{ENDFOR}
\]
\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VPERMPD (VEX.256 encoded version)
\[
\text{DEST}[63:0] := (\text{SRC}[255:0] >> (\text{IMM8}[1:0] * 64))[63:0];
\]
\[
\text{DEST}[127:64] := (\text{SRC}[255:0] >> (\text{IMM8}[3:2] * 64))[63:0];
\]
\[
\text{DEST}[191:128] := (\text{SRC}[255:0] >> (\text{IMM8}[5:4] * 64))[63:0];
\]
\[
\text{DEST}[255:192] := (\text{SRC}[255:0] >> (\text{IMM8}[7:6] * 64))[63:0];
\]
\[
\text{DEST}[\text{MAXVL}-1:256] := 0
\]

Intel C/C++ Compiler Intrinsic Equivalent
\[
\begin{align*}
\text{VPERMPD } \_m512d \_mm512\_permutex\_pd(\_m512d \_a, \text{int \_imm}); \\
\text{VPERMPD } \_m512d \_mm512\_maskz\_permutex\_pd(\_mmask16 \_k, \_m512d \_a, \text{int \_imm}); \\
\text{VPERMPD } \_m512d \_mm512\_maskz\_permutex\_var\_pd(\_mm512i \_i, \_m512d \_a); \\
\text{VPERMPD } \_m512d \_mm512\_maskz\_permutex\_var\_pd(\_mmask16 \_k, \_m512d \_a, \text{int \_imm}); \\
\text{VPERMPD } \_m256d \_mm256\_permutex\_epi64(\_m256d \_a, \text{int \_imm}); \\
\text{VPERMPD } \_m256d \_mm256\_maskz\_permutex\_epi64(\_m256d \_a, \text{int \_imm});
\end{align*}
\]

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”; additionally:

#UD
\[
\begin{align*}
\text{If } \text{VEX.L} = 0. \\
\text{If } \text{VEX.vvvv} != 1111B.
\end{align*}
\]

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”; additionally:

#UD
\[
\begin{align*}
\text{If encoded with } \text{EVEX.128.} \\
\text{If } \text{EVEX.vvvv} != 1111B \text{ and with } \text{imm8}.
\end{align*}
\]
VPERMPS—Permute Single Precision Floating-Point Elements

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256:66.0F38.W0 16 /r VPERMPS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256:66.0F38.W0 16 /r VPERMPS ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.512:66.0F38.W0 16 /r VPERMPS zmm1 [k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values in zmm3/m512/m32bcst using indexes in zmm2 and store the result in zmm1 subject to write mask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation**

VPERMPS (EVEX forms)

(KL, VL) (8, 256),= (16, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0];
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[31:0] := (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
  TMP_DEST[63:32] := (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
  TMP_DEST[95:64] := (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
  TMP_DEST[127:96] := (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
VPERMPS—Permute Single Precision Floating-Point Elements

\[
\text{TMP\_DEST}[255:224] := (\text{TMP\_SRC2}[255:0] >> (\text{SRC1}[226:224] * 32))[31:0];
\]

\[\text{FI;}\]

\[\text{IF } \text{VL} = 512\]

\[
\begin{align*}
\text{TMP\_DEST}[31:0] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[3:0] * 32))[31:0]; \\
\text{TMP\_DEST}[63:32] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[34:32] * 32))[31:0]; \\
\text{TMP\_DEST}[95:64] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[66:64] * 32))[31:0]; \\
\text{TMP\_DEST}[127:96] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[98:96] * 32))[31:0]; \\
\text{TMP\_DEST}[159:128] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[130:128] * 32))[31:0]; \\
\text{TMP\_DEST}[191:160] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[162:160] * 32))[31:0]; \\
\text{TMP\_DEST}[223:192] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[194:192] * 32))[31:0]; \\
\text{TMP\_DEST}[255:224] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[226:224] * 32))[31:0]; \\
\text{TMP\_DEST}[287:256] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[258:256] * 32))[31:0]; \\
\text{TMP\_DEST}[319:288] & := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[290:288] * 32))[31:0]; \\
\end{align*}
\]

\[\text{FI;}\]

\[\text{FOR } j := 0 \text{ TO } \text{KL}-1 \]

\[
i := j * 32
\]

\[\text{IF } \text{k1}[j] \text{ OR } \text{no writemask}\]

\[
\begin{align*}
\text{THEN } & \text{DEST}[i+31:i] := \text{TMP\_DEST}[i+31:i] \\
\text{ELSE} & \\
\text{IF } & \text{merging-masking} \text{ THEN } \text{DEST}[i+31:i] \text{ remains unchanged}\text{ ELSE zeroing-masking ENDFOR; }
\text{DEST}[i+31:i] := 0
\end{align*}
\]

\[\text{FI;}\]

\[\text{ENDFOR} \]

\[\text{DEST}[\text{MAXVL}-1:256] := 0\]

\[\text{VPERMPS (VEX.256 encoded version)}\]

\[
\begin{align*}
\text{DEST}[31:0] & := (\text{SRC2}[255:0] >> (\text{SRC1}[20:0] * 32))[31:0]; \\
\text{DEST}[63:32] & := (\text{SRC2}[255:0] >> (\text{SRC1}[34:32] * 32))[31:0]; \\
\text{DEST}[95:64] & := (\text{SRC2}[255:0] >> (\text{SRC1}[66:64] * 32))[31:0]; \\
\text{DEST}[127:96] & := (\text{SRC2}[255:0] >> (\text{SRC1}[98:96] * 32))[31:0]; \\
\text{DEST}[159:128] & := (\text{SRC2}[255:0] >> (\text{SRC1}[130:128] * 32))[31:0]; \\
\text{DEST}[191:160] & := (\text{SRC2}[255:0] >> (\text{SRC1}[162:160] * 32))[31:0]; \\
\text{DEST}[223:192] & := (\text{SRC2}[255:0] >> (\text{SRC1}[194:192] * 32))[31:0]; \\
\text{DEST}[255:224] & := (\text{SRC2}[255:0] >> (\text{SRC1}[226:224] * 32))[31:0]; \\
\text{DEST}[\text{MAXVL}-1:256] := 0
\end{align*}
\]

\[\text{Intel C/C++ Compiler Intrinsic Equivalent}\]

\[
\begin{align*}
\text{VPERMPS\_m512\_mm512\_permutevar\_ps}(\_m512i, \_m512 a); \\
\text{VPERMPS\_m512\_mm512\_mask\_permutevar\_ps}(\_m512 s, \_mmsk16 k, \_m512i, \_m512 a); \\
\text{VPERMPS\_m512\_mm512\_maskz\_permutevar\_ps}(\_mmsk16 k, \_m512i, \_m512 a); \\
\text{VPERMPS\_m256\_mm256\_permutevar\_ps}(\_m256 i, \_m256 a); \\
\text{VPERMPS\_m256\_mm256\_mask\_permutevar\_ps}(\_m256 s, \_mmsk8 k, \_m256 i, \_m256 a); \\
\text{VPERMPS\_m256\_mm256\_maskz\_permutevar\_ps}(\_mmsk8 k, \_m256 i, \_m256 a);
\end{align*}
\]
**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Additionally:

#UD If VEX.L = 0.
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMQ—Qwords Element Permutation

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM[reg (w)]</td>
<td>ModRM[r/m (r)]</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM[reg (w)]</td>
<td>ModRM[r/m (r)]</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM[reg (w)]</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM[r/m (r)]</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which qword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.
**Operation**

**VPERMQ (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] := SRC[63:0];
        ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
    FI;
ENDFOR;

TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];

IF VL >= 512
    TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*  
            THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPERMQ (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;

IF VL = 512
    TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
VPERMQ (VEX.256 encoded version)
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
Additionally:
#UD If VEX.L = 0.
    If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”
Additionally:
#UD If encoded with EVEX.128.
    If VEX.vvvv != 1111B and with imm8.
VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table

### Opcode/Instruction
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 7D /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7D /r</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7D /r</td>
<td>A V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX:vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Permutates byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.
Operation

**VPERMT2B (EVEX encoded versions)**

\((KL, VL) = (16, 128), (32, 256), (64, 512)\)

IF \(VL = 128\):

\[\text{id} := 3;\]

ELSE IF \(VL = 256\):

\[\text{id} := 4;\]

ELSE IF \(VL = 512\):

\[\text{id} := 5;\]

\[\text{FI;}\]

\[\text{TMP\_DEST}[VL-1:0] := DEST[VL-1:0];\]

FOR \(j := 0 \text{ TO } KL-1\)

\[\text{off} := 8*SRC1[^{8} + \text{id};^{8}];\]

IF \(k1[j] \text{ OR } \text{no writemask}\):

\[\text{DEST}[^{8} + 7;^{8}] := SRC1[^{8}+\text{id}+1]; \text{SRC2}[off+7:off] : TMP\_DEST[off+7:off];\]

ELSE IF \(\text{zeroing-masking}\)

\[\text{DEST}[^{8} + 7;^{8}] := 0;\]

*ELSE

\[\text{DEST}[^{8} + 7;^{8}] \text{ remains unchanged};\]

\[\text{FI;}\]

\[\text{ENDFOR}\]

\[\text{DEST}[\text{MAX\_VL-1:VL}] := 0;\]

**Intel C/C++ Compiler Intrinsic Equivalent**

\[\text{VPERMT2B} __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);\]

\[\text{VPERMT2B} __m512i _mm512_mask_permutex2var_epi8(__m512i a, __mmask64 k, __m512i idx, __m512i b);\]

\[\text{VPERMT2B} __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);\]

\[\text{VPERMT2B} __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);\]

\[\text{VPERMT2B} __m256i _mm256_mask_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);\]

\[\text{VPERMT2B} __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);\]

\[\text{VPERMT2B} __m128i _mm128_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);\]

\[\text{VPERMT2B} __m128i _mm128_mask_permutex2var_epi8(__m128i a, __mmask16 k, __m128i idx, __m128i b);\]

\[\text{VPERMT2B} __m128i _mm128_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);\]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."
## VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 7D / r VPERMT2W xmm1[k1][z], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7D / r VPERMT2W ymm1[k1][z], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7D / r VPERMT2W zmm1[k1][z], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7E / r VPERMT2D xmm1[k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7E / r VPERMT2D ymm1[k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7E / r VPERMT2D zmm1[k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7E / r VPERMT2Q xmm1[k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7E / r VPERMT2Q ymm1[k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7E / r VPERMT2Q zmm1[k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7F / r VPERMT2PS xmm1[k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7F / r VPERMT2PS ymm1[k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7F / r VPERMT2PS zmm1[k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7F / r VPERMT2PD xmm1[k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7F / r VPERMT2PD ymm1[k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7F / r VPERMT2PD zmm1[k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
**Description**

Permutates 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands. If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation**

**VPERMT2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128
   id := 2
FI;

IF VL = 256
   id := 3
FI;

IF VL = 512
   id := 4
FI;

TMP_DEST := DEST
FOR j := 0 TO KL-1
   i := j * 16
   off := 16*SRC1[i+id]
   IF k1[j] OR *no writemask*
      THEN
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:j] remains unchanged*;
         ELSE ; zeroing-masking


---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRMreg (r,w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
DEST[i+15:i] := 0
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMT2D/VPERMT2PS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id := 1
FI;
IF VL = 256
    id := 2
FI;
IF VL = 512
    id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 32
    off := 32*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] := SRC1[i+id+1] ? SRC2[31:0]
                    : TMP_DEST[off+31:off]
                ELSE
                    : TMP_DEST[off+31:off]
                FI
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
            ELSE
                DEST[i+31:i] := 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMT2Q/VPERMT2PD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id := 0
FI;
IF VL = 256
    id := 1
FI;
IF VL = 512
    id := 2
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

i := j * 64
off := 64 * SRC1[i+idi]
IF k1[i] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
          : TMP_DEST[off+63:off]
      ELSE
          : TMP_DEST[off+63:off]
    FI
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI
ENDIF
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2D __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i id, __m128i b);
VPERMT2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i id, __m128i b);
VPERMT2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i id, __mmask8 k, __m128i b);
VPERMT2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i id, __m128i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512d idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512d idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512d idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512d id, __m512d b);
VPERMT2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256d idx, __m256d b);
VPERMT2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256d idx, __m256d b);
VPERMT2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256d idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256d id, __m256d b);
VPERMT2PD __m128d _mm_permutex2var_pd(__m128d a, __m128d id, __m128d b);
VPERMT2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128d id, __m128d b);
VPERMT2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128d id, __mmask8 k, __m128d b);
VPERMT2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128d id, __m128d b);
VPERMT2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i id, __m512 b);
VPERMT2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i id, __m512 b);
VPERMT2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i id, __mmask16 k, __m512 b);
VPERMT2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i id, __m512 b);
VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

INSTRUCTION SET REFERENCE, V

Vol. 2C 5-509

VPERMT2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 __m256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 __m256_mask2_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 __m256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 __m128_mask_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 __m128_mask2_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 __m128_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMT2Q __m512i __m512i_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i __m512i_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i __m512i_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i __m512i_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m256i __m256i_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMT2Q __m256i __m256i_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2Q __m256i __m256i_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2Q __m256i __m256i_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2Q __m128i __m128i_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i __m128i_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i __m128i_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2Q __m128i __m128i_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMT2D/Q/PS/PD: See Table 2-50, “Type E4NF Class Exception Conditions.”
VPERMT2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions.”
## VPEXPANDB/VPEXPANDW—Expand Byte/Word Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Expands up to 512 bits of packed word integer values from zmm2 to zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the writemask operand.

Note: EVEX.vvv is reserved and must be 111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.
Operation

VPEXPANDB

\((KL, VL) = (16, 128), (32, 256), (64, 512)\)

\(k := 0\)

FOR \(j := 0\) TO \(KL - 1\):
  IF \(k1[j]\) OR *no writemask*:
    DEST.byte[j] := SRC.byte[k];
    \(k := k + 1\)
  ELSE:
    IF *merging-masking*:
      *DEST.byte[j] remains unchanged*
    ELSE:
      *zeroing-masking
      DEST.byte[j] := 0

DEST[MAX_VL-1:VL] := 0

VPEXPANDW

\((KL, VL) = (8,128), (16,256), (32, 512)\)

\(k := 0\)

FOR \(j := 0\) TO \(KL - 1\):
  IF \(k1[j]\) OR *no writemask*:
    DEST.word[j] := SRC.word[k];
    \(k := k + 1\)
  ELSE:
    IF *merging-masking*:
      *DEST.word[j] remains unchanged*
    ELSE:
      *zeroing-masking
      DEST.word[j] := 0

DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i _mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i _mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i _mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i _mm_maskz_expandw_epi16(__m128i, __m128i);
VPEXPANDW __m128i _mm_maskz_expandloaduw_epi16(__m128i, const void*);
VPEXPANDW __m256i _mm256_maskz_expandw_epi16(__m256i, __m256i);
VPEXPANDW __m256i _mm256_maskz_expandloaduw_epi16(__m256i, const void*);
VPEXPANDW __m512i _mm512_maskz_expandw_epi16(__m512i, __m512i);
VPEXPANDW __m512i _mm512_maskz_expandloaduw_epi16(__m512i, const void*);
**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-49, “Type E4 Class Exception Conditions.”
VPCTSTRDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VPCTSTRDD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

k := 0

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+31:i] := SRC[k+31:k];
      k := k + 32
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged* ; zeroing-masking
          ELSE
            DEST[i+31:i] := 0
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VPCTSTRDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i __mm512_mask_exploadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i __mm512_maskz_exploadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i __mm512_mask_exp_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i __mm512_maskz_exp_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i __mm256_mask_exploadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i __mm256_maskz_exploadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i __mm256_mask_exp_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i __mm256_maskz_exp_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i __mm128_mask_exploadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i __mm128_maskz_exploadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i __mm128_mask_exp_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i __mm128_maskz_exp_epi32(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."
Additionally:

#UD If EVEX.vvvv != 1111B.
VPEXPANDQ—Load Sparse Packed Quadword Integer Values From Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 89 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPEXPANDQ xmm1 [k1][z], xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 89 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPEXPANDQ ymm1 [k1][z], ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 89 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPEXPANDQ zmm1 [k1][z], zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

k := 0

FOR j := 0 TO KL-1

  i := j * 64

  IF k1[j] OR *no writemask*

    THEN

    DEST[i+63:j] := SRC[k+63:k];

    k := k + 64

  ELSE

    IF *merging-masking*

    THEN *DEST[i+63:j] remains unchanged*

    ELSE

    THEN DEST[i+63:j] := 0

  FI

ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_mask_exploado_ep164(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_exploado_ep164(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_mask_exp_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_exp_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_exploado_ep164(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_exploado_ep164(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_exp_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_exp_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_exploado_ep164(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_exploado_ep164(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_mask_exp_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_exp_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.
VPGATHERDD/VPGATHERQD—Gather Packed Dword Values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 90 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 91 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 90 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 91 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX:vvv (r, w)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:
If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.

The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.

Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

This instruction does not perform AC checks, and so will never deliver an AC fault.

This instruction will cause a #UD if the address size attribute is 16-bit.

This instruction will cause a #UD if the memory operand is encoded without the SIB byte.

This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

```plaintext
DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VPGATHERDD (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 3
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i+31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i+31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[i+31:j])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i+31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i+31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
```
VPGATHERQD (VEX.128 version)
MASK[MAXVL-1:64] := 0;
FOR j := 0 to 3
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  k := j * 64;
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITs(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:64] := 0;

VPGATHERDD (VEX.256 version)
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 7
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITs(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
VPGATHERQD (VEX.256 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  k := j * 64;
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;

Intel C/C++ Compiler Intrinsic Equivalent
VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDD: __m256i _mm256_i32gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);
VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQD: __m256i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERQD: __m256i _mm256_mask_i64gather_epi32 (__m256i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-27, "Type 12 Class Exception Conditions.”
VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword With Signed Dword Indices

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 [k1], vm32x</td>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 [k1], vm32y</td>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 [k1], vm32z</td>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion. Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1). The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

**VPGATHERDD (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j]
    THEN DEST[i+31:i] := MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
      k1[j] := 0
    ELSE *DEST[i+31:i] := remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0

**VPGATHERDQ (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j]
    THEN DEST[i+63:i] :=
      MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
      k1[j] := 0
    ELSE *DEST[i+63:i] := remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i __mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i __mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i __mm256_mmask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i __mm128i_mask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i __mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i __mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERDQ __m256i __mm256_mmask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i __mm128i_mask_i32logather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-61, "Type E12 Class Exception Conditions."
VPGATHERDQ/VPGATHERQQ—Gather Packed Qword Values Using Signed Dword/Qword Indices

**Description**

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

---

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 90/r</td>
<td>VPGATHERDQ xmm1, vm32x, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 91/r</td>
<td>VPGATHERQQ xmm1, vm64x, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 90/r</td>
<td>VPGATHERDQ ymm1, vm32x, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 91/r</td>
<td>VPGATHERQQ ymm1, vm64y, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>
VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

**VPGATHERDQ (VEX.128 version)**

MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  k := j * 32;
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP);
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
VPGATHERQQ (VEX.128 version)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP);
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
  MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;

VPGATHERQQ (VEX.256 version)
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP);
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
  MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;

VPGATHERDQ (VEX.256 version)
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 3
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  i := j * 32;
  k := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP);
IF MASK[63+i] THEN
  DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

- `VPGATHERDQ: __m128i _mm_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);`
- `VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);`
- `VPGATHERDQ: __m256i _mm256_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);`
- `VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, __int64 const * base, __m128i index, __m256i mask, const int scale);`
- `VPGATHERQQ: __m128i _mm_i64gather_epi64 (__int64 const * base, __m128i index, const int scale);`
- `VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);`
- `VPGATHERQQ: __m256i _mm256_i64gather_epi64 (__int64 const * base, __m256i index, const int scale);`
- `VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, __int64 const * base, __m256i index, __m256i mask, const int scale);`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-27, "Type 12 Class Exception Conditions.”

---

**INSTRUCTION SET REFERENCE, V**
VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x</td>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64y</td>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD zmm1 {k1}, vm64z</td>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description
A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.
• These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).
The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.
The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

**VPGATHERQD (EVEX encoded version)**

KL, VL = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j]
    THEN DEST[i+31:i] := MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+31:i] := remains unchanged* ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0

**VPGATHERQQ (EVEX encoded version)**

KL, VL = (2, 64), (4, 128), (8, 256)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j]
    THEN DEST[i+63:i] :=
        MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+63:i] := remains unchanged* ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQD __m512i _mm512_i64gather_epi64( __m512i vdx, void * base, int scale);
VPGATHERQD __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m256i _mm_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-61, “Type E12 Class Exception Conditions.”
VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst</td>
<td>A V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

**VPLZCNTD**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 \) TO \(KL-1\)

\[ i := j * 32 \]

IF MaskBit\((j)\) OR *no writemask*

THEN

\[ \text{temp := 32} \]
\[ \text{DEST}[i+31:i] := 0 \]
\[ \text{WHILE (temp > 0) AND (SRC}[i+\text{temp-1}] = 0) \]
\[ \text{DO} \]
\[ \text{temp := temp - 1} \]
\[ \text{DEST}[i+31:i] := \text{DEST}[i+31:i] + 1 \]
\[ \text{OD} \]
ELSE

IF *merging-masking*

THEN *DEST*[i+31:i] remains unchanged*

ELSE \(\text{DEST}[i+31:i] := 0\)

FI

ENDIF

\[ \text{DEST}[\text{MAXVL-1:VL}] := 0 \]

**VPLZCNTQ**

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)

\[ i := j * 64 \]

IF MaskBit\((j)\) OR *no writemask*

THEN

\[ \text{temp := 64} \]
\[ \text{DEST}[i+63:i] := 0 \]
\[ \text{WHILE (temp > 0) AND (SRC}[i+\text{temp-1}] = 0) \]
\[ \text{DO} \]
\[ \text{temp := temp - 1} \]
\[ \text{DEST}[i+63:i] := \text{DEST}[i+63:i] + 1 \]
\[ \text{OD} \]
ELSE

IF *merging-masking*

THEN *DEST*[i+63:i] remains unchanged*

ELSE \(\text{DEST}[i+63:i] := 0\)

FI

ENDIF

\[ \text{DEST}[\text{MAXVL-1:VL}] := 0 \]
Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions
None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers and Add High 52-Bit Products to 64-Bit Accumulators

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA</td>
<td>Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m(r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.
Operation

VPMADD52HUQ (EVEX encoded)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0\) TO \(KL-1\)

\(i := j * 64;\)

IF \(k1[j]\) OR *no writemask* THEN

IF src2 is Memory AND EVEX.b=1 THEN

\(\text{tsrc2}[63:0] := \text{ZeroExtend64}(\text{src2}[51:0]);\)

ELSE

\(\text{tsrc2}[63:0] := \text{ZeroExtend64}(\text{src2}[i+51:i]);\)

FI;

\(\text{Temp128}[127:0] := \text{ZeroExtend64}(\text{src1}[i+51:i]) \times \text{tsrc2}[63:0];\)

\(\text{Temp2}[63:0] := \text{DEST}[i+63:i] + \text{ZeroExtend64}(\text{temp128}[103:52]);\)

\(\text{DEST}[i+63:i] := \text{Temp2}[63:0];\)

ELSE

IF *zeroing-masking* THEN

\(\text{DEST}[i+63:i] := 0;\)

ELSE *merge-masking*

\(\text{DEST}[i+63:i]\) is unchanged;

FI;

FI;

ENDFOR

\(\text{DEST}[\text{MAX}_V \text{L}-1:VL] := 0;\)

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VPMADD52HUQ} & \quad _{\text{m}512i} \ _{\text{mm}512i} \ _{\text{madd}52hi} \ _{\text{epu}64}(\ _{\text{m}512i} \ a, \ _{\text{m}512i} \ b, \ _{\text{m}512i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}512i} \ _{\text{mm}512i} \ _{\text{mask}52hi} \ _{\text{epu}64}(\ _{\text{m}512i} \ s, \ _{\text{mmask}8} \ k, \ _{\text{m}512i} \ a, \ _{\text{m}512i} \ b, \ _{\text{m}512i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}512i} \ _{\text{mm}512i} \ _{\text{maskz}52hi} \ _{\text{epu}64}(\ _{\text{mmask}8} \ k, \ _{\text{m}512i} \ a, \ _{\text{m}512i} \ b, \ _{\text{m}512i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}256i} \ _{\text{mm}256i} \ _{\text{madd}52hi} \ _{\text{epu}64}(\ _{\text{m}256i} \ a, \ _{\text{m}256i} \ b, \ _{\text{m}256i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}256i} \ _{\text{mm}256i} \ _{\text{mask}52hi} \ _{\text{epu}64}(\ _{\text{m}256i} \ s, \ _{\text{mmask}8} \ k, \ _{\text{m}256i} \ a, \ _{\text{m}256i} \ b, \ _{\text{m}256i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}256i} \ _{\text{mm}256i} \ _{\text{maskz}52hi} \ _{\text{epu}64}(\ _{\text{mmask}8} \ k, \ _{\text{m}256i} \ a, \ _{\text{m}256i} \ b, \ _{\text{m}256i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}128i} \ _{\text{mm}128i} \ _{\text{madd}52hi} \ _{\text{epu}64}(\ _{\text{m}128i} \ a, \ _{\text{m}128i} \ b, \ _{\text{m}128i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}128i} \ _{\text{mm}128i} \ _{\text{mask}52hi} \ _{\text{epu}64}(\ _{\text{m}128i} \ s, \ _{\text{mmask}8} \ k, \ _{\text{m}128i} \ a, \ _{\text{m}128i} \ b, \ _{\text{m}128i} \ c); \\
\text{VPMADD52HUQ} & \quad _{\text{m}128i} \ _{\text{mm}128i} \ _{\text{maskz}52hi} \ _{\text{epu}64}(\ _{\text{mmask}8} \ k, \ _{\text{m}128i} \ a, \ _{\text{m}128i} \ b, \ _{\text{m}128i} \ c);
\end{align*}
\]

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."
**VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 Bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512_IFMA</td>
<td>Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM'r/m(r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Multiples packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.
Operation

VPMADD52LUQ (EVEX encoded)

\[(K, L) = (2, 128), (4, 256), (8, 512)\]

FOR \(j := 0\) TO \(KL-1\)

\(i := j \times 64;\)

IF \(k1[j]\) OR *no writemask* THEN

IF src2 is Memory AND EVEX.b=1 THEN

\(tsrc2[63:0] := \text{ZeroExtend64}(src2[51:0]);\)

ELSE

\(tsrc2[63:0] := \text{ZeroExtend64}(src2[i+51:i]);\)

FI;

\(Temp128[127:0] := \text{ZeroExtend64}(src1[i+51:i]) \times tsrc2[63:0];\)

\(Temp2[63:0] := \text{DEST}[i+63:i] + \text{ZeroExtend64}(temp128[51:0]);\)

\(\text{DEST}[i+63:i] := Temp2[63:0];\)

ELSE

IF *zeroing-masking* THEN

\(\text{DEST}[i+63:i] := 0;\)

ELSE *merge-masking*

\(\text{DEST}[i+63:i] \text{ is unchanged};\)

FI;

FI;

ENDFOR

\(\text{DEST}[\text{MAX}_VL-1:VL] := 0;\)

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52LUQ __m512i _mm512_madd52lo_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_mask_madd52lo_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_maskz_madd52lo_epu64(__mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_mask_madd52lo_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_maskz_madd52lo_epu64(__mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m128i _mm128_madd52lo_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm128_mask_madd52lo_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm128_maskz_madd52lo_epu64(__mmask8 k, __m128i a, __m128i b, __m128i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."
VPMASKMOV—Conditional SIMD Integer Packed Loads and Stores

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 8C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load dword values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VPMASKMOVD xmm1, xmm2, m128</td>
<td></td>
<td></td>
<td></td>
<td>Conditionally load dword values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 8C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load qword values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VPMASKMOVQ xmm1, xmm2, m128</td>
<td></td>
<td></td>
<td></td>
<td>Conditionally load qword values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 8C /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load dword values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VPMASKMOVDQ xmm1, xmm2, m128</td>
<td></td>
<td></td>
<td></td>
<td>Conditionally load qword values from xmm2 using mask in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.
VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

**VPMASKMOVD - 256-bit load**

DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0  
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0  
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0  
DEST[127:96] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0  
DEST[159:128] := IF (SRC1[159]) Load_32(mem + 16) ELSE 0  
DEST[191:160] := IF (SRC1[191]) Load_32(mem + 20) ELSE 0  
DEST[223:192] := IF (SRC1[223]) Load_32(mem + 24) ELSE 0  
DEST[255:224] := IF (SRC1[255]) Load_32(mem + 28) ELSE 0

**VPMASKMOVD - 128-bit load**

DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0  
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0  
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0  
DEST[127:97] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0

**VPMASKMOVQ - 256-bit load**

DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0  
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 8) ELSE 0  
DEST[191:128] := IF (SRC1[191]) Load_64(mem + 16) ELSE 0  
DEST[255:196] := IF (SRC1[255]) Load_64(mem + 24) ELSE 0

**VPMASKMOVQ - 128-bit load**

DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0  
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 16) ELSE 0  
DEST[MAXVL-1:128] := 0

**VPMASKMOVD - 256-bit store**

IF (SRC1[31]) DEST[31:0] := SRC2[31:0]  
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]  
IF (SRC1[159]) DEST[159:128] := SRC2[159:128]  
VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]

VPMASKMOVD - 256-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

VPMASKMOVD - 128-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)
VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)
VPMASKMOVD: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);
VPMASKMOVD: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);
VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)
VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)
VPMASKMOVD: __m128i _mm_maskload_epi64(__int64 cont *a, __m128i mask);
VPMASKMOVD: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-23, “Type 6 Class Exception Conditions” (No AC# reported for any mask bit combinations).
VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 29 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.</td>
</tr>
<tr>
<td>VPMOVB2M k1, xmm1</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 29 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.</td>
</tr>
<tr>
<td>VPMOVB2M k1, ymm1</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 29 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.</td>
</tr>
<tr>
<td>VPMOVB2M k1, zmm1</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 29 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.</td>
</tr>
<tr>
<td>VPMOVW2M k1, xmm1</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 29 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.</td>
</tr>
<tr>
<td>VPMOVW2M k1, ymm1</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 29 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.</td>
</tr>
<tr>
<td>VPMOVW2M k1, zmm1</td>
<td></td>
<td></td>
<td>AVX512BW</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 39 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.</td>
</tr>
<tr>
<td>VPMOVD2M k1, xmm1</td>
<td></td>
<td></td>
<td>AVX512DQ</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 39 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.</td>
</tr>
<tr>
<td>VPMOVD2M k1, ymm1</td>
<td></td>
<td></td>
<td>AVX512DQ</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 39 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.</td>
</tr>
<tr>
<td>VPMOVD2M k1, zmm1</td>
<td></td>
<td></td>
<td>AVX512DQ</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 39 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.</td>
</tr>
<tr>
<td>VPMOVQ2M k1, xmm1</td>
<td></td>
<td></td>
<td>AVX512DQ</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 39 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.</td>
</tr>
<tr>
<td>VPMOVQ2M k1, ymm1</td>
<td></td>
<td></td>
<td>AVX512DQ</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 39 /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.</td>
</tr>
<tr>
<td>VPMOVQ2M k1, zmm1</td>
<td></td>
<td></td>
<td>AVX512DQ</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVSB2M (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF SRC[i+7]
    THEN DEST[j] := 1
    ELSE DEST[j] := 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPMOVSb2M (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF SRC[i+15]
    THEN DEST[j] := 1
    ELSE DEST[j] := 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPMOVD2M (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF SRC[i+31]
    THEN DEST[j] := 1
    ELSE DEST[j] := 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPMOVQ2M (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF SRC[i+63]
    THEN DEST[j] := 1
    ELSE DEST[j] := 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
Intel C/C++ Compiler Intrinsic Equivalents

VPMPOVB2M __mmask64 _mm512_movepi8_mask(__m512i);
VPMPOVD2M __mmask16 _mm512_movepi32_mask(__m512i);
VPMPOVQ2M __mmask8 _mm512_movepi64_mask(__m512i);
VPMPOVw2M __mmask32 _mm512_movepi16_mask(__m512i);
VPMPOVB2M __mmask32 _mm256_movepi8_mask(__m256i);
VPMPOVD2M __mmask8 _mm256_movepi32_mask(__m256i);
VPMPOVQ2M __mmask8 _mm256_movepi64_mask(__m256i);
VPMPOVw2M __mmask16 _mm256_movepi16_mask(__m256i);
VPMPOVB2M __mmask16 _mm_movepi8_mask(__m128i);
VPMPOVD2M __mmask8 _mm_movepi32_mask(__m128i);
VPMPOVQ2M __mmask8 _mm_movepi64_mask(__m128i);
VPMPOVw2M __mmask8 _mm_movepi16_mask(__m128i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-55, “Type E7NM Class Exception Conditions.”

Additionally:

#UD If EVEX.vvv = 111B.
### VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 31 /r VPMOVDB xmm1/m32 (k1){z}, xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 21 /r VPMOVSDB xmm1/m32 (k1){z}, xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m32 (k1){z}, xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 31 /r VPMOVDB xmm1/m64 (k1){z}, ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 21 /r VPMOVSDB xmm1/m64 (k1){z}, ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m64 (k1){z}, ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 31 /r VPMOVDB xmm1/m128 (k1){z}, zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 21 /r VPMOVSDB xmm1/m128 (k1){z}, zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m128 (k1){z}, zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Quarter Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB converts unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.
EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

**VPMOVDB instruction (EVEX encoded versions) when dest is a register**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 \text{ TO } KL-1\)

\(i := j * 8\)

\(m := j * 32\)

IF \(k1[j]\) OR *no writemask*

THEN \(\text{DEST}[i+7:i] := \text{TruncateDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[i+7:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(\text{DEST}[i+7:i] := 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL/4] := 0;\)

**VPMOVDB instruction (EVEX encoded versions) when dest is memory**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 \text{ TO } KL-1\)

\(i := j * 8\)

\(m := j * 32\)

IF \(k1[j]\) OR *no writemask*

THEN \(\text{DEST}[i+7:i] := \text{TruncateDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE *DEST*[i+7:i] remains unchanged* ; merging-masking

FI;

ENDFOR

**VPMOVUSDB instruction (EVEX encoded versions) when dest is a register**

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0 \text{ TO } KL-1\)

\(i := j * 8\)

\(m := j * 32\)

IF \(k1[j]\) OR *no writemask*

THEN \(\text{DEST}[i+7:i] := \text{SaturateSignedDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE *merging-masking* ; merging-masking

THEN *DEST*[i+7:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(\text{DEST}[i+7:i] := 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL/4] := 0;\)
VPMOVDB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] := 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL/4] := 0;

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_cvtsepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm512_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVDB __m128i _mm512_cvtusepi32_epi8(__m256i a);
VPMOVDB __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i b);
VPMOVDB __m128i _mm_mask_cvtusepi32_storeu_epi8(void * d, __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.
VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 33 /r VPMOVDW xmm1/m64 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed double-word integers from xmm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 23 /r VPMOVSDW xmm1/m64 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed double-word integers from xmm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m64 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 33 /r VPMOVDW xmm1/m128 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed double-word integers from ymm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 23 /r VPMOVSDW xmm1/m128 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed double-word integers from ymm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m128 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 33 /r VPMOVDW ymm1/m256 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed word integers in ymm1/m256 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 23 /r VPMOVSDW ymm1/m256 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from zmm2 into 16 packed signed word integers in ymm1/m256 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW ymm1/m256 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned word integers in ymm1/m256 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half Mem</td>
<td>ModRMtr/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW converts unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.
EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

**VPMOVFW instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL/2] := 0;

**VPMOVFW instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

**VPMOVUSDW instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR

DEST[MAXVL-1:VL/2] := 0;
VPMOVSDw instruction (EVEX encoded versions) when dest is memory

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
     THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
     ELSE
          *DEST[i+15:i] remains unchanged* ; merging-masking
     FI;
  ENDFOR

VPMOVUSDw instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
     THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
     ELSE
          IF *merging-masking* ; merging-masking
          THEN *DEST[i+15:i] remains unchanged*
          ELSE *zeroing-masking* ; zeroing-masking
              DEST[i+15:i] := 0
          FI
     FI;
  ENDFOR

DEST[MAXVL-1:VL/2] := 0;

VPMOVUSDw instruction (EVEX encoded versions) when dest is memory

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
     THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
     ELSE
          *DEST[i+15:i] remains unchanged* ; merging-masking
     FI;
  ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDW __m256i _mm512_cvtepi32_epi16( __m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m526i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i _mm512_mask_cvtsepi32_epi16(__m512i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void _mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtusepi32_epi16(__m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtusepi32_epi16(__m512i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtepi32_epi16(__m512i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.
### Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register. 

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 28 /r VPMOVM2B xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 28 /r VPMOVM2B ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 28 /r VPMOVM2B zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 38 /r VPMOVM2D xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 38 /r VPMOVM2D ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 38 /r VPMOVM2D zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Operation

**VPMOVM2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
  i := j * 8
  IF SRC[j]
    THEN DEST[i+7:i] := -1
    ELSE DEST[i+7:i] := 0
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPMOVM2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF SRC[j]
    THEN DEST[i+15:i] := -1
    ELSE DEST[i+15:i] := 0
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPMOVM2D (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF SRC[j]
    THEN DEST[i+31:i] := -1
    ELSE DEST[i+31:i] := 0
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VPMOVM2Q (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF SRC[j]
    THEN DEST[i+63:i] := -1
    ELSE DEST[i+63:i] := 0
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i __mm512_movm_epi8(__mmask64 );
VPMOVM2D __m512i __mm512_movm_epi32(__mmask8 );
VPMOVM2Q __m512i __mm512_movm_epi64(__mmask16 );
VPMOVM2W __m512i __mm512_movm_epi16(__mmask32 );
VPMOVM2B __m256i __mm256_movm_epi8(__mmask32 );
VPMOVM2D __m256i __mm256_movm_epi32(__mmask8 );
VPMOVM2Q __m256i __mm256_movm_epi64(__mmask8 );
VPMOVM2W __m256i __mm256_movm_epi16(__mmask16 );
VPMOVM2B __m128i __mm_movm_epi8(__mmask16 );
VPMOVM2D __m128i __mm_movm_epi32(__mmask8 );
VPMOVM2Q __m128i __mm_movm_epi64(__mmask8 );
VPMOVM2W __m128i __mm_movm_epi16(__mmask8 );

SIMD Floating-Point Exceptions
None.

Other Exceptions
EVEX-encoded instruction, see Table 2-55, ”Type E7NM Class Exception Conditions. “
Additionally:
#UD If EVEX.vvvv != 1111B.
### VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 32 /r VPMOVQB xmm1/m16 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 22 /r VPMOVSQB xmm1/m16 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m16 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 32 /r VPMOVQB xmm1/m32 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 22 /r VPMOVSQB xmm1/m32 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m32 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 32 /r VPMOVQB xmm1/m64 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 22 /r VPMOVSQB xmm1/m64 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m64 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Eighth Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL:1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVQB instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;

VPMOVQB instruction (EVEX encoded versions) when dest is memory

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVSB instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;
VPMOVVSQB instruction (EVEX encoded versions) when dest is memory

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)

\(i := j \times 8\)

\(m := j \times 64\)

IF \(k1[j] \) OR *no writemask*
    THEN \(\text{DEST}[i+7:i] := \text{SaturateSignedQuadWordToByte}(\text{SRC}[m+63:m])\)
    ELSE
        \*[\(\text{DEST}[i+7:i] \) remains unchanged*]; merging-masking
    FI;

ENDFOR

VPMOVVSQB instruction (EVEX encoded versions) when dest is a register

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)

\(i := j \times 8\)

\(m := j \times 64\)

IF \(k1[j] \) OR *no writemask*
    THEN \(\text{DEST}[i+7:i] := \text{SaturateUnsignedQuadWordToByte}(\text{SRC}[m+63:m])\)
    ELSE
        IF *merging-masking*; merging-masking
            THEN \*[\(\text{DEST}[i+7:i] \) remains unchanged*]
            ELSE *zeroing-masking*; zeroing-masking
                \(\text{DEST}[i+7:i] := 0\)
            FI
        FI;

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL/8}] := 0;\)

VPMOVVSQB instruction (EVEX encoded versions) when dest is memory

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0 \) TO \(KL-1\)

\(i := j \times 8\)

\(m := j \times 64\)

IF \(k1[j] \) OR *no writemask*
    THEN \(\text{DEST}[i+7:i] := \text{SaturateUnsignedQuadWordToByte}(\text{SRC}[m+63:m])\)
    ELSE
        \*[\(\text{DEST}[i+7:i] \) remains unchanged*]; merging-masking
    FI;

ENDFOR
Intel C/C++ Compiler Intrinsics Equivalents

VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m512i s, __mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m512i s, __mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvv != 111B.
## VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 25 /r VPMOVSQD xmm1/m64 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed double-word integers in xmm1/m64 using signed saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m64 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned double-word integers in xmm1/m64 using unsigned saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 25 /r VPMOVSQD xmm1/m128 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed double-word integers in xmm1/m128 using signed saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD xmm1/m128 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned double-word integers in xmm1/m128 using unsigned saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 35 /r VPMOVQD ymm1/m256 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 25 /r VPMOVSQD ymm1/m256 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed double-word integers in ymm1/m256 using signed saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD ymm1/m256 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned double-word integers in ymm1/m256 using unsigned saturation subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half Mem</td>
<td>ModRMbr/m (w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

VPMOQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.
Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQD instruction (EVEX encoded version) reg-reg form (KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

VPMOVQD instruction (EVEX encoded version) memory form (KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVQD instruction (EVEX encoded version) reg-reg form (KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
VPMOVUSQD instruction (EVEX encoded version) memory form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSQD instruction (EVEX encoded version) reg-reg form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

VPMOVUSQD instruction (EVEX encoded version) memory form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSQD instruction (EVEX encoded version) memory form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
Intrinsic Equivalents

VPMOVQD __m256i __m512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD void __m512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD void __m512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD void __m512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);

VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”
Additionally:

#UD If EVEX.vvvv != 1111B.
# VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

## Opcode/Description

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 34 /r VPMOVQW xmm1/m32 (k1)(z), xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 24 /r VPMOVSQW xmm1/m32 (k1)(z), xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed quad-word integers from xmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m32 (k1)(z), xmm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 34 /r VPMOVQW xmm1/m64 (k1)(z), ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 24 /r VPMOVSQW xmm1/m64 (k1)(z), ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m64 (k1)(z), ymm2</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 34 /r VPMOVQW xmm1/m128 (k1)(z), zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 24 /r VPMOVSQW xmm1/m128 (k1)(z), zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW xmm1/m128 (k1)(z), zmm2</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Quarter Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

## Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.
EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

**VPMOVQW instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1  
i := j * 16  
m := j * 64  
IF k1[j] OR *no writemask*
  THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] := 0
  FI
ENDFOR

DEST[MAXVL-1:VL/4] := 0;

**VPMOVQW instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1  
i := j * 16  
m := j * 64  
IF k1[j] OR *no writemask*
  THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI
ENDFOR

**VPMOVUSQW instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1  
i := j * 16  
m := j * 64  
IF k1[j] OR *no writemask*
  THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] := 0
  FI
ENDFOR

DEST[MAXVL-1:VL/4] := 0;
VPMOVQSW/DOWN CONVERT QWORD TO WORD

VPMOVQSW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVQSW instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;

VPMOVQSW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i _mm512_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_cvtsepi64_epi16( __m512i a);
VPMOVQW __m128i _mm512_mask_cvtsepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtsepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtsepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_cvtusepi64_epi16( __m512i a);
VPMOVQW __m128i _mm512_mask_cvtusepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtusepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtusepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtepi64_epi32( __m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtepi64_epi32( __m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtepi64_storeu_epi32( void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtsepi64_epi32( __m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtsepi64_epi32( __m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtsepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtsepi64_storeu_epi32( void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtepi64_epi32( __m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtepi64_epi32( __m256i s, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtepi64_storeu_epi32( void * d, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_cvtsepi64_epi32( __m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtsepi64_epi32( __m256i s, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtsepi64_storeu_epi32( void * d, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtsepi64_epi32( __m128i a);
VPMOVUSQD __m128i _mm_mask_cvtsepi64_epi32( __m128i s, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtsepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtsepi64_storeu_epi32( void * d, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32( __m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32( __m256i s, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32( void * d, __mmask8 k, __m256i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-53, "Type E6 Class Exception Conditions."

Additionally:

#UD If EVEX.vvv = 1111B.
# VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

## Opcode/Operand Description

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 30 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVWB xmm1/m64 [k1][z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 20 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSWB xmm1/m64 [k1][z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSWB xmm1/m64 [k1][z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 30 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVWB xmm1/m128 [k1][z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 20 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSWB xmm1/m128 [k1][z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSWB xmm1/m128 [k1][z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 30 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts 32 packed word integers from zmm2 into 32 packed bytes in xmm1/m256 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVWB ymm1/m256 [k1][z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 20 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in xmm1/m256 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSWB ymm1/m256 [k1][z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 10 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in xmm1/m256 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSWB ymm1/m256 [k1][z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Half Mem</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVWB instruction (EVEX encoded versions) when dest is a register

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \( j := 0 \) TO \( KL-1 \)
  \( i := j * 8 \)
  \( m := j * 16 \)
  IF \( k1[j] \) OR *no writemask*
    THEN \( \text{DEST}[i+7:i] := \text{TruncateWordToByte} \ (\text{SRC}[m+15:m]) \)
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *\( \text{DEST}[i+7:i] \) remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        \( \text{DEST}[i+7:i] = 0 \)
      FI
    FI;
  ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL/2] := 0; \)

VPMOVWB instruction (EVEX encoded versions) when dest is memory

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \( j := 0 \) TO \( KL-1 \)
  \( i := j * 8 \)
  \( m := j * 16 \)
  IF \( k1[j] \) OR *no writemask*
    THEN \( \text{DEST}[i+7:i] := \text{TruncateWordToByte} \ (\text{SRC}[m+15:m]) \)
    ELSE
      \( *\text{DEST}[i+7:i] \) remains unchanged* ; merging-masking
    FI;
  ENDFOR

VPMOVUSwB instruction (EVEX encoded versions) when dest is a register

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \( j := 0 \) TO \( KL-1 \)
  \( i := j * 8 \)
  \( m := j * 16 \)
  IF \( k1[j] \) OR *no writemask*
    THEN \( \text{DEST}[i+7:i] := \text{SaturateSignedWordToByte} \ (\text{SRC}[m+15:m]) \)
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *\( \text{DEST}[i+7:i] \) remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        \( \text{DEST}[i+7:i] = 0 \)
      FI
    FI;
  ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL/2] := 0; \)
VPMOVSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Ki-1
  i := j * 8
  m := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])
    ELSE
       *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Ki-1
  i := j * 8
  m := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
    ELSE
       IF *merging-masking* ; merging-masking
          THEN *DEST[i+7:i] remains unchanged*
          ELSE *zeroing-masking* ; zeroing-masking
             DEST[i+7:i] = 0
       FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

VPMOVUSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO Ki-1
  i := j * 8
  m := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
    ELSE
       *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVUSWB __m256i mm512_cvtepi16_epi8(__m512i a);
VPMOVUSWB __m256i _mm512_mask_cvtepi16_epi8(__m512i a, __mmask32 k, __m512i b);
VPMOVUSWB void _mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m512i _mm512_cvtepi16_epi8(__m512i a);
VPMOVUSWB __m512i _mm512_maskz_cvtepi16_epi8(__m512i a, __mmask32 k, __m512i b);
VPMOVUSWB __m512i _mm512_maskz_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m256i _mm512_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m256i _mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m256i _mm256_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m256i _mm256_maskz_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m256i _mm256_maskz_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.
VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 83 /r</td>
<td>VPMULTISHIFTQB xmm1 [k1][z], xmm2,xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI AVX512VL</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 83 /r</td>
<td>VPMULTISHIFTQB ymm1 [k1][z], ymm2,ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI AVX512VL</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 83 /r</td>
<td>VPMULTISHIFTQB zmm1 [k1][z], zmm2,zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is extracted from the input qword source at the location specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.
**Operation**

**VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR i := 0 TO KL-1
  IF EVEX.b=1 AND src2 is memory THEN
    tcur := src2.qword[0]; //broadcasting
  ELSE
    tcur := src2.qword[i];
  FI;
  FOR j := 0 to 7
    ctrl := src1.qword[i].byte[j] & 63;
    FOR k := 0 to 7
      res.bit[k] := tcur.bit[(ctrl+k) mod 64 ];
    ENDFOR
    IF k1[i*8+j] or no writemask THEN
      DEST.qword[i].byte[j] := res;
    ELSE IF zeroing-masking THEN
      DEST.qword[i].byte[j] := 0;
    ENDIF
  ENDFOR
ENDFOR
DEST.qword[MAX_VL-1:VL] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

\[
\begin{align*}
&\text{VPMULTISHIFTQB } \_\_m512i \_\_mm512\_\_multishift\_\_epi64\_\_epi8(\_\_m512i a, \_\_m512i b); \\
&\text{VPMULTISHIFTQB } \_\_m512i \_\_mm512\_\_mask\_\_multishift\_\_epi64\_\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b); \\
&\text{VPMULTISHIFTQB } \_\_m512i \_\_mm512\_\_maskz\_\_multishift\_\_epi64\_\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b); \\
&\text{VPMULTISHIFTQB } \_\_m256i \_\_mm256\_\_multishift\_\_epi64\_\_epi8(\_\_m256i a, \_\_m256i b); \\
&\text{VPMULTISHIFTQB } \_\_m256i \_\_mm256\_\_mask\_\_multishift\_\_epi64\_\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b); \\
&\text{VPMULTISHIFTQB } \_\_m256i \_\_mm256\_\_maskz\_\_multishift\_\_epi64\_\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b); \\
&\text{VPMULTISHIFTQB } \_\_m128i \_\_mm128\_\_multishift\_\_epi64\_\_epi8(\_\_m128i a, \_\_m128i b); \\
&\text{VPMULTISHIFTQB } \_\_m128i \_\_mm128\_\_mask\_\_multishift\_\_epi64\_\_epi8(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b); \\
&\text{VPMULTISHIFTQB } \_\_m128i \_\_mm128\_\_maskz\_\_multishift\_\_epi64\_\_epi8(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-50, "Type E4NF Class Exception Conditions."
### VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 54 /r VPOPCNTB xmm1[k1]{z}, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 54 /r VPOPCNTB ymm1[k1]{z}, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 54 /r VPOPCNTB zmm1[k1]{z}, zmm2/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 54 /r VPOPCNTW xmm1[k1]{z}, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 54 /r VPOPCNTW ymm1[k1]{z}, ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 54 /r VPOPCNTW zmm1[k1]{z}, zmm2/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 55 /r VPOPCNTD xmm1[k1]{z}, xmm2/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 55 /r VPOPCNTD ymm1[k1]{z}, ymm2/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1[k1]{z}, zmm2/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ</td>
<td>Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 55 /r VPOPCNTQ xmm1[k1]{z}, xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128/m64bcst and puts the result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 55 /r VPOPCNTQ ymm1[k1]{z}, ymm2/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256/m64bcst and puts the result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1[k1]{z}, zmm2/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ</td>
<td>Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.
Operation

VPOPCNTB

\((KL, VL) = (16, 128), (32, 256), (64, 512)\)

FOR \(j := 0\) TO \(KL-1\):
  IF \(\text{MaskBit}(j) \) OR *no writemask*:
    \(\text{DEST}.byte[j] := \text{POPCNT}(\text{SRC}.byte[j])\)
  ELSE IF *merging-masking*:
    \(\text{DEST}.byte[j]\) remains unchanged*
  ELSE:
    \(\text{DEST}.byte[j] := 0\)

\(\text{DEST}[\text{MAX}_V L-1: V L] := 0\)

VPOPCNTW

\((KL, VL) = (8, 128), (16, 256), (32, 512)\)

FOR \(j := 0\) TO \(KL-1\):
  IF \(\text{MaskBit}(j) \) OR *no writemask*:
    \(\text{DEST}.word[j] := \text{POPCNT}(\text{SRC}.word[j])\)
  ELSE IF *merging-masking*:
    \(\text{DEST}.word[j]\) remains unchanged*
  ELSE:
    \(\text{DEST}.word[j] := 0\)

\(\text{DEST}[\text{MAX}_V L-1: V L] := 0\)

VPOPCNTD

\((KL, VL) = (4, 128), (8, 256), (16, 512)\)

FOR \(j := 0\) TO \(KL-1\):
  IF \(\text{MaskBit}(j) \) OR *no writemask*:
    IF \(\text{SRC}\) is broadcast memop:
      \(t := \text{SRC}.\text{dword}[0]\)
    ELSE:
      \(t := \text{SRC}.\text{dword}[j]\)
    \(\text{DEST}.\text{dword}[j] := \text{POPCNT}(t)\)
  ELSE IF *merging-masking*:
    \(\text{DEST}.\text{dword}[j]\) remains unchanged*
  ELSE:
    \(\text{DEST}.\text{dword}[j] := 0\)

\(\text{DEST}[\text{MAX}_V L-1: V L] := 0\)

VPOPCNTQ

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)

FOR \(j := 0\) TO \(KL-1\):
  IF \(\text{MaskBit}(j) \) OR *no writemask*:
    IF \(\text{SRC}\) is broadcast memop:
      \(t := \text{SRC}.\text{qword}[0]\)
    ELSE:
      \(t := \text{SRC}.\text{qword}[j]\)
    \(\text{DEST}.\text{qword}[j] := \text{POPCNT}(t)\)
  ELSE IF *merging-masking*:
    \(\text{DEST}.\text{qword}[j]\) remains unchanged*
  ELSE:
    \(\text{DEST}.\text{qword}[j] := 0\)

\(\text{DEST}[\text{MAX}_V L-1: V L] := 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTW __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi64(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi64(__mmask32, __m512i);
VPOPCNTW __m128i _mm_popcnt_epi32(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi32(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi32(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi32(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi32(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi32(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi32(__mmask32, __m512i);
VPOPCNTW __m128i _mm_popcnt_epi8(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi8(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi8(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi8(__mmask64, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."
### VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 15 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 72 /1 ib VPROLD xmm1 [k1][z], xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 15 /r VPROLQ xmm1 [k1][z], xmm2/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 72 /1 ib VPROLQ xmm1 [k1][z], xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 15 /r VPROLD ymm1 [k1][z], ymm2/m256/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 72 /1 ib VPROLD ymm1 [k1][z], ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 15 /r VPROLQ ymm1 [k1][z], ymm2/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 72 /1 ib VPROLQ ymm1 [k1][z], ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 15 /r VPROLD zmm1 [k1][z], zmm2/m512/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 72 /1 ib VPROLD zmm1 [k1][z], zmm2/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm3/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 15 /r VPROLQ zmm1 [k1][z], zmm2/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 72 /1 ib VPROLQ zmm1 [k1][z], zmm2/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description
Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation
LEFT.Rotate.DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT.Rotate.QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC << COUNT) | (SRC >> (64 - COUNT));

VPROLD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := LEFT.Rotate.DWORDS(SRC1[31:0], imm8)
    ELSE DEST[i+31:i] := LEFT.Rotate.DWORDS(SRC1[i+31:i], imm8)
  FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPROLVD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := LEFT.Rotate.DWORDS(SRC1[i+31:i], SRC2[31:0])
    ELSE DEST[i+31:i] := LEFT.Rotate.DWORDS(SRC1[i+31:i], SRC2[i+31:i])
  FI;
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] := 0
    FI
ENDIF
ENDFOR
DEST[MAXVL-1:VL] := 0

VPROLQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[63:0], imm8)
            ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPROLVQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
            ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32(__mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32(__mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32(__mmask8 k, __m128i a, int imm);
VPROLD __m128i _mm_rol_epi64(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi64(__mmask8 k, __m128i a, int imm);
VPROLD __m256i _mm256_rol_epi64(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi64(__mmask8 k, __m256i a, int imm);
VPROLD __m512i _mm512_rol_epi64(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi64(__mmask8 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi64(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi64(__mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi32(__mmask8 k, __m512i a, __m512i cnt);
VPROLD __m256i _mm256_rolv_epi64(__m256i a, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPROLD __m128i _mm_rolv_epi64(__m128i a, __m128i cnt);
VPROLD __m128i _mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLD __m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."
## VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right

<table>
<thead>
<tr>
<th>Opcode/</th>
<th>Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 14 /r</td>
<td>VPRORD xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.0F.W0 72 /0 ib</td>
<td>VPRORD xmm1 [k1]{z}, xmm2/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 14 /r</td>
<td>VPRORQ xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.0F.W1 72 /0 ib</td>
<td>VPRORQ xmm1 [k1]{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 14 /r</td>
<td>VPRORD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.0F.W0 72 /0 ib</td>
<td>VPRORD ymm1 [k1]{z}, ymm2/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 14 /r</td>
<td>VPRORQ ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.0F.W1 72 /0 ib</td>
<td>VPRORQ ymm1 [k1]{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 14 /r</td>
<td>VPRORD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.0F.W0 72 /0 ib</td>
<td>VPRORD zmm1 [k1]{z}, zmm2/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 14 /r</td>
<td>VPRORQ zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.0F.W1 72 /0 ib</td>
<td>VPRORQ zmm1 [k1]{z}, zmm2/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>VEX.vvvv (w)</td>
<td>ModRM/r/m (R)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM/reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (R)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description
Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC >> COUNT) | (SRC << (64 - COUNT));

VPRORD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
      ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
  ENDFOR
DEST[MAXVL-1:VL] := 0

VPRORVD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], SRC2[31:0])
      ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
    FI;
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+31:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPRORQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] := RIGHT_ROTATE_QWORMDS(SRC1[63:0], imm8)
    ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
      FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPRORVQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
    ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
      FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, int imm);

VPRORQ __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORQ __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORQ __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORQ __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORQ __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORQ __m256i _mm256_maskz_ror_epi64(__mmask8 k, __m256i a, int imm);
VPRORQ __m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORQ __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORQ __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);

VPRORVD __m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
VPRORVD __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
**VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter qword values to memory using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory
ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.
The instruction will #UD fault if EVEX.Z = 1.

**Operation**

**BASE_ADDR** stands for the memory operand base address (a GPR); may not exist

**VINDEX** stands for the memory operand vector of indices (a ZMM register)

**SCALE** stands for the memory operand scalar (1, 2, 4 or 8)

**DISP** is the optional 1 or 4 byte displacement

**VPSCATTERDD (EVEX encoded versions)**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j\) := 0 TO KL-1
  \(i := j * 32\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(MEM[BASE_ADDR + \text{SignExtend}(VINDEX[i+31:i]) * SCALE + DISP] := SRC[i+31:i]\)
    \(k1[j] := 0\)
  Fi;
ENDFOR
\(k1[\text{MAX}_KL-1:KL] := 0\)

**VPSCATTERDQ (EVEX encoded versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j\) := 0 TO KL-1
  \(i := j * 64\)
  \(k := j * 32\)
  IF \(k1[j]\) OR *no writemask*
    THEN \(MEM[BASE_ADDR + \text{SignExtend}(VINDEX[k+31:k]) * SCALE + DISP] := SRC[i+63:i]\)
    \(k1[j] := 0\)
  Fi;
VPSCATTERDD (EVEX encoded versions)
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] := SRC[i+31:i]
    k1[j] := 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

VPSCATTERDQ (EVEX encoded versions)
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] := SRC[i+63:i]
    k1[j] := 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi64(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);

SIMD Floating-Point Exceptions
None.
Other Exceptions
See Table 2-61, "Type E12 Class Exception Conditions."
### VPSHLD—Concatenate and Shift Packed Data Left Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 70 /r /ib VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 70 /r /ib VPSHLDW ymm1{k1}[z], ymm2, ymm3/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 70 /r /ib VPSHLDW zmm1{k1}[z], zmm2, zmm3/m512, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 71 /r /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 71 /r /ib VPSHLDD ymm1{k1}[z], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 71 /r /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 71 /r /ib VPSHLDQ ymm1{k1}[z], ymm2, ymm3/m256/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 71 /r /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

### Description
Concatenate packed data, extract result shifted to the left by constant value.
This instruction supports memory fault suppression.
**Operation**

**VPSHLDw DEST, SRC2, SRC3, imm8**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:

  IF MaskBit(j) OR *no writemask*:
    tmp := concat(SRC2.word[j], SRC3.word[j]) << (imm8 & 15)
    DEST.word[j] := tmp.word[1]
  ELSE IF *zeroing*:
    DEST.word[j] := 0
  *ELSE DEST.word[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHLDd DEST, SRC2, SRC3, imm8**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1:

  IF SRC3 is broadcast memop:
    tsrc3 := SRC3.dword[0]
  ELSE:
    tsrc3 := SRC3.dword[j]

  IF MaskBit(j) OR *no writemask*:
    tmp := concat(SRC2.dword[j], tsrc3) << (imm8 & 31)
    DEST.dword[j] := tmp.dword[1]
  ELSE IF *zeroing*:
    DEST.dword[j] := 0
  *ELSE DEST.dword[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHLDq DEST, SRC2, SRC3, imm8**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1:

  IF SRC3 is broadcast memop:
    tsrc3 := SRC3.qword[0]
  ELSE:
    tsrc3 := SRC3.qword[j]

  IF MaskBit(j) OR *no writemask*:
    tmp := concat(SRC2.qword[j], tsrc3) << (imm8 & 63)
    DEST.qword[j] := tmp.qword[1]
  ELSE IF *zeroing*:
    DEST.qword[j] := 0
  *ELSE DEST.qword[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0
INSTRUCTION SET REFERENCE, V

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);

VPSHLDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDQ __m512i _mm512_maskz_shldi_epi64(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, “Type E4 Class Exception Conditions.”
VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 70 /r VPSHLDV w xmm1{k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 70 /r VPSHLDV w ymm1{k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 70 /r VPSHLDV w zmm1{k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the left by value in xmm3/m512 into zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V AVX512_VBMI2 AVX512VL</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the left by value in xmm3/m512 into zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM.r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM.r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Concatenate packed data, extract result shifted to the left by variable value.
This instruction supports memory fault suppression.
**Operation**

FUNCTION concat(a, b):
   IF words:
      d.word[1] := a
      d.word[0] := b
      return d
   ELSE IF dwords:
      q.dword[1] := a
      q.dword[0] := b
      return q
   ELSE IF qwords:
      o.qword[1] := a
      o.qword[0] := b
      return o

**VPSHLDVW DEST, SRC2, SRC3**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:
   IF MaskBit(j) OR *no writemask*:
      tmp := concat(DEST.word[j], SRC2.word[j]) << (SRC3.word[j] & 15)
      DEST.word[j] := tmp.word[1]
   ELSE IF *zeroing*:
      DEST.word[j] := 0
   *ELSE DEST.word[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHLDVD DEST, SRC2, SRC3**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1:
   IF SRC3 is broadcast memop:
      tsrc3 := SRC3.dword[0]
   ELSE:
      tsrc3 := SRC3.dword[j]
   IF MaskBit(j) OR *no writemask*:
      tmp := concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
      DEST.dword[j] := tmp.dword[1]
   ELSE IF *zeroing*:
      DEST.dword[j] := 0
   *ELSE DEST.dword[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0
VPShldVQ DEST, SRC2, SRC3

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
    DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i  _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i  _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i  _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i  _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i  _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i  _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m128i  _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i  _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i  _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i  _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i  _mm256_mask_shldv_epi32(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVD __m256i  _mm256_maskz_shldv_epi32(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVD __m512i  _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVD __m512i  _mm512_mask_shldv_epi64(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVD __m512i  _mm512_maskz_shldv_epi64(__mmask32, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."

VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical
### VPSHRD—Concatenate and Shift Packed Data Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 72 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 72 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 72 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.
Operation

\texttt{VPSHRDW \textit{DEST}, \textit{SRC2}, \textit{SRC3}, \textit{imm8}}

\((\text{KL, VL}) = (8, 128), (16, 256), (32, 512)\)

\text{FOR } j := 0 \text{ TO } \text{KL}-1:\n
\quad \text{IF} \text{ MaskBit}(j) \text{ OR } \text{no writemask}:\n
\qquad \text{DEST}.\text{word}[j] := \text{concat}([\text{SRC3}.\text{word}[j], \text{SRC2}.\text{word}[j]]) \gg (\text{imm8} \& 15)

\quad \text{ELSE IF} \text{ zeroing}:\n
\qquad \text{DEST}.\text{word}[j] := 0

\quad \text{ELSE DEST}.\text{word}[j] \text{ remains unchanged}

\quad \text{DEST}[\text{MAX}_\text{VL}-1:\text{VL}] := 0

\texttt{VPSHRDD \textit{DEST}, \textit{SRC2}, \textit{SRC3}, \textit{imm8}}

\((\text{KL, VL}) = (4, 128), (8, 256), (16, 512)\)

\text{FOR } j := 0 \text{ TO } \text{KL}-1:\n
\quad \text{IF} \text{ SRC3 is broadcast memop}:\n
\qquad \text{tsrc3} := \text{SRC3}.\text{dword}[0]

\quad \text{ELSE}:\n
\qquad \text{tsrc3} := \text{SRC3}.\text{dword}[j]

\quad \text{IF} \text{ MaskBit}(j) \text{ OR } \text{no writemask}:\n
\qquad \text{DEST}.\text{dword}[j] := \text{concat}([\text{tsrc3}, \text{SRC2}.\text{dword}[j]]) \gg (\text{imm8} \& 31)

\quad \text{ELSE IF} \text{ zeroing}:\n
\qquad \text{DEST}.\text{dword}[j] := 0

\quad \text{ELSE DEST}.\text{dword}[j] \text{ remains unchanged}

\quad \text{DEST}[\text{MAX}_\text{VL}-1:\text{VL}] := 0

\texttt{VPSHRDQ \textit{DEST}, \textit{SRC2}, \textit{SRC3}, \textit{imm8}}

\((\text{KL, VL}) = (2, 128), (4, 256), (8, 512)\)

\text{FOR } j := 0 \text{ TO } \text{KL}-1:\n
\quad \text{IF} \text{ SRC3 is broadcast memop}:\n
\qquad \text{tsrc3} := \text{SRC3}.\text{qword}[0]

\quad \text{ELSE}:\n
\qquad \text{tsrc3} := \text{SRC3}.\text{qword}[j]

\quad \text{IF} \text{ MaskBit}(j) \text{ OR } \text{no writemask}:\n
\qquad \text{DEST}.\text{qword}[j] := \text{concat}([\text{tsrc3}, \text{SRC2}.\text{qword}[j]]) \gg (\text{imm8} \& 63)

\quad \text{ELSE IF} \text{ zeroing}:\n
\qquad \text{DEST}.\text{qword}[j] := 0

\quad \text{ELSE DEST}.\text{qword}[j] \text{ remains unchanged}

\quad \text{DEST}[\text{MAX}_\text{VL}-1:\text{VL}] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPShrdQ __m128i _mm_shrdi_epi64(__m128i, __m128i, int);
VPShrdQ __m128i _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPShrdQ __m128i _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPShrdQ __m256i _mm256_shrdi_epi64(__m256i, __m256i, int);
VPShrdQ __m256i _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPShrdQ __m256i _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPShrdQ __m512i _mm512_shrdi_epi64(__m512i, __m512i, int);
VPShrdQ __m512i _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPShrdQ __m512i _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPShrdQ __m512i _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);

VPSHRDD __m128i _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i _mm256_mask_shrdi_epi32(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDD __m256i _mm256_maskz_shrdi_epi32(__mmask16, __m256i, __m256i, int);
VPSHRDD __m512i _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i _mm512_mask_shrdi_epi32(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDD __m512i _mm512_maskz_shrdi_epi32(__mmask32, __m512i, __m512i, int);

VPSHRDW __m128i _mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i _mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i _mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i _mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i _mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i _mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i _mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i _mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i _mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”
### VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 72 / r VPSHRDVW xmm1[k1]{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 72 / r VPSHRDVW ymm1[k1]{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the right by value in ymm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 72 / r VPSHRDVW zmm1[k1]{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 73 / r VPSHRDVQ xmm1[k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 73 / r VPSHRDVQ ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the right by value in ymm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 73 / r VPSHRDVQ zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 73 / r VPSHRDVQ xmm1[k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 73 / r VPSHRDVQ ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the right by value in ymm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 73 / r VPSHRDVQ zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Concatenate packed data, extract result shifted to the right by variable value.

This instruction supports memory fault suppression.
**Operation**

**VPSHRDVW DEST, SRC2, SRC3**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

FOR \(j := 0 \) TO \(KL-1\):

- **IF MaskBit\((j)\) OR *no writemask***:
  
  \[\text{DEST}.\text{word}[j] := \text{concat}(\text{SRC2}.\text{word}[j], \text{DEST}.\text{word}[j]) \gg (\text{SRC3}.\text{word}[j] \& 15)\]

- **ELSE IF *zeroing***:
  
  \[\text{DEST}.\text{word}[j] := 0\]

  *ELSE DEST.word[j] remains unchanged*

\[\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\]

**VPSHRDVD DEST, SRC2, SRC3**

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

FOR \(j := 0 \) TO \(KL-1\):

- **IF SRC3 is broadcast memop**:
  
  \[t\text{src3} := \text{SRC3}.\text{dword}[0]\]

- **ELSE**:
  
  \[t\text{src3} := \text{SRC3}.\text{dword}[j]\]

- **IF MaskBit\((j)\) OR *no writemask***:
  
  \[\text{DEST}.\text{dword}[j] := \text{concat}(\text{SRC2}.\text{dword}[j], \text{DEST}.\text{dword}[j]) \gg (t\text{src3} \& 31)\]

- **ELSE IF *zeroing***:
  
  \[\text{DEST}.\text{dword}[j] := 0\]

  *ELSE DEST.dword[j] remains unchanged*

\[\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\]

**VPSHRDVQ DEST, SRC2, SRC3**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j := 0 \) TO \(KL-1\):

- **IF SRC3 is broadcast memop**:
  
  \[t\text{src3} := \text{SRC3}.\text{qword}[0]\]

- **ELSE**:
  
  \[t\text{src3} := \text{SRC3}.\text{qword}[j]\]

- **IF MaskBit\((j)\) OR *no writemask***:
  
  \[\text{DEST}.\text{qword}[j] := \text{concat}(\text{SRC2}.\text{qword}[j], \text{DEST}.\text{qword}[j]) \gg (t\text{src3} \& 63)\]

- **ELSE IF *zeroing***:
  
  \[\text{DEST}.\text{qword}[j] := 0\]

  *ELSE DEST.qword[j] remains unchanged*

\[\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVQ __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi32(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi32(__mmask32, __m512i, __m512i, __m512i);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”
VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Extract values in zmm2 using control bits of zmm3/m512 with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The VPSHUFBITQMB instruction performs a bit gather select using second source as control and first source as data. Each bit uses 6 control bits (2nd source operand) to select which data bit is going to be gathered (first source operand). A given bit can only access 64 different bits of data (first 64 destination bits can access first 64 data bits, second 64 destination bits can access second 64 data bits, etc.).

Control data for each output bit is stored in 8 bit elements of SRC2, but only the 6 least significant bits of each element are used.

This instruction uses write masking (zeroing only). This instruction supports memory fault suppression.

The first source operand is a ZMM register. The second source operand is a ZMM register or a memory location. The destination operand is a mask register.

Operation

VPSHUFBITQMB DEST, SRC1, SRC2
(KL, VL) = (16,128), (32,256), (64, 512)
FOR i := 0 TO KL/8-1:
  //Qword
  FOR j := 0 to 7:
    // Byte
    IF k2[i*8+j] or *no writemask*:
      m := SRC2.qword[i].byte[j] & 0x3F
      k1[i*8+j] := SRC1.qword[i].bit[m]
    ELSE:
      k1[i*8+j] := 0
  k1[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 _mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 _mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 _mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 _mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 _mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 _mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);
VPSLLVw/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 12 /r VPSLLVW xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 12 /r VPSLLVW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 12 /r VPSLLVW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 47 /r VPSLLVD xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 47 /r VPSLLVQ ymm1 [k1]{z}, xmm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m256/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 47 /r VPSLLVQ zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 47 /r VPSLLVQ xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 47 /r VPSLLVQ ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 47 /r VPSLLVQ zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:mr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:mr/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:mr/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description
Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation
VPSLLVW (EVEX encoded version)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSLLVD (VEX.128 version)
COUNT_0 := SRC2[31 : 0]
   (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
   DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
   DEST[31:0] := 0;
   (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
   DEST[127:96] := ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
   DEST[127:96] := 0;
   DEST[MAXVL-1:128] := 0;

VPSLLVD (VEX.256 version)
COUNT_0 := SRC2[31 : 0];
   (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
   DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
   DEST[31:0] := 0;
   (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
   DEST[255:224] := ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
   DEST[255:224] := 0;
   DEST[MAXVL-1:256] := 0;

VPSLLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
         THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
         ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] := 0
      FI
   FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSLLVQ (VEX.128 version)
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
   DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
   DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
   DEST[127:64] := ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
   DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;

VPSLLVQ (VEX.256 version)
COUNT_0 := SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
   DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
   DEST[63:0] := 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
   DEST[255:192] := ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
   DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;

VPSLLVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
         THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
         ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+63:i] := 0
      FI
   FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSLLW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_sllv_epi32(__m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm128i_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm128i_mask_sllv_epi32(__m128i s, __mmask16 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm128i_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m256i _mm256_sllv_epi32 (__m256i m, __m256i count);
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPSLLVD/VPSLLVQ, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
VPSRAVw/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 11 /r VPSRAVw xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 11 /r VPSRAVw ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 11 /r VPSRAVw zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512Bw</td>
<td>Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 46 /r VPSRAVD xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 46 /r VPSRAVD ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 46 /r VPSRAVD zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 11 /r VPSRAVQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 11 /r VPSRAVQ ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 11 /r VPSRAVQ zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description
Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element is filled with the corresponding sign bit of the source element.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVQ/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVW (EVEX encoded version)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[
\begin{align*}
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j \times 16 \\
\quad \text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \quad \text{THEN} \\
\quad \quad \quad \text{COUNT} := \text{SRC2}[i+3:j] \\
\quad \quad \quad \text{IF COUNT < 16} \\
\quad \quad \quad \quad \text{THEN} \\
\quad \quad \quad \quad \quad \text{DEST}[i+15:j] := \text{SignExtend}(\text{SRC1}[i+15:j] >> \text{COUNT}) \\
\quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \text{FOR } k := 0 \text{ TO } 15 \\
\quad \quad \quad \quad \quad \text{DEST}[i+k] := \text{SRC1}[i+15] \\
\quad \quad \quad \quad \text{ENDFOR;} \\
\quad \quad \quad \text{FI} \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{IF *merging-masking* ; merging-masking} \\
\quad \quad \quad \quad \text{THEN } \text{*DEST}[i+15:j] \text{ remains unchanged*} \\
\quad \quad \quad \quad \text{ELSE } ; \text{zeroing-masking} \\
\quad \quad \quad \quad \text{DEST}[i+15:j] := 0 \\
\quad \quad \text{FI} \\
\quad \text{FI} \text{;} \\
\text{ENDFOR;} \\
\text{DEST}[\text{MAXVL-1:VL}] := 0;
\end{align*}
\]
VPSRAVD (VEX.128 version)
COUNT_0 := SRC2[31:0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127:96];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] := SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] := 0;

VPSRAVD (VEX.256 version)
COUNT_0 := SRC2[31:0];
(* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 := SRC2[255:224];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] := SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] := 0;

VPSRAVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        COUNT := SRC2[4:0]
        IF COUNT < 32
          THEN
            DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
          ELSE
            FOR k := 0 TO 31
              DEST[i+k] := SRC1[i+31]
            ENDFOR;
          FI
        ELSE
          COUNT := SRC2[i+4:i]
          IF COUNT < 32
            THEN
              DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
            ELSE
              FOR k := 0 TO 31
                DEST[i+k] := SRC1[i+31]
              ENDFOR;
            FI
          ELSE
            FI;
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
          ELSE ; zeroing-masking
            DEST[31:0] := 0
          FI
        ENDIF;
      ENDIF;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
        DEST[31:0] := 0
      FI
    ENDIF;
  ENDIF;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSRAVQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                COUNT := SRC2[5:0]
                IF COUNT < 64
                    THEN
                        DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                    ELSE
                        FOR k := 0 TO 63
                            DEST[i+k] := SRC1[i+63]
                        ENDFOR;
                FI
            ELSE
                COUNT := SRC2[i+5:i]
                IF COUNT < 64
                    THEN
                        DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
                    ELSE
                        FOR k := 0 TO 63
                            DEST[i+k] := SRC1[i+63]
                        ENDFOR;
                FI
            ELSE
                ; merging-masking
                THEN
                    *DEST[63:0] remains unchanged*
                ELSE
                    ; zeroing-masking
                    DEST[63:0] := 0
                FI
        FI;
    ENDFOR;

DEST[MAXVL-1:VL] := 0;
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi16(__m128i s, __mmask16 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i m, __m256i count)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instruction, see Table 2-49, “Type 4E Class Exception Conditions.”
### VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 10 /r VPSRLVW xmm1 [k1]{z}, xmm2, ymm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 10 /r VPSRLVW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 10 /r VPSRLVW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>B</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 45 /r VPSRLVW xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 45 /r VPSRLVW ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 45 /r VPSRLVW zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 45 /r VPSRLVQ zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (w)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
**Description**

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

**VEX.128 encoded version:** The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

**VEX.256 encoded version:** The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

**EVEX encoded VPSRLVD/Q:** The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

**EVEX encoded VPSRLVW:** The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation**

**VPSRLVW (EVEX encoded version)**

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
\text{ i := } j \times 16 \\
\text{ IF } k1[j] \text{ OR } \text{no writemask*} \\
\quad \text{THEN } \text{DEST}[i+15:i] := \text{ZeroExtend}(\text{SRC1}[i+15:i] >> \text{SRC2}[i+15:i]) \\
\quad \text{ELSE} \\
\quad \quad \text{IF } \text{merging-masking} \quad ; \text{merging-masking} \\
\quad \quad \quad \text{THEN } \text{DEST}[i+15:i] \text{ remains unchanged*} \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{zeroing-masking} \\
\quad \quad \quad \text{DEST}[i+15:i] := 0 \\
\quad \text{FI} \\
\text{FI}; \\
\text{ENDFOR}; \\
\text{DEST}[\text{MAXVL-1:VL}] := 0;
\]

**VPSRLVD (VEX.128 version)**

\[
\text{COUNT}_0 := \text{SRC2}[31 : 0] \\
\quad \text{(* Repeat Each COUNT}_i \text{ for the 2nd through 4th dwords of SRC2*)} \\
\text{COUNT}_3 := \text{SRC2}[127 : 96]; \\
\text{IF } \text{COUNT}_0 < 32 \text{ THEN} \\
\quad \text{DEST}[31:0] := \text{ZeroExtend}(\text{SRC1}[31:0] >> \text{COUNT}_0); \\
\text{ELSE} \\
\quad \text{DEST}[31:0] := 0; \\
\quad \text{(* Repeat shift operation for 2nd through 4th dwords *)} \\
\text{IF } \text{COUNT}_3 < 32 \text{ THEN} \\
\quad \text{DEST}[127:96] := \text{ZeroExtend}(\text{SRC1}[127:96] >> \text{COUNT}_3); \\
\text{ELSE} \\
\quad \text{DEST}[127:96] := 0; \\
\text{DEST}[\text{MAXVL-1:128}] := 0;
\]
VPSRLVD (VEX.256 version)
COUNT_0 := SRC2[31 : 0];
  (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
  DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
  DEST[31:0] := 0;
  (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
  DEST[255:224] := ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
  DEST[255:224] := 0;
ENDFOR;
DEST[MAXVL-1:256] := 0;

VPSRLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
      ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;

VPSRLVQ (VEX.128 version)
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
  DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
  DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
  DEST[127:64] := ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
  DEST[127:64] := 0;
DEST[MAXVL-1:128] := 0;
VPSRLVQ (VEX.256 version)
COUNT_0 := SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
  DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
  DEST[63:0] := 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
  DEST[255:192] := ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
  DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;

VPSRLVQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
      ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_srlv_epi32(__m256i m, __m256i count);
VPSRLWD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLWD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLWD __m512i _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64(__m256i m, __m256i count);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded VPSRLVD/Q, see Table 2-49, “Type E4 Class Exception Conditions.”
EVEX-encoded VPSRLVW, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2, and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2, and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Bitwise ternary logic taking zmm1, zmm2, and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2, and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2, and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Bitwise ternary logic taking zmm1, zmm2, and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second, and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.
Table 5-22 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

<table>
<thead>
<tr>
<th>Bit(reg1)</th>
<th>Bit(reg2)</th>
<th>Bit(src3)</th>
<th>Bit Result with Imm8=0xE2</th>
<th>Bit(reg1)</th>
<th>Bit(reg2)</th>
<th>Bit(src3)</th>
<th>Bit Result with Imm8=0xE4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-11 and Table 5-12 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

**Operation**

VPTERNLOGD (EVEX encoded versions)

 KL, VL = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

 i := j * 32

IF k1[j] OR *no writemask*

 THEN

 FOR k := 0 TO 31

 IF (EVEX.b = 1) AND (SRC2 "is memory")

 THEN DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]

 ELSE DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]

 FI;

 ELSE

 IF *merging-masking* ; merging-masking

 THEN *DEST[31+i][i] remains unchanged*

 ELSE ; zeroing-masking

 DEST[31+i][i] := 0

 FI;

 END;

 ENDFOR;

 DEST[MAXVL-1:VL] := 0
VPTERNLOGQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    FOR k := 0 TO 63
      IF (EVEX.b = 1) AND (SRC2 *is memory*) THEN
        DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
      ELSE
        DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
      FI;
    FI;
  ELSE
    IF *merging-masking* THEN
      *DEST[63+i:i] remains unchanged*
    ELSE
      DEST[63+i:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents
VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i __mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i __mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i __mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i __mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i __mm128_mask_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i __mm128_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i __mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i __mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i __mm256_mask_ternarylogic_epi64(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i __mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i __mm128_mask_ternarylogic_epi64(__mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i __mm128_ternarylogic_epi64(__m128i a, __m128i b, int imm);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, “Type E4 Class Exception Conditions.”
### VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 26 /r VPTESTMB k2 [k1], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 26 /r VPTESTMB k2 [k1], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 26 /r VPTESTMB k2 [k1], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 26 /r VPTESTMW k2 [k1], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 26 /r VPTESTMW k2 [k1], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 26 /r VPTESTMW k2 [k1], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 27 /r VPTESTMD k2 [k1], xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 27 /r VPTESTMD k2 [k1], ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 27 /r VPTESTMD k2 [k1], zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 27 /r VPTESTMQ k2 [k1], xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 27 /r VPTESTMQ k2 [k1], ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 27 /r VPTESTMQ k2 [k1], zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)

\[(KL, VL) = (16, 128), (32, 256), (64, 512)\]

\[
\begin{align*}
\text{FOR } j & : 0 \text{ TO } KL-1 \\
\quad i & := j * 8 \\
\quad \text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\qquad \text{THEN } & \text{DEST}[j] := (\text{SRC1}[i+7:i] \text{ BITWISE AND SRC2}[i+7:i] \neq 0)? 1 : 0; \\
\qquad \text{ELSE } & \text{DEST}[j] = 0; \text{ zeroing-masking only} \\
\quad \text{FI;} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX}_K L-1:KL] := 0
\end{align*}
\]

VPTESTMW (EVEX encoded versions)

\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[
\begin{align*}
\text{FOR } j & : 0 \text{ TO } KL-1 \\
\quad i & := j * 16 \\
\quad \text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\qquad \text{THEN } & \text{DEST}[j] := (\text{SRC1}[i+15:i] \text{ BITWISE AND SRC2}[i+15:i] \neq 0)? 1 : 0; \\
\qquad \text{ELSE } & \text{DEST}[j] = 0; \text{ zeroing-masking only} \\
\quad \text{FI;} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX}_K L-1:KL] := 0
\end{align*}
\]

VPTESTMD (EVEX encoded versions)

\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]

\[
\begin{align*}
\text{FOR } j & : 0 \text{ TO } KL-1 \\
\quad i & := j * 32 \\
\quad \text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\qquad \text{THEN} \\
\qquad \quad \text{IF (EVEX.b = 1) AND (SRC2 *is memory*)} \\
\qquad \qquad \text{THEN } & \text{DEST}[j] := (\text{SRC1}[i+31:i] \text{ BITWISE AND SRC2}[31:0] \neq 0)? 1 : 0; \\
\qquad \qquad \text{ELSE } & \text{DEST}[j] := (\text{SRC1}[i+31:i] \text{ BITWISE AND SRC2}[i+31:i] \neq 0)? 1 : 0; \\
\qquad \quad \text{FI;} \\
\qquad \text{ELSE } & \text{DEST}[j] = 0; \text{ zeroing-masking only} \\
\quad \text{FI;} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX}_K L-1:KL] := 0
\end{align*}
\]
VPTESTMQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
      ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
    FI;
  ELSE DEST[j] := 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB __mmask64 _mm512_test_epi8_mask( __m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask( __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask( __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPTESTMD/Q: See Table 2-49, "Type E4 Class Exception Conditions."
VPTESTMB/W: See Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."
## VPTESTNMB/W/D/Q—Logical NAND and Set

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512BW</td>
<td>Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512BW</td>
<td>Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
</tbody>
</table>
Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Operation

VPTESTNMB

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1
  i := j*8
  IF MaskBit(j) OR *no writemask*
    THEN
      DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
    ELSE DEST[j] := 0; zeroing masking only
  FI
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPTESTNMw

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j*16
  IF MaskBit(j) OR *no writemask*
    THEN
      DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
    ELSE DEST[j] := 0; zeroing masking only
  FI
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPTESTNMD

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j*32
  IF MaskBit(j) OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[i+31:i] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
      ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0
    FI
ENDFOR

VPTESTNMB/W/D/Q—Logical NAND and Set
ELSE DEST[j] := 0; zeroing masking only
FI
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPTESTNMQ
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j*64
  IF MaskBit(j) OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] == 0)? 1 : 0;
        ELSE DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] == 0)? 1 : 0;
      FI;
    ELSE DEST[j] := 0; zeroing masking only
  FI
ENDFOR
DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPTESTNMB __mmask64 _mm512_testn_epi8_mask( __m512i a, __m512i b);
VPTESTNMB __mmask64 _mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 _mm256_testn_epi8_mask( __m256i a, __m256i b);
VPTESTNMB __mmask32 _mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 _mm128_testn_epi8_mask( __m128i a, __m128i b);
VPTESTNMB __mmask16 _mm128_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 _mm512_testn_epi16_mask( __m512i a, __m512i b);
VPTESTNMW __mmask32 _mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 _mm256_testn_epi16_mask( __m256i a, __m256i b);
VPTESTNMW __mmask16 _mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 _mm128_testn_epi16_mask( __m128i a, __m128i b);
VPTESTNMW __mmask8 _mm128_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 _mm512_testn_epi32_mask( __m512i a, __m512i b);
VPTESTNMD __mmask16 _mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 _mm256_testn_epi32_mask( __m256i a, __m256i b);
VPTESTNMD __mmask8 _mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 _mm128_testn_epi32_mask( __m128i a, __m128i b);
VPTESTNMD __mmask8 _mm128_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 _mm512_testn_epi64_mask( __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm256_testn_epi64_mask( __m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm128_testn_epi64_mask( __m128i a, __m128i b);
VPTESTNMQ __mmask8 _mm128_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
VPTESTNMD/VPTESTNMQ: See Table 2-49, “Type E4 Class Exception Conditions.”
VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”
VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

### Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

---

**Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS**

---
When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-23. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-23.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-24.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-25.

### Table 5-23. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
<th>IE Signaling Due to Comparison</th>
<th>Imm8[3:2] Effect to Range Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sNaN1</td>
<td>sNaN2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>sNaN1</td>
<td>qNaN2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>sNaN1</td>
<td>Norm2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>qNaN1</td>
<td>sNaN2</td>
<td>Quiet(sNaN2)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>qNaN1</td>
<td>qNaN2</td>
<td>qNaN1</td>
<td>No</td>
<td>Applicable</td>
</tr>
<tr>
<td>qNaN1</td>
<td>Norm2</td>
<td>Norm2</td>
<td>No</td>
<td>Applicable</td>
</tr>
<tr>
<td>Norm1</td>
<td>sNaN2</td>
<td>Quiet(sNaN2)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>Norm1</td>
<td>qNaN2</td>
<td>Norm1</td>
<td>No</td>
<td>Applicable</td>
</tr>
</tbody>
</table>

### Table 5-24. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS, and MAX, MAX_ABS

<table>
<thead>
<tr>
<th>MIN and MIN_ABS</th>
<th>MAX and MAX_ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src2</td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
</tr>
</tbody>
</table>

### Table 5-25. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)

| MIN_ABS (|a| = |b|, a>0, b<0) | MAX_ABS (|a| = |b|, a>0, b<0) |
|----------------|----------------|
| Src1 | Src2 | Result | Src1 | Src2 | Result |
| a | b | b | a | b | a |
| b | a | b | a | b | a |

**Operation**

```c
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-23
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[62:52];
    Src1.fraction := SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
```
Src2.exp := SRC2[62:52];
Src2.fraction := SRC2[51:0];
IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN// Src2 is a denormal number
  IF DAZ THEN Src2.fraction := 0;
  ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN[TMP[63:0] := SRC1[63:0]]
ELSE IF(SRC1 = QNAN) THEN[TMP[63:0] := SRC2[63:0]]
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] := from Table 5-24
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-25
ELSE
  Case(CmpOpCtl[1:0])
  00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
  01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
  10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
  11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
  ESAC;
FI;

Case(SignSelCtl[1:0])
  00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
  01: dest := TMP[63:0];// Preserve sign of compare result
  10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
  11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
  ESAC;
RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

VRANGEPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
      ELSE DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] = 0
      FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

The following example describes a common usage of this instruction for checking that the input operand is
bounded between ±1023.
VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:

- zmm_dst is the destination operand.
- zmm_src is the input operand to compare against ±1023 (this is SRC1).
- zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).
- IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e., SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d __m512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d __m512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d __m512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d __m512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d __m512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d __m512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d __mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d __mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d __mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d __mm128_range_pd (__m128 a, __m128 b, int imm);
VRANGEPD __m128d __mm128_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d __mm128_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”
VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.

- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-23. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-23.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-24.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-25.
VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values

INSTRUCTION SET REFERENCE, V

Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])

// Check if SNAN and report IE, see also Table 5-23
IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

Src1.exp := SRC1[30:23];
Src1.fraction := SRC1[22:0];
IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN // Src1 is a denormal number
  IF DAZ THEN Src1.fraction := 0;
  ELSE IF (SRC2 <> QNAN) Set DE; FI;
FI;
Src2.exp := SRC2[30:23];
Src2.fraction := SRC2[22:0];
IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN // Src2 is a denormal number
  IF DAZ THEN Src2.fraction := 0;
  ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[31:0] := SRC1[31:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] := SRC2[31:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] := from Table 5-24
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-25
ELSE
  Case(CmpOpCtl[1:0])
  00: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
  01: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
  10: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
  11: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
  ESAC;
FI;
Case(SignSelCtl[1:0])
00: dest := (SRC1[31] << 31) OR (TMP[30:0]) // Preserve Src1 sign bit
01: dest := TMP[31:0] // Preserve sign of compare result
10: dest := (0 << 31) OR (TMP[30:0]) // Zero out sign bit
11: dest := (1 << 31) OR (TMP[30:0]) // Set the sign bit
ESAC;
RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

VRANGEPS
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
    FI;
The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

```
VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:
- zmm_dst is the destination operand.
- zmm_src is the input operand to compare against ±150.
- zmm_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VRANGEPS __m512 _mm512_range_ps ( __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_range_round_ps ( __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_mask_range_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_mask_range_round_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_maskz_range_ps ( __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_maskz_range_round_ps ( __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 _mm256_range_ps ( __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_mask_range_ps ( __m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_maskz_range_ps ( __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 _mm128_range_ps ( __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm128_mask_range_ps ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm128_maskz_range_ps ( __mmask8 k, __m128 a, __m128 b, int imm);
```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Table 2-46, "Type E2 Class Exception Conditions."
VRANGESD—Range Restriction Calculation From a Pair of Scalar Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W1 S1 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate a RANGE operation output value from 2 double precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

This instruction calculates a range operation output from two input double precision floating-point values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-23. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-23.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-24.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-25.
Operation

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
   // Check if SNAN and report IE, see also Table 5-23
   IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
   IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

   Src1.exp := SRC1[62:52];
   Src1.fraction := SRC1[51:0];
   IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN // Src1 is a denormal number
      IF DAZ THEN Src1.fraction := 0;
      ELSE IF (SRC2 <> QNAN) Set DE; FI;
   FI;

   Src2.exp := SRC2[62:52];
   Src2.fraction := SRC2[51:0];
   IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN // Src2 is a denormal number
      IF DAZ THEN Src2.fraction := 0;
      ELSE IF (SRC1 <> QNAN) Set DE; FI;
   FI;

   IF (SRC2 = QNAN) THEN
      TMP[63:0] := SRC1[63:0]
   ELSE IF (SRC1 = QNAN) THEN
      TMP[63:0] := SRC2[63:0]
   ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed)
      TMP[63:0] := from Table 5-24
   ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01)
      TMP[63:0] := from Table 5-25
   ELSE
      Case(CmpOpCtl[1:0])
         00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
         01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
         10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
         11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
      ESAC;
   FI;

   Case(SignSelCtl[1:0])
      00: dest := (SRC1[63] << 63) OR (TMP[62:0]); // Preserve Src1 sign bit
      01: dest := TMP[63:0]; // Preserve sign of compare result
      10: dest := (0 << 63) OR (TMP[62:0]); // Zero out sign bit
      11: dest := (1 << 63) OR (TMP[62:0]); // Set the sign bit
   ESAC;
   RETURN dest[63:0];
}

CmpOpCtl[1:0] = imm8[1:0];
SignSelCtl[1:0] = imm8[3:2];
VRANGESD

IF k1[0] OR *no writemask*
    THEN DEST[63:0] := RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*  
        ELSE ; zeroing-masking
            DEST[63:0] = 0
    FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±1023.
xmm_1023 is the reference operand, contains the value of 1023.
IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).
However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.
Thus, if xmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent
VRANGESD __m128d _mm_range_sd (__m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd (__m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W0 51 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction calculates a range operation output from two input single-precision floating-point values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.

- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand. When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NaN is listed in Table 5-23. If the comparison raises an IE, the sign select control (imm8[3:2]) has no effect to the range operation output; this is indicated also in Table 5-23.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-24.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABSENT or MAX_ABSENT comparison operation with result listed in Table 5-25.
Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-23
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp := SRC1[30:23];
    Src1.fraction := SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction := 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp := SRC2[30:23];
    Src2.fraction := SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction := 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[31:0] := SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] := SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] := from Table 5-24
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-25
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] := (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] := (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest := TMP[31:0];// Preserve sign of compare result
    10: dest := (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest := (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGESS
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged* ; zeroing-masking
            DEST[31:0] = 0
    FI;
FI;
DEST[MAXVL-1:128] := 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±150.
xmm_150 is the reference operand, contains the value of 150.
IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).
However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.
Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent
VRANGESS __m128 _mm_range_ss ( __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal.

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 4C / r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 4C / r VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 4C / r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocals of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand. The maximum relative error for this approximation is less than $2^{-14}$.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a $0.0$, $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a $0.0$). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-26. VRCP14PD/VRCP14SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq X \leq 2^{-1024}$</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$-2^{-1024} \leq X \leq -0$</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$X &gt; 2^{1022}$</td>
<td>Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>$X &lt; -2^{1022}$</td>
<td>-Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>$X = 2^n$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X = -2^{-n}$</td>
<td>$-2^n$</td>
<td></td>
</tr>
</tbody>
</table>

* in this case the mantissa is shifted right by one or two bits

**Operation**

VRCP14PD ((EVEX encoded versions))

 KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] := APPROXIMATE(1.0/SRC[63:0]);
      ELSE DEST[i+63:i] := APPROXIMATE(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PD __m512d _mm512_rcp14_pd( __m512d a);
VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);
VRCP14PD __m512d _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal of the scalar double precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-26 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


Operation

**VRCP14SD (EVEX version)**

IF k1[0] OR *no writemask*

    THEN DEST[63:0] := APPROXIMATE(1.0/SRC2[63:0]);

ELSE

    IF *merging-masking* ; merging-masking

        THEN *DEST[63:0] remains unchanged* 

    ELSE ; zeroing-masking

        DEST[63:0] := 0

    FI;

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd( __m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd( __mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-51, "Type E5 Class Exception Conditions."
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

VRCP14PS is a SIMD instruction that computes the approximate reciprocals of the packed single-precision floating-point values in the source operand and stores the results in the destination operand. The maximum relative error for this approximation is less than 2^-14.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask. The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is 0.0, an infinity with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:

Table 5-27. VRCP14PS/VRCP14SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ X ≤ 2^-128</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>-2^-128 ≤ X ≤ -0</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
</tbody>
</table>
| X > 2^126 | Underflow | Up to 18 bits of fractions are returned
| X < -2^126 | -Underflow | Up to 18 bits of fractions are returned
| X = 2^n | 2^n | |
| X = -2^n | -2^n | |

NOTES:
1. In this case, the mantissa is shifted right by one or two bits.
Operation

**VRCP14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] := APPROXIMATE(1.0/SRC[31:0]);
      ELSE DEST[i+31:i] := APPROXIMATE(1.0/SRC[i+31:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 _mm512_rcp14_ps( __m512 a);
VRCP14PS __m512 _mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 _mm512_maskz_rcp14_ps( __mmask16 k, __m512 a);
VRCP14PS __m256 _mm256_rcp14_ps( __m256 a);
VRCP14PS __m256 _mm256_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 _mm256_maskz_rcp14_ps( __mmask8 k, __m256 a);
VRCP14PS __m128 _mm128_rcp14_ps( __m128 a);
VRCP14PS __m128 _mm128_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 _mm128_maskz_rcp14_ps( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”
**VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 4D /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

| Op/En | Tuple Type |_operand 1_operation|_operand 2_operation|_operand 3_operation|Operand 4| |---|---|---|---|---|
|-------|------------|-------------------|-------------------|-------------------|---------|
| A     | Tuple1 Scalar | ModRMreg (w) | EVEX.vvvv (r) | ModRMbr/m (r) | N/A |

**Description**

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-27 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


**Operation**

**VRCP14SS (EVEX version)**

IF k1[0] OR *no writemask*

THEN DEST[31:0] := APPROXIMATE(1.0/SRC2[31:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged* ; zeroing-masking

ELSE ; zeroing-masking

DEST[31:0] := 0

FI;


DEST[MAXVL-1:128] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

- `VRCP14SS __m128 _mm_rcp14_ss(__m128 a, __m128 b);`
- `VRCP14SS __m128 _mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);`
- `VRCP14SS __m128 _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-51, “Type E5 Class Exception Conditions.”
INSTRUCTION SET REFERENCE, V

VRCPPH—Compute Reciprocals of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.MAP6.W0 4C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compute the approximate reciprocals of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VRCPPH xmm1<a href="z">k1</a>, xmm2/m128/ m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.MAP6.W0 4C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compute the approximate reciprocals of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VRCPPH ymm1<a href="z">k1</a>, ymm2/m256/ m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.MAP6.W0 4C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compute the approximate reciprocals of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VRCPPH zmm1<a href="z">k1</a>, zmm2/m512/ m16bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocals of 8/16/32 packed FP16 values in the source operand (the second operand) and stores the packed FP16 results in the destination operand. The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

For special cases, see Table 5-28.

**Table 5-28. VRCPPH/VRCPSH Special Cases**

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq X \leq 2^{-16}$</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$-2^{-16} \leq X \leq -0$</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$X &gt; +\infty$</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>$X &lt; -\infty$</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>$X = 2^{-n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X = -2^{-n}$</td>
<td>$-2^n$</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

VRCPPH dest[k1]. src
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF SRC is memory and (EVEX.b = 1):
      tsrc := src.fp16[0]
    ELSE:
      tsrc := src.fp16[i]
    DEST.fp16[i] := APPROXIMATE(1.0 / tsrc)
  ELSE IF *zeroing*:
    DEST.fp16[i] := 0
  //else DEST.fp16[i] remains unchanged
  DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VRCPH __m128h _mm_mask_rcp_ph (__m128h src, __mmask8 k, __m128h a);

VRCPHP __m128h _mm_maskz_rcp_ph (__mmask8 k, __m128h a);

VRCPH __m128h _mm_rcp_ph (__m128h a);

VRCPHP __m256h _mm256_mask_rcp_ph (__m256h src, __mmask16 k, __m256h a);

VRCPHP __m256h _mm256_maskz_rcp_ph (__mmask16 k, __m256h a);

VRCPH __m256h _mm256_rcp_ph (__m256h a);

VRCPHP __m512h _mm512_mask_rcp_ph (__m512h src, __mmask32 k, __m512h a);

VRCPHP __m512h _mm512_maskz_rcp_ph (__mmask32 k, __m512h a);

VRCPH __m512h _mm512_rcp_ph (__m512h a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VRCPSH—Compute Reciprocal of Scalar FP16 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 4D /r VRCPSH xmm1[k1]{z}, xmm2, xmm3/ m16</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compute the approximate reciprocal of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand (the second operand).</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocal of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1. Bits 127:16 of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For special cases, see Table 5-28.

**Operation**

VRCPSH dest{k1}, src1, src2

IF k1[0] or *no writemask*:

\[
\text{DEST.fp16}[0] := \text{APPROXIMATE}(1.0 / \text{src2.fp16}[0])
\]

ELSE IF *zeroing*:

\[
\text{DEST.fp16}[0] := 0
\]

//else DEST.fp16[0] remains unchanged

\[
\text{DEST}[127:16] := \text{src1}[127:16]
\]

\[
\text{DEST}[,\text{MAXVL}-1:128] := 0
\]

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCPSH _m128h __mm_mask_rcp_sh (_m128h src, __mmask8 k, _m128h a, _m128h b);
VRCPSH _m128h __mm_maskz_rcp_sh (_mmask8 k, _m128h a, _m128h b);
VRCPSH _m128h __mm_rcp_sh (_m128h a, _m128h b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 56 / r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed double precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 56 / r ib VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed double precision floating-point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 56 / r ib VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform reduction transformation on packed double precision floating-point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Perform reduction transformation of the packed binary encoded double precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M\times\text{src}))\times2^{-M};
\]

where “Round()” treats “src”, “2^M×r”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p×man2, where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if RC = RNE: 0 <= |Reduced Result| <= 2^p-M-1

Then if RC ≠ RNE: 0 < |Reduced Result| < 2^p-M

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS**

VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values
Handling of special case of input values are listed in Table 5-29.

<table>
<thead>
<tr>
<th>Table 5-29. VREDUCEPD/SD/PS/SS Special Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>RNE</td>
</tr>
<tr>
<td>[Src1] &lt; 2^{-M}</td>
</tr>
<tr>
<td>[Src1] &lt; 2^{-M}</td>
</tr>
<tr>
<td>[Src1] ≤ 0</td>
</tr>
<tr>
<td>[Src1] ≥ 0</td>
</tr>
<tr>
<td>Src1 = ±0, or Dest = ±0 (Src1≠±INF)</td>
</tr>
<tr>
<td>Src1 = ±INF</td>
</tr>
<tr>
<td>Src1 = ±NAN</td>
</tr>
</tbody>
</table>

* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

**Operation**

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```c
// Check for NaN
IF (SRC[63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
RC := imm8[1:0]; // Round Control for ROUND() operation
RC source := imm[2];
SPE := imm[3]; // Suppress Precision Exception
TMP[63:0] := 2-M *ROUND(2M*SRC[63:0], SPE, RC_source, RC); // ROUND() treats SRC and 2^M as standard binary FP values
TMP[63:0] := SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
```

**VREducePD**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b == 1) AND (SRC *is memory*)
            THEN DEST[i+63:j] := ReduceArgumentDP(SRC[63:0], imm8[7:0]);
            ELSE DEST[i+63:j] := ReduceArgumentDP(SRC[i+63:j], imm8[7:0]);
            FI;
    ELSE
        IF *merging-masking*; merging-masking
            THEN *DEST[i+63:j] remains unchanged*;
        ELSE ; zeroing-masking
            DEST[i+63:j] = 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPD __m512d __mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREDUCEPD __m512d __mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d __mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d __mm256_mask_reduce_pd(__m256d a, int imm)
VREDUCEPD __m256d __mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d __mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d __mm_mask_reduce_pd(__m128d a, int imm)
VREDUCEPD __m128d __mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d __mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD  If EVEX.vvvv != 1111B.
VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.0F3A.W0 56 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Perform reduction transformation on packed FP16 values in xmm2/m128/m16bcst, imm8</td>
</tr>
<tr>
<td>EVEX.256.NP.0F3A.W0 56 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Perform reduction transformation on packed FP16 values in ymm2/m256/m16bcst, imm8</td>
</tr>
<tr>
<td>EVEX.512.NP.0F3A.W0 56 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Perform reduction transformation on packed FP16 values in zmm2/m512/m16bcst, imm8</td>
</tr>
</tbody>
</table>

Description

This instruction performs a reduction transformation of the packed binary encoded FP16 values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is an unsigned integer specified by imm8[7:4]. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - \left(\text{ROUND}(2^M \times \text{src})\right) \times 2^{-M}
\]

where ROUND() treats src, \(2^M\), and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src = \(2^p \times \text{man2}\), where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if RC=RNE: \(0 \leq |\text{ReducedResult}| \leq 2^{-M-1}\).

Then if RC \(\neq\) RNE: \(0 \leq |\text{ReducedResult}| < 2^{-M}\).

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-30.
**Table 5-30. VREDECPH/VREDUCESH Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Round Mode</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>\text{Src1}</td>
<td>&lt; 2^{-M-1})</td>
</tr>
<tr>
<td>(</td>
<td>\text{Src1}</td>
<td>&lt; 2^{-M})</td>
</tr>
<tr>
<td></td>
<td>RU, (\text{Src1} \leq 0)</td>
<td>(\text{Src1})</td>
</tr>
<tr>
<td></td>
<td>RD, (\text{Src1} \geq 0)</td>
<td>(\text{Src1})</td>
</tr>
<tr>
<td></td>
<td>RD, (\text{Src1} &lt; 0)</td>
<td>(\text{Round}(\text{Src1} + 2^{-M}))</td>
</tr>
<tr>
<td>(\text{Src1} = \pm 0) or</td>
<td>NOT RD</td>
<td>+0.0</td>
</tr>
<tr>
<td>(\text{Dest} = \pm 0) ((\text{Src1} \neq \infty))</td>
<td>RD</td>
<td>-0.0</td>
</tr>
<tr>
<td>(\text{Src1} = \pm \infty)</td>
<td>Any</td>
<td>+0.0</td>
</tr>
<tr>
<td>(\text{Src1} = \pm \text{NAN})</td>
<td>Any</td>
<td>QNaN ((\text{Src1}))</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The \(\text{Round}()\) function uses rounding controls specified by \((\text{imm8}[2]? \text{MXCSR.RC}: \text{imm8}[1:0])\).

**Operation**

```python
def reduce_fp16(src, imm8):
    nan := (src.exp = 0x1F) and (src.fraction != 0)
    if nan:
        return QNAN(src)
    m := imm8[7:4]
    rc := imm8[1:0]
    rc_source := imm8[2]
    tmp := \(2^{-m} \times \text{ROUND}(2^{m} \times \text{src}, \text{spe}, \text{rc} \_\text{source}, \text{rc})\)
    tmp := src - tmp // using same RC, SPE controls
    return tmp
```

VREDUCEPH dest\(k1\), src, imm8

VL = 128, 256 or 512
KL := VL/16

FOR \(i := 0\) to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
        DEST.fp16[i] := reduce_fp16(tsrc, imm8)
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPH __m128h _mm_mask_reduce_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_maskz_reduce_ph (__mmask8 k, __m128h a, int imm8);
VREDUCEPH __m128h _mm_reduce_ph (__m128h a, int imm8);
VREDUCEPH __m256h _mm256_mask_reduce_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_maskz_reduce_ph (__mmask16 k, __m256h a, int imm8);
VREDUCEPH __m256h _mm256_reduce_ph (__m256h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_maskz_reduce_ph (__mmask32 k, __m512h a, int imm8);
VREDUCEPH __m512h _mm512_reduce_ph (__m512h a, int imm8);
VREDUCEPH __m512h _mm512_mask_reduce_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_maskz_reduce_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VREDUCEPH __m512h _mm512_reduce_round_ph (__m512h a, int imm8, const int sae);

SIMD Floating-Point Exceptions
Invalid, Precision.

Other Exceptions
EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions."
VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 56 / r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 56 / r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating-point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 56 / r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating-point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Perform reduction transformation of the packed binary encoded single-precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point format to the destination operand (the first operand) under writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M};
\]

where “Round()” treats “src”, “2^M”, and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2, where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if RC = RNE: \(0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}\)

Then if RC ≠ RNE: \(0 < |\text{Reduced Result}| < 2^{p-M}\)

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-29.
Operation

\[
\text{ReduceArgumentSP}(\text{SRC}[31:0], \text{imm8}[7:0])
\]

\[
\]

// Check for NaN
\[
\]
\[
\]
\[
\]
\[
\]
VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W1 57</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform a reduction transformation on a scalar double precision floating-point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Perform a reduction transformation of the binary encoded double precision floating-point value in the low qword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[ \text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M}; \]

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2, where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0 <= |Reduced Result| <= 2^{p-M-1}

Then if RC ≠ RNE: 0 < |Reduced Result| < 2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-29.

**Operation**

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```c
{ 
  // Check for NaN
  IF (SRC[63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); Fl;
  M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC := imm8[1:0]; // Round Control for ROUND() operation
  RC source := imm[2];
  SPE := imm[3]; // Suppress Precision Exception
  TMP[63:0] := 2^M × ROUND(2^M×SRC[63:0], SPE, RC_source, RC); // ROUND() treats SRC and 2^M as standard binary FP values
  TMP[63:0] := SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value
VREDUCED
IF k1[0] or *no writemask*
    THEN DEST[63:0] := ReduceArgumentDP(SRC2[63:0], imm8[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0] = 0
    FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VREDUCED __m128d _mm_mask_reduce_sd(__m128d a, __m128d b, int imm, int sae)
VREDUCED __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCED __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions
Invalid, Precision.
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP:0F3A.W0 57 /r /ib VREDUCESH xmm1{k1}[12][2], xmm2, xmm3/m16 (sae), imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Perform a reduction transformation on the low binary encoded FP16 value in xmm3/m16 by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

Description

This instruction performs a reduction transformation of the low binary encoded FP16 value in the source operand (the second operand) and store the reduced result in binary FP format to the low element of the destination operand (the first operand) under the writemask k1. For further details see the description of VREDUCEPH. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-30.

Operation

VREDUCESH dest{k1}, src, imm8

IF k1[0] or *writemask*:
  dest.fp16[0] := reduce_fp16(src2.fp16[0], imm8)  // see VREDUCEPH
ELSE IF *zeroing*:
  dest.fp16[0] := 0
//else dest.fp16[0] remains unchanged

DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESH __m128h _mm_mask_reduce_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_maskz_reduce_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_reduce_round_sh (__m128h a, __m128h b, int imm8, const int sae);
VREDUCESH __m128h _mm_mask_reduce_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_maskz_reduce_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VREDUCESH __m128h _mm_reduce_sh (__m128h a, __m128h b, int imm8);

SIMD Floating-Point Exceptions

Invalid, Precision.
Other Exceptions
EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
**VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W0 57 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform a reduction transformation on a scalar single-precision floating-point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Perform a reduction transformation of the binary encoded single-precision floating-point value in the low dword element of the second source operand (the third operand) and store the reduced result in binary floating-point format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary floating-point source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M};
\]

where "Round()" treats "src", "2^M", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2, where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-29.

**Operation**

ReduceArgumentSP(SRC[31:0], imm8[7:0])

{  
  // Check for NaN
  if (SRC[31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC := imm8[1:0]; // Round Control for ROUND() operation
  RC source := imm[2];
  SPE := imm[3]; // Suppress Precision Exception
  TMP[31:0] := 2^M * ROUND(2^M * SRC[31:0], SPE, RC, source, RC); // ROUND() treats SRC and 2^M as standard binary FP values
  TMP[31:0] := SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
VREDUCESS
IF k1[0] or *no writemask*
    THEN DEST[31:0] := ReduceArgumentSP(SRC2[31:0], imm8[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[31:0] = 0
FI;
FL;
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VREDUCESS __m128_mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128_mm_mask_reduce_ss( __m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128_mm_maskz_reduce_ss( __mmask16 k, __m128 a, __m128 b, int imm, int sae)

SIMD Floating-Point Exceptions
Invalid, Precision.
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-47, "Type E3 Class Exception Conditions."
VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed double precision floating-point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 09 /r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed double precision floating-point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds packed double precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Round the double precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

\[ \text{ROUND} (x) = 2^{-M} \times \text{Round to INT} (x \times 2^M, \text{round_ctrl}), \]

\[ \text{round_ctrl} = \text{imm8}[3:0] ; \]

\[ M = \text{imm8}[7:4] ; \]

The operation of \( x \times 2^M \) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).
VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round}_\to_{\text{INT}}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Handling of special cases

Handling of special case of input values are listed in Table 5-31.

<table>
<thead>
<tr>
<th>Src1 = ±inf</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1 = ±NAN</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1 converted to QNAN</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1 = ±0</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

### Operation

\[
\text{RoundToIntegerDP}(\text{SRC}[63:0], \text{imm8}[7:0]) \{
\]

\[
\text{if} (\text{imm8}[2] = 1) \\
\text{rounding\_direction := MXCSR:RC} ; \text{get round control from MXCSR}
\]

\[
\text{else} \\
\text{rounding\_direction := imm8[1:0]} ; \text{get round control from imm8[1:0]}
\]

\[
\text{FI}
\]

\[
M := \text{imm8}[7:4] ; \text{get the scaling factor}
\]

\[
\text{case (rounding\_direction)}
\]

\[
\begin{align*}
00 & : \text{TMP}[63:0] := \text{round\_to\_nearest\_even\_integer}(2^M \times \text{SRC}[63:0]) \\
01 & : \text{TMP}[63:0] := \text{round\_to\_equal\_or\_smaller\_integer}(2^M \times \text{SRC}[63:0]) \\
10 & : \text{TMP}[63:0] := \text{round\_to\_equal\_or\_larger\_integer}(2^M \times \text{SRC}[63:0]) \\
11 & : \text{TMP}[63:0] := \text{round\_to\_nearest\_smallest\_magnitude\_integer}(2^M \times \text{SRC}[63:0])
\end{align*}
\]

\[
\text{ESAC}
\]

\[
\text{Dest}[63:0] := 2^M \times \text{TMP}[63:0] ; \text{scale down back to } 2^M
\]

\[
\text{if} (\text{imm8}[3] = 0) \text{ Then} ; \text{check SPE}
\]

\[
\text{if} (\text{SRC}[63:0] \neq \text{Dest}[63:0]) \text{ Then} ; \text{check precision lost}
\]

\[
\text{set\_precision()} ; \text{set #PE}
\]

\[
\text{FI};
\]

\[
\text{FI};
\]

---

Table 5-31. VRNDSCALEPD/SD/PS/SS Special Cases

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1 converted to QNAN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src1</td>
</tr>
</tbody>
</table>

---

Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS
VRNDSCALEPD—Round Packed Float64 Values to Include a Given Number of Fraction Bits

VRNDSCALEPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
  THEN TMP_SRC := BROADCAST64(SRC, VL, k1)
  ELSE TMP_SRC := SRC
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_round_pd( __mmask8 k, __m256d a, int imm, int sae);
VRNDSCALEPD __m128d _mm128_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm128_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm128_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm, int sae);
VRNDSCALEPD __m128d _mm128_maskz_roundscale_round_pd( __mmask8 k, __m128d a, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Precision.
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-46, “Type E2 Class Exception Conditions.”
VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.0F3A.W0 08 /r /ib</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Round packed FP16 values in xmm2/m128/m16bcst, imm8</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.NP.0F3A.W0 08 /r /ib</td>
<td>A V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Round packed FP16 values in ymm2/m256/m16bcst, imm8</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.NP.0F3A.W0 08 /r /ib</td>
<td>A V/V</td>
<td>AVX512-FP16</td>
<td>Round packed FP16 values in zmm2/m512/m16bcst, imm8</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction rounds the FP16 values in the source operand by the rounding mode specified in the immediate operand (see Table 5-32) and places the result in the destination operand. The destination operand is conditionally updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as an FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 5-32, “Imm8 Controls for VRNDSCALEPH/VRNDSCALESH.” Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-33.

The formula of the operation on each data element for VRNDSCALEPH is

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_{\text{to} \_ \text{INT}}(x \times 2^M, \text{round} \_ \text{ctrl}),
\]

\[
\text{round} \_ \text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

If this instruction encoding’s SPE bit (bit 3) in the immediate operand is 1, VRNDSCALEPH can set MXCSR.UE without MXCSR.PE.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
### Table 5-32. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH

<table>
<thead>
<tr>
<th>Imm8 Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm8[7:4]</td>
<td>Number of fixed points to preserve.</td>
</tr>
<tr>
<td>imm8[3]</td>
<td>Suppress Precision Exception (SPE)</td>
</tr>
<tr>
<td></td>
<td>0b00: Implies use of MXCSR exception mask.</td>
</tr>
<tr>
<td></td>
<td>0b01: Implies suppress.</td>
</tr>
<tr>
<td>imm8[2]</td>
<td>Round Select (RS)</td>
</tr>
<tr>
<td></td>
<td>0b00: Implies use of imm8[1:0].</td>
</tr>
<tr>
<td></td>
<td>0b01: Implies use of MXCSR.</td>
</tr>
<tr>
<td>imm8[1:0]</td>
<td>Round Control Override:</td>
</tr>
<tr>
<td></td>
<td>0b00: Round nearest even.</td>
</tr>
<tr>
<td></td>
<td>0b01: Round down.</td>
</tr>
<tr>
<td></td>
<td>0b10: Round up.</td>
</tr>
<tr>
<td></td>
<td>0b11: Truncate.</td>
</tr>
</tbody>
</table>

### Table 5-33. VRNDSCALEPH/VRNDSCALESH Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1 = ±∞</td>
<td>Src1</td>
</tr>
<tr>
<td>Src1 = ±NaN</td>
<td>Src1 converted to QNaN</td>
</tr>
<tr>
<td>Src1 = ±0</td>
<td>Src1</td>
</tr>
</tbody>
</table>

**Operation**

```python
def round_fp16_to_integer(src, imm8):
    if imm8[2] == 1:
        rounding_direction := MXCSR.RC
    else:
        rounding_direction := imm8[1:0]
    m := imm8[7:4] // scaling factor

    tsrc1 := 2^m * src

    if rounding_direction == 0b00:
        tmp := round_to_nearest_even_integer(trc1)
    elif rounding_direction == 0b01:
        tmp := round_to_equal_or_smaller_integer(trc1)
    elif rounding_direction == 0b10:
        tmp := round_to_equal_or_larger_integer(trc1)
    else:
        tmp := round_to_smallest_magnitude_integer(trc1)

    dst := 2^(-m) * tmp

    if imm8[3] == 0: // check SPE
        if src != dst:
            MXCSR.PE := 1
    return dst
```

VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits
VRNDScalePH dest(k1), src, imm8
VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF SRC is memory and (EVEX.b = 1):
      tsrc := src.fp16[0]
    ELSE:
      tsrc := src.fp16[i]
    DEST.fp16[i] := round_fp16_to_integer(tsrc, imm8)
  ELSE IF *zeroing*:
    DEST.fp16[i] := 0
  //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**
VRNDScalePH __m128h _mm_mask_roundscale_ph (__m128h src, __mmask8 k, __m128h a, int imm8);
VRNDScalePH __m128h _mm_maskz_roundscale_ph (__mmask8 k, __m128h a, int imm8);
VRNDScalePH __m128h _mm_roundscale_ph (__m128h a, int imm8);
VRNDScalePH __m256h _mm256_mask_roundscale_ph (__m256h src, __mmask16 k, __m256h a, int imm8);
VRNDScalePH __m256h _mm256_maskz_roundscale_ph (__mmask16 k, __m256h a, int imm8);
VRNDScalePH __m256h _mm256_roundscale_ph (__m256h a, int imm8);
VRNDScalePH __m512h _mm512_mask_roundscale_ph (__m512h src, __mmask32 k, __m512h a, int imm8);
VRNDScalePH __m512h _mm512_maskz_roundscale_ph (__mmask32 k, __m512h a, int imm8);
VRNDScalePH __m512h _mm512_roundscale_ph (__m512h a, int imm8);
VRNDScalePH __m512h _mm512_mask_roundscale_round_ph (__m512h src, __mmask32 k, __m512h a, int imm8, const int sae);
VRNDScalePH __m512h _mm512_maskz_roundscale_round_ph (__mmask32 k, __m512h a, int imm8, const int sae);
VRNDScalePH __m512h _mm512_roundscale_round_ph (__m512h a, int imm8, const int sae);

**SIMD Floating-Point Exceptions**
Invalid, Underflow, Precision.

**Other Exceptions**
EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions."
VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.w0 08 /r ib VRNDSCALEPS xmm1 {k1}[z], xmm2/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Rounds packed single-precision floating-point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.w0 08 /r ib VRNDSCALEPS ymm1 {k1}[z], ymm2/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Rounds packed single-precision floating-point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w0 08 /r ib VRNDSCALEPS zmm1 {k1}[z], zmm2/m512/m32bcst</td>
<td>safe, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>

**Description**

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

\[
ROUND(x) = 2^{-M} \times \text{Round\_to\_INT}(x \times 2^M, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

\[
ROUND(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]
Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Handling of special case of input values are listed in Table 5-31.

**Operation**

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction := MXCSR:RC ; get round control from MXCSR
  else
    rounding_direction := imm8[1:0] ; get round control from imm8[1:0]
  FI
  M := imm8[7:4] ; get the scaling factor

  case (rounding_direction)
  00: TMP[31:0] := round_to_nearest_even_integer(2^M*SRC[31:0])
  01: TMP[31:0] := round_to_equal_or_smaller_integer(2^M*SRC[31:0])
  10: TMP[31:0] := round_to_equal_or_larger_integer(2^M*SRC[31:0])
  11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
  ESAC;

  Dest[31:0] := 2^-M* TMP[31:0] ; scale down back to 2^-M
  if (imm8[3] = 0) Then ; check SPE
    if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
      set_precision() ; set #PE
    FI;
  FI;
  return(Dest[31:0])
}

VRNDSCALEPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
  THEN TMP_SRC := BROADCAST32(SRC, VL, k1)
ELSE TMP_SRC := SRC
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := RoundToIntegerSP(TMP_SRC[i+31:i], imm8[7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPS __m512 __mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 __mm256_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_roundscale_round_ps( __m256 a, int imm, int sae);
VRNDSCALEPS __m256 __mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_mask_roundscale_round_ps(__m256 s, __mmask8 k, __m256 a, int imm, int sae);
VRNDSCALEPS __m128 __mm128_roundscale_ps( __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 __mm128_roundscale_round_ps( __mmask8 k, __m128 a, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."
VRNDSCALES—Round Scalar Float64 Value to Include a Given Number of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W1 0B /r ib</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Rounds scalar double precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

Description

Rounds a double precision floating-point value in the low quadword (see Figure 5-29) element of the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALES is

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_\to\_\text{INT}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALES is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

\[
\text{ROUND}(x) = \text{Round}_\to\_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-31.
Operation
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0] ; get round control from imm8[1:0]
    FI
    M := imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] := round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] := round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] := round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0] := 2^{-M} * TMP[63:0] ; scale down back to 2^{-M}

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

VRNDSCALES (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN DEST[63:0] := RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0] := 0
    FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VRNDSCALES __m128d _mm_roundscale_sd (__m128d a, __m128d b, int imm);
VRNDSCALES __m128d _mm_roundscale_round_sd (__m128d a, __m128d b, int imm, int sae);
VRNDSCALES __m128d _mm_mask_roundscale_sd (__m128d s, __m128d k, __m128d a, __m128d b, int imm);
VRNDSCALES __m128d _mm_mask_roundscale_round_sd (__m128d s, __m128d k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALES __m128d _mm_mask_roundscale_roundk_sd (__mask8 k, __m128d a, __m128d b, int imm);
VRNDSCALES __m128d _mm_maskz_roundscale_roundk_sd (__mask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VRNSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP.0F3A.W0 0A /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Round the low FP16 value in xmm3/m16 to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

This instruction rounds the low FP16 value in the second source operand by the rounding mode specified in the immediate operand (see Table 5-32) and places the result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result), and returns the result as a FP16 value.

Note that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation. Three bit fields are defined and shown in Table 5-32, “Imm8 Controls for VRNDSCALEPH/VRNDSCALESH.” Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-33.

If this instruction encoding’s SPE bit (bit 3) in the immediate operand is 1, VRNDSCALESH can set MXCSR.UE without MXCSR.PE.

The formula of the operation on each data element for VRNDSCALESH is:

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_{-\text{to}\_\text{INT}}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

**Operation**

VRNDSCALESH dest[k1], src1, src2, imm8

IF k1[0] or *no writemask*:

\[
\text{DEST}.fp16[0] := \text{round}_{-\text{fp16}\_\text{to}\_\text{integer}}(\text{src2}.fp16[0], \text{imm8}) \text{ // see VRNDSCALEPH}
\]

ELSE IF *zeroing*:

\[
\text{DEST}.fp16[0] := 0
\]

//else DEST.fp16[0] remains unchanged

\[
\text{DEST}[127:16] = \text{src1}[127:16]
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESH __m128h _mm_mask_roundscale_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VRNDSCALESH __m128h _mm_maskz_roundscale_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
VRNDSCALESH __m128h _mm_roundscale_round_sh (__m128h a, __m128h b, int imm8, const int sae);
VRNDSCALESH __m128h _mm_mask_roundscale_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
VRNDSCALESH __m128h _mm_maskz_roundscale_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
VRNDSCALESH __m128h _mm_roundscale_sh (__m128h a, __m128h b, int imm8);

SIMD Floating-Point Exceptions
Invalid, Underflow, Precision.

Other Exceptions
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”
VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F3A.W0 0A / r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand). The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1’ then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}\_\text{to}\_\text{INT}(x \times 2^M, \text{round}\_\text{ctrl}),
\]

\[
\text{round}\_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round}\_\text{to}\_\text{INT}(x, \text{round}\_\text{ctrl}),
\]

\[
\text{round}\_\text{ctrl} = \text{imm}[3:0];
\]

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-31.
Operation

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction := MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction := imm8[1:0] ; get round control from imm8[1:0]
    FI
    M := imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] := round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
    ESAC;

    Dest[31:0] := 2^-M*TMP[31:0] ; scale down back to 2^-M
    if (imm8[3] = 0) Then ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

VRNDSCALESS (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN DEST[31:0] := RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[31:0] := 0
    FI;
FI;
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VRNDSCALESS __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Precision.
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-47, "Type E3 Class Exception Conditions."

VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits
VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 (k1)[z], xmm2/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 (k1)[z], ymm2/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 (k1)[z], zmm2/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double precision floating-point values in the source operand (the second operand) and stores the packed double precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $\pm\infty$ then $+\text{ZERO}$ value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VRSQRT14PD (EVEX encoded versions)

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j := 0 \text{ TO } KL-1\)

\[i := j \times 64\]

IF \(k1[j] \text{ OR *no writemask* THEN}\)

IF \((\text{EVEX.b = 1}) \text{ AND (SRC *is memory*)} \)

THEN \(\text{DEST}[i+63:i] := \text{APPROXIMATE}(1.0/ \text{SQRT}(\text{SRC}[63:0]));\)

ELSE \(\text{DEST}[i+63:i] := \text{APPROXIMATE}(1.0/ \text{SQRT}(\text{SRC}[i+63:i]));\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged*

ELSE ; zeroing-masking

\(\text{DEST}[i+63:i] := 0\)

FI;

FI;

ENDFOR;

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PD __m512d _mm512_rsqrt14_pd( __m512d a);
VRSQRT14PD __m512d _mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d _mm512_maskz_rsqrt14_pd( __mmask8 k, __m512d a);
VRSQRT14PD __m256d _mm256_rsqrt14_pd( __m256d a);
VRSQRT14PD __m256d _mm256_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm256_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm_rsqrt14_pd( __m128d a);
VRSQRT14PD __m128d _mm_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm_maskz_rsqrt14_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions."

Table 5-34. VRSQRT14PD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>(X = 2^{-2^n})</td>
<td>(2^n)</td>
<td></td>
</tr>
<tr>
<td>(X &lt; 0)</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>(X = 0)</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>(X = +0)</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>(X = +INF)</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W1 4F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square root of the scalar double precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM{r/m (r)}</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Computes the approximate reciprocal of the square roots of the scalar double precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $+\infty$ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


#### Operation

**VRSQRT14SD (EVEX version)**

IF k1[0] or *no writemask*

THEN

DEST[63:0] := APPROXIMATE(1.0 / SQRT(SRC2[63:0]))

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[63:0] := 0

FI;

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0

---

5-680 Vol. 2C VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value
Table 5-35. VRSQRT14SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +\infty$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SD __m128d _mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d __m128d __m128d __m128d k, __m128d a, __m128d b);
VRSQRT14SD __m128d __m128d __m128d __m128d m, __m128d a, __m128d b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-51, “Type E5 Class Exception Conditions.”
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 (k1)[z], xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 4E /r VRSQRT14PS ymm1 (k1)[z], ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 (k1)[z], zmm2/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an ±∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

**VRSQRT14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST[i+31:i] := APPROXIMATE(1.0/ SQRT(SRC[31:0]));

ELSE DEST[i+31:i] := APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+31:i] remains unchanged*

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0

---

### Table 5-36. VRSQRT14PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>X = 2^-2^n</td>
<td>2^n</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>X = +0</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

---

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PS __m512 _mm512_rsqrt14_ps( __m512 a);
VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);
VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps( __mmask16 k, __m512 a);
VRSQRT14PS __m256 _mm256_rsqrt14_ps( __m256 a);
VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);
VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps( __mmask8 k, __m256 a);
VRSQRT14PS __m128 _mm128_rsqrt14_ps( __m128 a);
VRSQRT14PS __m128 _mm128_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);
VRSQRT14PS __m128 _mm128_maskz_rsqrt14_ps( __mmask8 k, __m128 a);

---

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, “Type 4 Class Exception Conditions.”
VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than \(2^{-14}\). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \(\infty\) with the sign of the source value is returned. When the source operand is an \(\infty\), zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


### Operation

**VRSQRT14SS (EVEX version)**

IF k1[0] or *no writemask*

THEN DEST[31:0] := APPROXIMATE(1.0/ SQRT(SRC2[31:0]))

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged* 
ELSE ; zeroing-masking

THEN DEST[31:0] := 0

FI;

Fi;


DEST[MAXVL-1:128] := 0
Table 5-37. VRSQRT14SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +\text{INF}$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14SS __m128 _mm_rsqrt14_ss(__m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-51, “Type E5 Class Exception Conditions.”
**VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values**

**Opcode/Instruction** | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVEX.128.66.MAP6.W0 4E /r VRSQRTPH xmm1[k1]{z}, xmm2/m128/m16bcst | A | V/V | AVX512-FP16 AVX512VL | Compute the approximate reciprocals of the square roots of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1.

EVEX.256.66.MAP6.W0 4E /r VRSQRTPH ymm1[k1]{z}, ymm2/m256/m16bcst | A | V/V | AVX512-FP16 AVX512VL | Compute the approximate reciprocals of the square roots of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1.

EVEX.512.66.MAP6.W0 4E /r VRSQRTPH zmm1[k1]{z}, zmm2/m512/m16bcst | A | V/V | AVX512-FP16 | Compute the approximate reciprocals of the square roots of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocals square-root of 8/16/32 packed FP16 floating-point values in the source operand (the second operand) and stores the packed FP16 floating-point results in the destination operand.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$. For special cases, see Table 5-38. The destination elements are updated according to the writemask.

### Table 5-38. VRSQRTPH/VRSQRTSH Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Reset Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including $-\infty$</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>$-\infty$</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>$+\infty$</td>
<td></td>
</tr>
<tr>
<td>$X = +\infty$</td>
<td>$+0$</td>
<td></td>
</tr>
</tbody>
</table>
Operation

VRSQRTPH dest[k1], src

VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.fp16[0]
        ELSE:
            tsrc := src.fp16[i]
    ELSE:
        DEST.fp16[i] := APPROXIMATE(1.0 / SQRT(tsrc))
    ELSE IF *zeroing*:
        DEST.fp16[i] := 0
    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRTPH __m128h _mm_mask_rsqrt_ph (__m128h src, __mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_maskz_rsqrt_ph (__mmask8 k, __m128h a);
VRSQRTPH __m128h _mm_rsqrt_ph (__m128h a);
VRSQRTPH __m256h _mm256_mask_rsqrt_ph (__m256h src, __mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_maskz_rsqrt_ph (__mmask16 k, __m256h a);
VRSQRTPH __m256h _mm256_rsqrt_ph (__m256h a);
VRSQRTPH __m512h _mm512_mask_rsqrt_ph (__m512h src, __mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_maskz_rsqrt_ph (__mmask32 k, __m512h a);
VRSQRTPH __m512h _mm512_rsqrt_ph (__m512h a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.MAP6.W0 4F /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compute the approximate reciprocal square root of the FP16 value in xmm3/m16 and store the result in the low word element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

Description

This instruction performs the computation of the approximate reciprocal square-root of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1.

The maximum relative error for this approximation is less than $2^{-11} + 2^{-14}$.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL−1:128 of the destination operand are zeroed.

For special cases, see Table 5-38.

Operation

VRSQRTSH dest{k1}, src1, src2

VL = 128, 256 or 512
KL := VL/16

IF k1[0] or *no writemask*:
  DEST.fp16[0] := APPROXIMATE(1.0 / SQRT(src2.fp16[0]))
ELSE IF *zeroing*:
  DEST.fp16[0] := 0
  //else DEST.fp16[0] remains unchanged

DEST[MAXVL−1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRTSH __m128h _mm_mask_rsqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_maskz_rsqrt_sh (__mmask8 k, __m128h a, __m128h b);
VRSQRTSH __m128h _mm_rsqrt_sh (__m128h a, __m128h b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Table 2-58, “Type E10 Class Exception Conditions.”
VSACPDPD—Scale Packed Float64 Values With Float64 Values

**Op/ En Instruction**  
A Help

64/32 bit Mode Support CPUID Feature

Description

Performs a floating-point scale of the packed double precision floating-point values in the first source operand by multiplying them by 2 to the power of the double precision floating-point values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*2floor(zmm3).

Floor(zmm3) means maximum integer value \( \leq \) zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-39 and Table 5-40.

### Table 5-39. VSCALEFPD/SD/PS/SS Special Cases

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>O/Denorm/Norm</th>
<th>Set IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>±NaN</td>
<td>+Inf</td>
<td>+0 QNaN(Src1)</td>
<td>IF either source is SNAN</td>
</tr>
<tr>
<td>±NaN</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>YES</td>
</tr>
<tr>
<td>±Inf</td>
<td>QNaN(Src2)</td>
<td>Src1 QNaN_Indefinite</td>
<td>IF Src2 is SNAN or -INF</td>
</tr>
<tr>
<td>±0</td>
<td>QNaN(Src2)</td>
<td>Src1</td>
<td>IF Src2 is SNAN or +INF</td>
</tr>
<tr>
<td>Denorm/Norm</td>
<td>QNaN(Src2)</td>
<td>±INF (Src1 sign)</td>
<td>Compute Result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>±0 (Src1 sign)</td>
<td>IF Src2 is SNAN</td>
</tr>
</tbody>
</table>
Table 5-40. Additional VSCALEFPD/SD Special Cases

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>result</td>
<td>&lt; 2^{-1074})</td>
</tr>
<tr>
<td>(\text{\emph{</td>
<td>result}} \geq 2^{1024})</td>
<td>±INF (Src1 sign) or ±Max-normal (Src1 sign)</td>
</tr>
</tbody>
</table>

Operation

SCALE(SRC1, SRC2)
{
   TMP_SRC2 := SRC2
   TMP_SRC1 := SRC1
   IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
   IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
   /* SRC2 is a 64 bits floating-point value */
   DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}

VSCALEFPD (EVEX encoded versions)

\((KL, VL) = (2, 128), (4, 256), (8, 512)\)
\(\text{IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)}
\)
\(\text{THEN}
\)
\(\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);}\)
\(\text{ELSE}
\)
\(\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);}\)
\(\text{FI;}\)
\(\text{FOR j := 0 TO KL-1}
\)
\(\text{i := j * 64}
\)
\(\text{IF k1[j] OR *no writemask* THEN}
\)
\(\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)}
\)
\(\text{THEN DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[63:0]);}
\)
\(\text{ELSE DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[i+63:i]);}
\)
\(\text{FI;}
\)
\(\text{ELSE}
\)
\(\text{IF *merging-masking* ; merging-masking}
\)
\(\text{THEN *DEST[i+63:i] remains unchanged*}
\)
\(\text{ELSE ; zeroing-masking}
\)
\(\text{DEST[i+63:i] := 0}
\)
\(\text{FI}
\)
\(\text{ENDIF;}
\)
\(\text{DEST[MAXVL-1:VL] := 0}
\)
**Intel C/C++ Compiler Intrinsic Equivalent**

VSACLEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSACLEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSACLEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSACLEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSACLEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSACLEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

**Other Exceptions**

See Table 2-46, "Type E2 Class Exception Conditions.”
VSSCALEFPH—Scale Packed FP16 Values with FP16 Values

**Op Code/ Instruction**

| EVEX.128.66.MAP6.W0 2C /r | VSSCALEFPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst |
| EVEX.256.66.MAP6.W0 2C /r | VSSCALEFPH ymm1{k1}{z}, ymm2, ymm3/m256/m16bcst |
| EVEX.512.66.MAP6.W0 2C /r | VSSCALEFPH zmm1{k1}{z}, zmm2, zmm3/m512/m16bcst (er) |

**Op/En**

A

**64/32 bit Mode Support**

V/V

**CPUID Feature Flag**

AVX512-FP16

**Description**

This instruction performs a floating-point scale of the packed FP16 values in the first source operand by multiplying it by 2 to the power of the FP16 values in second source operand. The destination elements are updated according to the writemask.

The equation of this operation is given by:

\[ zmm1 := zmm2 \times 2^{\text{floor}(zmm3)} \]

Floor(zmm3) means maximum integer value \( \leq \) zmm3.

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits), and on the SAE bit.

Handling of special-case input values are listed in Table 5-41 and Table 5-42.

**Table 5-41. VSSCALEFPH/VSSCALEFSH Special Cases**

<table>
<thead>
<tr>
<th>Src1</th>
<th>+/-INF</th>
<th>-INF</th>
<th>0/Denorm/Norm</th>
<th>Set IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-NaN</td>
<td>QNaN(Src1)</td>
<td>+INF</td>
<td>+0</td>
<td>QNaN(Src1)</td>
</tr>
<tr>
<td>+/-SNaN</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
</tr>
<tr>
<td>+/-INF</td>
<td>QNaN(Src2)</td>
<td>Src1</td>
<td>QNaN_indefinite</td>
<td>Src1</td>
</tr>
<tr>
<td>+0</td>
<td>QNaN(Src2)</td>
<td>QNaN_indefinite</td>
<td>Src1</td>
<td>Src1</td>
</tr>
<tr>
<td>Denorm/Norm</td>
<td>QNaN(Src2)</td>
<td>+/-INF (Src1 sign)</td>
<td>+0 (Src1 sign)</td>
<td>Compute Result</td>
</tr>
</tbody>
</table>

**Table 5-42. Additional VSSCALEFPH/VSSCALEFSH Special Cases**

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned Value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(</td>
<td>result</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>result</td>
</tr>
</tbody>
</table>
**Operation**

```python
def scale_fp16(src1, src2):
    tmp1 := src1
    tmp2 := src2
    return tmp1 * POWER(2, FLOOR(tmp2))
```

**VSACLEFPH dest[k1], src1, src2**

VL = 128, 256, or 512

KL := VL / 16

IF (VL = 512) AND (EVEX.b = 1) and no memory operand:
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and (EVEX.b = 1):
            tsrc := src2.fp16[0]
        ELSE:
            tsrc := src2.fp16[i]
        dest.fp16[i] := scale_fp16(src1.fp16[i], tsrc)
    ELSE IF *zeroing*:
        dest.fp16[i] := 0
        //else dest.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSACLEFPH __m128h _mm_mask_scalef_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSACLEFPH __m128h _mm_maskz_scalef_ph (__mmask8 k, __m128h a, __m128h b);
VSACLEFPH __m128h _mm_maskz_scalef_round_ph (__mmask32 k, __m128h a, __m128h b, const int rounding);
VSACLEFPH __m128h _mm512_mask_scalef_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, const int);  

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions".

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.
**VSCALEFPS—Scale Packed Float32 Values With Float32 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}[z], xmm2, xmm3/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}[z], ymm2, ymm3/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Scale the packed single-precision values in ymm2 using floating-point values from ymm3/m256/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}[z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying them by 2 to the power of the float32 values in second source operand.

The equation of this operation is given by:

\[ \text{zmm1} := \text{zmm2} \times 2^{\text{floor(zmm3)}}. \]

Floor(zmm3) means maximum integer value \( \leq \text{zmm3} \).

If the result cannot be represented in single-precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is an XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-39 and Table 5-43.

**Table 5-43. Additional VSCALEFPS/SS Special Cases**

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>≤ 2⁻¹⁴⁹</td>
<td>±0 or ±Min-Denormal (Src1 sign)</td>
</tr>
<tr>
<td></td>
<td>≥ 2⁻¹²⁸</td>
<td>±INF (Src1 sign) or ±Max-normal (Src1 sign)</td>
</tr>
</tbody>
</table>
Operation

SCALE(SRC1, SRC2)
{ ; Check for denormal operands
  TMP_SRC2 := SRC2
  TMP_SRC1 := SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 32 bits floating-point value */
  DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

VSSCALEFPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[31:0]);
    ELSE DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[i+31:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

VSSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b);
VSSCALEFPS __m512 _mm512_scalef_ps(__m512 a, __m512 b);
VSSCALEFPS __m512 _mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSSCALEFPS __m512 _mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSSCALEFPS __m256 _mm256_scalef_ps(__m256 a, __m256 b);
VSSCALEFPS __m256 _mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSSCALEFPS __m256 _mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSSCALEFPS __m128 _mm_scalef_ps(__m128 a, __m128 b);
VSSCALEFPS __m128 _mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSSCALEFPS __m128 _mm_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.
**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”
VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

**Description**

Performs a floating-point scale of the scalar double precision floating-point value in the first source operand by multiplying it by 2 to the power of the double precision floating-point value in second source operand.

The equation of this operation is given by:

\[
xmm1 := xmm2 \times 2^{\text{floor}(xmm3)}.
\]

\text{Floor}(xmm3) means maximum integer value \( \leq xmm3 \).

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

**EVEX encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-39 and Table 5-40.

**Operation**

\[
\text{SCALE}(\text{SRC1}, \text{SRC2})
\]

```c
{  ; Check for denormal operands
    TMP_SRC2 := SRC2
    TMP_SRC1 := SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] := TMP_SRC1[63:0] * \text{POW}[2, \text{Floor}(TMP_SRC2[63:0])]
} 
```
VSDF (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*

THEN

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

IF k1[0] OR *no writemask*

THEN DEST[63:0] := SCALE(SRC1[63:0], SRC2[63:0])
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*
ELSE ; zeroing-masking

DEST[63:0] := 0
FI
FI

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VSDF __m128d __mm_scalef_round_sd(__m128d a, __m128d b, int);
VSDF __m128d __mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSDF __m128d __mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
VSCALEFSH—Scale Scalar FP16 Values with FP16 Values

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a floating-point scale of the low FP16 element in the first source operand by multiplying it by 2 to the power of the low FP16 element in second source operand, storing the result in the low element of the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The equation of this operation is given by:

\[ \text{xmm1} := \text{ xmm2} \times 2^{\text{floor}(\text{ xmm3})}. \]

Floor(\text{ xmm3}) means maximum integer value \( \leq \text{ xmm3}. \)

If the result cannot be represented in FP16, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand), is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

Handling of special-case input values are listed in Table 5-41 and Table 5-42.

Operation

VSCALEFSH dest[k1], src1, src2

IF (EVEX.b = 1) and no memory operand:

SET_RM(EVEX.RC)
ELSE

SET_RM(MXCSR.RC)

IF k1[0] or *no writemask*:

dest.fp16[0] := scale_fp16(src1.fp16[0], src2.fp16[0]) \ // see VSCALEFPH
ELSE IF *zeroing*:

dest.fp16[0] := 0
//else DEST.fp16[0] remains unchanged

DEST[MAXVL-1:128] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VSCALEFSH __m128h _mm_mask_scalef_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_maskz_scalef_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_scalef_round_sh (__m128h a, __m128h b, const int rounding);
VSCALEFSH __m128h _mm_mask_scalef_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_maskz_scalef_sh (__mmask8 k, __m128h a, __m128h b);
VSCALEFSH __m128h _mm_scalef_sh (__m128h a, __m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.
**VSCALEFSS—Scale Scalar Float32 Value With Float32 Value**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}[z], xmm2, xmm3/m32{er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 to the power of the float32 value in second source operand.

The equation of this operation is given by:

\[ \text{xmm1} := \text{xmm2} \times 2^{\text{floor}(\text{xmm3})} \]

Floor(xmm3) means maximum integer value \( \leq \text{xmm3} \).

If the result cannot be represented in single-precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-39 and Table 5-43.
Operation
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 := SRC2
    TMP_SRC1 := SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2 = 0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1 = 0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

VSACLEFSS (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] := SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] := 0
        FI
    FI;
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VSACLEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSACLEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSACLEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions.”
### VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/E n</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPSD vm32x {k1}, xmm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPSD vm32y {k1}, ymm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPSD vm32z {k1}, zmm1</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter double precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, ymm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, zmm1</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, scatter double precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory...
ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSZ of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be scattered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be scattered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

VSCATTERDPS (EVEX encoded versions)

(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASEx_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] :=
            SRC[i+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

VSCATTERDPD (EVEX encoded versions)

(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASEx_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] :=
            SRC[i+63:i]
            k1[j] := 0
    FI;
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword

ENDFOR
k1[MAX_KL-1:KL] := 0

VSCATTERQPS (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    k := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] :=
            SRC[i+31:i]
        k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

VSCATTERQPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] :=
            SRC[i+63:i]
        k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VSCATTERDPD void _mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m256d a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m256 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m128 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m128 a, int scale);
VSCATTERDPD void _mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void _mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERDPS void _mm256_i32scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERDPS void _mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
**SIMD Floating-Point Exceptions**
Invalid, Overflow, Underflow, Precision, Denormal.

**Other Exceptions**
See Table 2-61, “Type E12 Class Exception Conditions.”
VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, ymm2, zmm3/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, ymm2, zmm3/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, ymm2, zmm3/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, ymm2, zmm3/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

256-bit Version: Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.
The writemask updates the destination operand with the granularity of 32/64-bit data elements.

**Operation**

Select2(SRC, control) {
CASE (control[0]) OF
  0: TMP := SRC[127:0];
  1: TMP := SRC[255:128];
ESAC;
RETURN TMP
}

Select4(SRC, control) {
CASE (control[1:0]) OF
  0: TMP := SRC[127:0];
  1: TMP := SRC[255:128];
  2: TMP := SRC[383:256];
  3: TMP := SRC[511:384];
ESAC;
RETURN TMP
}

**VSHUFF32x4 (EVEX versions)**

(KL, VL) = (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;

IF VL = 512
  TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+31:i] := 0
      FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VSHUFF64x2 (EVEX 512-bit version)

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TEMP_SRC2[i+63:i] := SRC2[63:0]
        ELSE TEMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;

IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                THEN DEST[i+63:i] := 0
            FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VSHUFF32x4 (EVEX 512-bit version)

(KL, VL) = (8, 256), (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TEMP_SRC2[i+31:i] := SRC2[31:0]
        ELSE TEMP_SRC2[i+31:i] := SRC2[i+31:i]
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;

IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;

FOR j := 0 TO KL-1
    i := j * 32
IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+31:i] := 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VSHUFI64x2 (EVEX 512-bit version)
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] := SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
    FI;
IF VL = 512
    TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            THEN DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VSHUF32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUF32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUF32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUF32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUF32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUF32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);

VSHUFF32x4 __m256 _mm256_shuffle_f32x4(__m256 a, __m256 b, int imm);
VSHUFF32x4 __m256 _mm256_mask_shuffle_f32x4(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFF32x4 __m256 _mm256_maskz_shuffle_f32x4(__mmask8 k, __m256 a, __m256 b, int imm);

VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD If EVEX.L'L' = 0 for VSHUFF32x4/VSHUFF64x2.
VSQRTPH—Compute Square Root of Packed FP16 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAP5.W0 51 /r VSQRTPH xmm1{k1}{z}, xmm2/m128/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Compute square roots of the packed FP16 values in xmm2/m128/m16bcst, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAP5.W0 51 /r VSQRTPH ymm1{k1}{z}, ymm2/m256/m16bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Compute square roots of the packed FP16 values in ymm2/m256/m16bcst, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAP5.W0 51 /r VSQRTPH zmm1{k1}{z}, zmm2/m512/m16bcst {er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compute square roots of the packed FP16 values in zmm2/m512/m16bcst, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMs/r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction performs a packed FP16 square-root computation on the values from source operand and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

Operation

VSQRTPH dest[k1], src

VL = 128, 256 or 512
KL := VL/16

FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF SRC is memory and (EVEX.b = 1):
      tsrc := src.fp16[0]
    ELSE:
      tsrc := src.fp16[i]
    DEST.fp16[i] := SQRT(tsrc)
  ELSE IF *zeroing*:
    DEST.fp16[i] := 0
  //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTPH __m128h __m128h_mm_mask sqrt_ph (__m128h src, __mmask8 k, __m128h a);
VSQRTPH __m128h __m128h_mm_maskz_sqrt_ph (__mmask8 k, __m128h a);
VSQRTPH __m128h __m128h_mm_sqrt_ph (__m128h a);
VSQRTPH __m256h __m256h_mm256_mask_sqrt_ph (__m256h src, __mmask16 k, __m256h a);
VSQRTPH __m256h __m256h_mm256_maskz_sqrt_ph (__mmask16 k, __m256h a);
VSQRTPH __m256h __m256h_mm256_sqrt_ph (__m256h a);
VSQRTPH __m512h __m512h_mm512_mask_sqrt_ph (__m512h src, __mmask32 k, __m512h a);
VSQRTPH __m512h __m512h_mm512_maskz_sqrt_ph (__mmask32 k, __m512h a);
VSQRTPH __m512h __m512h_mm512_sqrt_ph (__m512h a);
VSQRTPH __m512h __m512h_mm512_mask_sqrt_round_ph (__m512h src, __mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h __m512h_mm512_maskz_sqrt_round_ph (__mmask32 k, __m512h a, const int rounding);
VSQRTPH __m512h __m512h_mm512_sqrt_round_ph (__m512h a, const int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
**VSQRTH—Compute Square Root of Scalar FP16 Value**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a scalar FP16 square-root computation on the source operand and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

**Operation**

\[ \text{VSQRTH dest}(k1), \text{src1}, \text{src2} \]

- IF \( k1[0] \) or *no writemask*:
  - \( \text{DEST.fp16}[0] := \text{SQRT(src2.fp16[0])} \)
- ELSE IF *zeroing*:
  - \( \text{DEST.fp16}[0] := 0 \)
- //else \( \text{DEST.fp16}[0] \) remains unchanged

\[ \text{DEST}[127:16] := \text{src1}[127:16] \]
\[ \text{DEST}[\text{MAXVL}-1:128] := 0 \]

**Intel C/C++ Compiler Intrinsic Equivalent**

- `VSQRTH _m128h _mm_mask_sqrt_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);`
- `VSQRTH _m128h _mm_maskz_sqrt_round_sh (__mmask8 k, __m128h a, __m128h b, const int rounding);`
- `VSQRTH _m128h _mm_sqrt_round_sh (__m128h a, __m128h b, const int rounding);`
- `VSQRTH _m128h _mm_mask_sqrt_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);`
- `VSQRTH _m128h _mm_maskz_sqrt_sh (__mmask8 k, __m128h a, __m128h b);`
- `VSQRTH _m128h _mm_sqrt_sh (__m128h a, __m128h b);`

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
**VSUBPH—Subtract Packed FP16 Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.NP.MAPS.WO 5C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Subtract packed FP16 values from xmm3/m128/m16bcst to xmm2, and store the result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.NP.MAPS.WO 5C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16 AVX512VL</td>
<td>Subtract packed FP16 values from ymm3/m256/m16bcst to ymm2, and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.NP.MAPS.WO 5C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Subtract packed FP16 values from zmm3/m512/m16bcst to zmm2, and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction subtracts packed FP16 values from the second source operand from the corresponding elements in the first source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

**Operation**

**VSUBPH (EVEX encoded versions) when src2 operand is a register**

VL = 128, 256 or 512
KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):
    SET_RM(EVEX.RC)
ELSE
    SET_RM(MXCSR.RC)

FOR j := 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
    ELSE IF *zeroing*:
        DEST.fp16[j] := 0
    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0
VSUBPH (EVEX encoded versions) when src2 operand is a memory source
VL = 128, 256 or 512
KL := VL/16

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[0]
    ELSE:
      DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
  // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VSUBPH __m128h _mm_mask_sub_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_maskz_sub_ph (__mmask8 k, __m128h a, __m128h b);
VSUBPH __m128h _mm_sub_ph (__m128h a, __m128h b);
VSUBPH __m256h _mm256_mask_sub_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_maskz_sub_ph (__mmask16 k, __m256h a, __m256h b);
VSUBPH __m256h _mm256_sub_ph (__m256h a, __m256h b);
VSUBPH __m512h _mm512_mask_sub_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_maskz_sub_ph (__mmask32 k, __m512h a, __m512h b);
VSUBPH __m512h _mm512_sub_ph (__m512h a, __m512h b);
VSUBPH __m512h _mm512_mask_sub_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_maskz_sub_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);
VSUBPH __m512h _mm512_sub_round_ph (__m512h a, __m512h b, int rounding);

SIMD Floating-Point Exceptions
Invalid, Underflow, Overflow, Precision, Denormal.

Other Exceptions
EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”
VSUBSH—Subtract Scalar FP16 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F3.MAP5.W0 5C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Subtract the low FP16 value in xmm3/m16 from xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction subtracts the low FP16 value from the second source operand from the corresponding value in the first source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

**Operation**

**VSUBSH (EVEX encoded versions)**

IF EVEX.b = 1 and SRC2 is a register:

SET_RM(EVEX.RC)
ELSE

SET_RM(MXCSR.RC)

IF k1[0] OR *no writemask*:

DEST.fp16[0] := SRC1.fp16[0] - SRC2.fp16[0]
ELSE IF *zeroing*:

DEST.fp16[0] := 0
// else dest.fp16[0] remains unchanged

DEST[MAXVL-1:128] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSUBSH _m128h _mm_mask_sub_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);  
VSUBSH _m128h _mm_maskz_sub_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);  
VSUBSH _m128h _mm_sub_round_sh (__m128h a, __m128h b, int rounding);  
VSUBSH _m128h _mm_mask_sub_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);  
VSUBSH _m128h _mm_maskz_sub_sh (__mmask8 k, __m128h a, __m128h b);  
VSUBSH _m128h _mm_sub_sh (__m128h a, __m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."
VTTESTPD/VTTESTPS—Packed Bit Test

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| VEX.128.66.0F38.W0 0E /r
VTTESTPS xmm1, xmm2/m128 | RM V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources. |
| VEX.256.66.0F38.W0 0E /r
VTTESTPS ymm1, ymm2/m256 | RM V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources. |
| VEX.128.66.0F38.W0 0F /r
VTTESTPD xmm1, xmm2/m128 | RM V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed double precision floating-point sources. |
| VEX.256.66.0F38.W0 0F /r
VTTESTPD ymm1, ymm2/m256 | RM V/V | AVX | Set ZF and CF depending on sign bit AND and ANDN of packed double precision floating-point sources. |

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (r)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

VTTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTTESTPS with VEX.W=1 will cause #UD.

VTTESTPD performs a bitwise comparison of all the sign bits of the double precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M reg field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
**Operation**

**VTESTPS (128-bit version)**

\[ \text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND DEST}[127:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0) \\
& \begin{array}{l}
\text{THEN ZF} := 1; \\
\text{ELSE ZF} := 0;
\end{array}
\end{align*}
\]

\[ \text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND NOT DEST}[127:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0) \\
& \begin{array}{l}
\text{THEN CF} := 1; \\
\text{ELSE CF} := 0;
\end{array}
\end{align*}
\]

DEST (unmodified)

\[ \text{AF} := \text{OF} := \text{PF} := \text{SF} := 0; \]

**VTESTPS (VEX.256 encoded version)**

\[ \text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND DEST}[255:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = \text{TEMP}[160] = \text{TEMP}[191] = \text{TEMP}[224] = \text{TEMP}[255] = 0) \\
& \begin{array}{l}
\text{THEN ZF} := 1; \\
\text{ELSE ZF} := 0;
\end{array}
\end{align*}
\]

\[ \text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND NOT DEST}[255:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = \text{TEMP}[160] = \text{TEMP}[191] = \text{TEMP}[224] = \text{TEMP}[255] = 0) \\
& \begin{array}{l}
\text{THEN CF} := 1; \\
\text{ELSE CF} := 0;
\end{array}
\end{align*}
\]

DEST (unmodified)

\[ \text{AF} := \text{OF} := \text{PF} := \text{SF} := 0; \]

**VTESTPD (128-bit version)**

\[ \text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND DEST}[127:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[63] = \text{TEMP}[127] = 0) \\
& \begin{array}{l}
\text{THEN ZF} := 1; \\
\text{ELSE ZF} := 0;
\end{array}
\end{align*}
\]

\[ \text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND NOT DEST}[127:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[63] = \text{TEMP}[127] = 0) \\
& \begin{array}{l}
\text{THEN CF} := 1; \\
\text{ELSE CF} := 0;
\end{array}
\end{align*}
\]

DEST (unmodified)

\[ \text{AF} := \text{OF} := \text{PF} := \text{SF} := 0; \]

**VTESTPD (VEX.256 encoded version)**

\[ \text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND DEST}[255:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[63] = \text{TEMP}[127] = \text{TEMP}[191] = \text{TEMP}[255] = 0) \\
& \begin{array}{l}
\text{THEN ZF} := 1; \\
\text{ELSE ZF} := 0;
\end{array}
\end{align*}
\]

\[ \text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND NOT DEST}[255:0] \]

\[
\begin{align*}
\text{IF} & (\text{TEMP}[63] = \text{TEMP}[127] = \text{TEMP}[191] = \text{TEMP}[255] = 0) \\
& \begin{array}{l}
\text{THEN CF} := 1; \\
\text{ELSE CF} := 0;
\end{array}
\end{align*}
\]

DEST (unmodified)

\[ \text{AF} := \text{OF} := \text{PF} := \text{SF} := 0; \]
**Intel C/C++ Compiler Intrinsic Equivalent**

**VTESTPS**

- `int _mm256_testz_ps (__m256 s1, __m256 s2);`
- `int _mm256_testc_ps (__m256 s1, __m256 s2);`
- `int _mm256_testnzc_ps (__m256 s1, __m128 s2);`
- `int _mm_testz_ps (__m128 s1, __m128 s2);`
- `int _mm_testc_ps (__m128 s1, __m128 s2);`
- `int _mm_testnzc_ps (__m128 s1, __m128 s2);`

**VTESTPD**

- `int _mm256_testz_pd (__m256d s1, __m256d s2);`
- `int _mm256_testc_pd (__m256d s1, __m256d s2);`
- `int _mm256_testnzc_pd (__m256d s1, __m256d s2);`
- `int _mm_testz_pd (__m128d s1, __m128d s2);`
- `int _mm_testc_pd (__m128d s1, __m128d s2);`
- `int _mm_testnzc_pd (__m128d s1, __m128d s2);`

**Flags Affected**
The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

**SIMD Floating-Point Exceptions**
None.

**Other Exceptions**
See Table 2-21, “Type 4 Class Exception Conditions.”
Additionally:

- **#UD**
  - If VEX.vvvv ≠ 1111B.
  - If VEX.W = 1 for VTESTPS or VTESTPD.
VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.NP.MAP5.W0 2E /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512-FP16</td>
<td>Compare low FP16 values in xmm1 and xmm2/m16 and set the EFLAGS flags accordingly.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Scalar</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VUCOMISH instruction differs from the VCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

**Operation**

VUCOMISH

RESULT := UnorderedCompare(SRC1.fp16[0],SRC2.fp16[0])
if RESULT is UNORDERED:
   ZF, PF, CF := 1, 1, 1
else if RESULT is GREATER_THAN:
   ZF, PF, CF := 0, 0, 0
else if RESULT is LESS_THAN:
   ZF, PF, CF := 0, 0, 1
else: // RESULT is EQUALS
   ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VUCOMISH int _mm_ucomieq_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomige_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomigt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomile_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomilt_sh (__m128h a, __m128h b);
VUCOMISH int _mm_ucomineq_sh (__m128h a, __m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions."
VZEROALL—Zero XMM, YMM, and ZMM Registers

Opcode/Instruction | Op/En | 64/32 bit Mode Support | CPUID Feature Flag | Description
---|---|---|---|---
VEX.256.0F.WIG 77 VZEROALL | ZO | V/V | AVX | Zero some of the XMM, YMM, and ZMM registers.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

In 64-bit mode, the instruction zeroes XMM0-XMM15, YMM0-YMM15, and ZMM0-ZMM15. Outside 64-bit mode, it zeroes only XMM0-XMM7, YMM0-YMM7, and ZMM0-ZMM7. VZEROALL does not modify ZMM16-ZMM31.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

`simd_reg_file[][]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm, and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0, and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512.

**VZEROALL (VEX.256 encoded version)**

IF (64-bit mode)

```
limit := 15
```

ELSE

```
limit := 7
```

FOR i in 0 .. limit:

```
simd_reg_file[i][MAXVL-1:0] := 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VZEROALL: `_mm256_zeroall()`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-25, “Type 8 Class Exception Conditions.”
**Description**

In 64-bit mode, the instruction zeroes the bits in positions 128 and higher in YMM0-YMM15 and ZMM0-ZMM15. Outside 64-bit mode, it zeroes those bits only in YMM0-YMM7 and ZMM0-ZMM7. VZEROUPPER does not modify the lower 128 bits of these registers and it does not modify ZMM16-ZMM31.

This instruction is recommended when transitioning between AVX and legacy SSE code; it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

`simd_reg_file[][]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm, and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0, and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits.

```
VZEROUPPER
IF (64-bit mode)
  limit := 15
ELSE
  limit := 7
FOR i in 0 .. limit:
  simd_reg_file[i][MAXVL-1:128] := 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VZEROUPPER: _mm256_zeroupper()
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-25, "Type 8 Class Exception Conditions."
6.1 INSTRUCTIONS (W-Z)

WAIT/FWAIT—Wait

**Opcode Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**
Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction’s results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**
CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**
The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**
None.

**Protected Mode Exceptions**
#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**
Same exceptions as in protected mode.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**
Same exceptions as in protected mode.
WBINVD—Write Back and Invalidate Cache

**Opcode** | **Instruction** | **Op/En** | **64-Bit Mode** | **Compat/ Leg Mode** | **Description**
---|---|---|---|---|---
0F 09 | WBINVD | ZO | Valid | Valid | Write back and flush internal caches; initiate writing-back and flushing of external caches.

**Description**

Writes back all modified cache lines in the processor’s internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see “Serializing Instructions” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction. This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**IA-32 Architecture Compatibility**

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

**Operation**

WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)

**Intel C/C++ Compiler Intrinsic Equivalent**

WBINVD void _wbinvd(void);

**Flags Affected**

None.
Protected Mode Exceptions
#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
WBNOINVD—Write Back and Do Not Invalidate Cache

**Description**
The WBNOINVD instruction writes back all modified cache lines in the processor’s internal cache to main memory but does not invalidate (flush) the internal caches.

After executing this instruction, the processor does not wait for the external caches to complete their write-back operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back signal. The amount of time or cycles for WBNOINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBNOINVD instruction can have an impact on logical processor interrupt/event response time.

The WBNOINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see “Serializing Instructions” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**
WriteBack(InternalCaches);
Continue; (* Continue execution *)

**Intel C/C++ Compiler Intrinsic Equivalent**
WBNOINVD void _wbnoinvd(void);

**Flags Affected**
None.

**Protected Mode Exceptions**
#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**
#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**
#GP(0) WBNOINVD cannot be executed at the virtual-8086 mode.

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.
64-Bit Mode Exceptions

Same exceptions as in protected mode.
WRFSBASE/WRGSBASE—Write FS/GS Segment Base

### Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field. The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared. This instruction is supported only in 64-bit mode.

### Operation

FS/GS segment base address := SRC;

### Flags Affected

None.

### C/C++ Compiler Intrinsic Equivalent

```c
WRFSBASE void _writefsbase_u32( unsigned int );
WRFSBASE _writefsbase_u64( unsigned __int64 );
WRGSBASE void _writegsbase_u32( unsigned int );
WRGSBASE _writegsbase_u64( unsigned __int64 );
```

### Protected Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

### Real-Address Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.
64-Bit Mode Exceptions

#UD  If the LOCK prefix is used.
    If CR4.FSGSBASE[bit 16] = 0.
    If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0

#GP(0)  If the source register contains a non-canonical address.
**WRMSR—Write to Model Specific Register**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 30</td>
<td>WRMSR</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Write the value in EDX:EAX to MSR specified by ECX.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, “Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

**IA-32 Architecture Compatibility**

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

**Operation**

\[
\text{MSR[ECX]} := \text{EDX:EAX};
\]

**Flags Affected**

None.
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the value in ECX specifies a reserved or unimplemented MSR address.
If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
If the source register contains a non-canonical address and ECX specifies one of the following MSR: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.

#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If the value in ECX specifies a reserved or unimplemented MSR address.
If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
If the source register contains a non-canonical address and ECX specifies one of the following MSR: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
### WRPKRU—Write Data to User Page Key Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 EF WRPKRU</td>
<td>ZO</td>
<td>V/V</td>
<td>OSPKE</td>
<td>Writes EAX into PKRU.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX, and RAX are ignored.

WRPKRU will never execute speculatively. Memory accesses affected by PKRU register will not execute (even speculatively) until all prior executions of WRPKRU have completed execution and updated the PKRU register.

#### Operation

IF (ECX = 0 AND EDX = 0)
   THEN PKRU := EAX;
   ELSE #GP(0);
FI;

#### Flags Affected

None.

#### C/C++ Compiler Intrinsic Equivalent

WRPKRU void _wrpkru(uint32_t);

#### Protected Mode Exceptions

- **#GP(0)** If ECX ≠ 0.
- **#UD** If EDX ≠ 0.

#### Real-Address Mode Exceptions

Same exceptions as in protected mode.

#### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

#### Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
WRSSD/WRSSQ—Write to Shadow Stack

**Description**
Writes bytes in register source to the shadow stack.

**Operation**
IF CPL = 3
  IF (CR4.CET & IA32_U.CET.SH_STK_EN) = 0
    THEN #UD; Fi;
  IF (IA32_U.CET.WR_SHSTK_EN) = 0
    THEN #UD; Fi;
ELSE
  IF (CR4.CET & IA32_S.CET.SH_STK_EN) = 0
    THEN #UD; Fi;
  IF (IA32_S.CET.WR_SHSTK_EN) = 0
    THEN #UD; Fi;
FI;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
  THEN
    (* Destination not 8B aligned *)
    IF DEST_LA[2:0]
      THEN GP(0); Fi;
    Shadow_stack_store 8 bytes of SRC to DEST_LA;
ELSE
    (* Destination not 4B aligned *)
    IF DEST_LA[1:0]
      THEN GP(0); Fi;
    Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA;
FI;

**Flags Affected**
None.

**C/C++ Compiler Intrinsic Equivalent**
WRSSD void _wrssd(__int32, void *);
WRSSQ void _wrssq(__int64, void *);
Protected Mode Exceptions

#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
   If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
   If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If destination is located in a non-writeable segment.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
   If linear address of destination is not 4 byte aligned.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD The WRSS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The WRSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
   If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
   If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.

#PF(fault-code) If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
   If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
   If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.

#GP(0) If a memory address is in a non-canonical form.
   If linear address of destination is not 4 byte aligned.

#PF(fault-code) If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
Other terminal and non-terminal faults.
WRUSSD/WRUSSQ—Write to User Shadow Stack

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 OF 38 F5 l(11)rrr:bbb WRUSSD m32, r32</td>
<td>MR</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Write 4 bytes to shadow stack.</td>
</tr>
<tr>
<td>66 REX.W OF 3B F5 l(11)rrr:bbb WRUSSQ m64, r64</td>
<td>MR</td>
<td>V/N.E.</td>
<td>CET_SS</td>
<td>Write 8 bytes to shadow stack.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Writes bytes in register source to a user shadow stack page. The WRUSS instruction can be executed only if CPL = 0, however the processor treats its shadow-stack accesses as user accesses.

### Operation

IF CR4.CET = 0  
    THEN #UD; Fl;
IF CPL > 0  
    THEN #GP(0); Fl;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)  
    THEN  
        (* Destination not 8B aligned *)  
        IF DEST_LA[2:0]  
            THEN GP(0); Fl;
            Shadow_stack_store 8 bytes of SRC to DEST_LA as user-mode access;
        ELSE  
            (* Destination not 4B aligned *)  
            IF DEST_LA[1:0]  
                THEN GP(0); Fl;
                Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA as user-mode access;
    Fl;

### Flags Affected

None.

### C/C++ Compiler Intrinsic Equivalent

`WRUSSD void _wrussd(__int32, void *);`
`WRUSSQ void _wrussq(__int64, void *);`
Protected Mode Exceptions

#UD  If the LOCK prefix is used.
    If CR4.CET = 0.

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If destination is located in a non-writeable segment.
    If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
    If linear address of destination is not 4 byte aligned.
    If CPL is not 0.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)  If destination is not a user shadow stack.
Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD  The WRUSS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD  The WRUSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD  If the LOCK prefix is used.
    If CR4.CET = 0.

#GP(0)  If a memory address is in a non-canonical form.
    If linear address of destination is not 4 byte aligned.
    If CPL is not 0.

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code)  If destination is not a user shadow stack.
Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD  If the LOCK prefix is used.
    If CR4.CET = 0.

#GP(0)  If a memory address is in a non-canonical form.
    If linear address of destination is not 4 byte aligned.
    If CPL is not 0.

#PF(fault-code)  If destination is not a user shadow stack.
Other terminal and non-terminal faults.
XABORT—Transactional Abort

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6 F8 ib XABORT imm8</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Causes an RTM abort if in RTM execution.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>imm8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

**Operation**

XABORT

IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
    FI;

(* For any RTM abort condition encountered during RTM execution *)

RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT:= 0;
    RTM_ACTIVE:= 0;
    SUSLDTRK_ACTIVE := 0;
    IF 64-bit Mode
        THEN
            RIP:= fallbackRIP;
        ELSE
            EIP := fallbackEIP;
        FI;
    END

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XABORT void _xabort( unsigned int);

**SIMD Floating-Point Exceptions**

None.
Other Exceptions

#UD

CPUID (EAX=7, ECX=0): EBX.RTM[bit 11] = 0.

If LOCK prefix is used.
XACQUIRE/XRELEASE—Hardware Lock Elision Prefix Hints

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 XACQUIRE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XACQUIRE-enabled” instruction to start lock elision on the instruction memory operand address.</td>
</tr>
<tr>
<td>F3 XRELEASE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XRELEASE-enabled” instruction to end lock elision on the instruction memory operand address.</td>
</tr>
</tbody>
</table>

NOTES:
1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.
Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to:

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to:

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

**XACQUIRE**

IF XACQUIRE-enabled instruction

THEN

IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN

HLE_NEST_COUNT++

IF (HLE_NEST_COUNT = 1) THEN

HLE_ACTIVE := 1

IF 64-bit mode

THEN

restartRIP := instruction pointer of the XACQUIRE-enabled instruction

ELSE

restartEIP := instruction pointer of the XACQUIRE-enabled instruction

FI;

Enter HLE Execution (* record register state, start tracking memory state *)

FI; (* HLE_NEST_COUNT = 1*)

IF ElisionBufferAvailable

THEN

Allocate elision buffer

Record address and data for forwarding and commit checking

Perform elision

ELSE

Perform lock acquire operation transactionally but without elision

FI;

ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT*)

GOTO HLE_ABORT_PROCESSING

FI;

ELSE

Treat instruction as non-XACQUIRE F2H prefixed legacy instruction

FI;
XRELEASE
IF XRELEASE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT > 0)
            THEN
                HLE_NEST_COUNT--
                IF lock address matches in elision buffer THEN
                    IF lock satisfies address and value requirements THEN
                        Deallocate elision buffer
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI;
        IF (HLE_NEST_COUNT = 0)
            THEN
                IF NoAllocatedElisionBuffer
                    THEN
                        Try to commit transactional execution
                        IF fail to commit transactional execution
                            THEN
                                GOTO HLE_ABORT_PROCESSING;
                            ELSE (* commit success *)
                                HLE_ACTIVE := 0
                            FI;
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                ELSE
                    GOTO HLE_ABORT_PROCESSING
                FI;
            FI; (* HLE_NEST_COUNT > 0 *)
        ELSE
            Treat instruction as non-XRELEASE F3H prefixed legacy instruction
        FI;
    (* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE := 0
    HLE_NEST_COUNT := 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP := restartRIP
        ELSE
            EIP := restartEIP
        FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
SIMD Floating-Point Exceptions
None.

Other Exceptions
#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.
XADD—Exchange and Add

**Description**

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

**IA-32 Architecture Compatibility**

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

**Operation**

\[
\text{TEMP} := \text{SRC} + \text{DEST};
\]
\[
\text{SRC} := \text{DEST};
\]
\[
\text{DEST} := \text{TEMP};
\]

**Flags Affected**

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

**Protected Mode Exceptions**

- **#GP(0)**  
  If the destination is located in a non-writable segment.
  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  If the DS, ES, FS, or GS register contains a NULL segment selector.

- **#SS(0)**  
  If a memory operand effective address is outside the SS segment limit.

- **#PF(fault-code)**  
  If a page fault occurs.

- **#AC(0)**  
  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

- **#UD**  
  If the LOCK prefix is used but the destination is not a memory operand.

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRM:reg (r, w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
Real-Address Mode Exceptions

#GP  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS  If a memory operand effective address is outside the SS segment limit.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD  If the LOCK prefix is used but the destination is not a memory operand.
XBEGIN—Transaction Begin

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C7 F8 XBEGIN rel16</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.</td>
</tr>
<tr>
<td>C7 F8 XBEGIN rel32</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.</td>
</tr>
</tbody>
</table>

**Description**

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort. (Use of the 16-bit operand size does not cause this address to be truncated to 16 bits, unlike a near jump to a relative offset.)

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

Execution of XBEGIN while in a suspend read address tracking region causes a transactional abort.

**Operation**

XBEGIN

IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT AND SUSLDTRK_ACTIVE = 0
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                    IF OperandSize = 16
                        THEN fallbackRIP := RIP + SignExtend64(rel16);
                    ELSE fallbackRIP := RIP + SignExtend64(rel32);
                    FI;
                    IF fallbackRIP is not canonical
                        THEN #GP(0);
                    FI;
                ELSE
                    IF OperandSize = 16
                        THEN fallbackEIP := EIP + SignExtend32(rel16);
                    ELSE fallbackEIP := EIP + rel32;
                    FI;
                    IF fallbackEIP outside code segment limit
                        THEN #GP(0);
                    FI;
            FI;
        FI;
    FI;

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Offset</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
RTM_ACTIVE := 1
Enter RTM Execution (* record register state, start tracking memory state *)
FI; (* RTM_NEST_COUNT = 1 *)
ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT OR SUSLDTRK_ACTIVE = 1 *)
    GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT := 0
    RTM_ACTIVE := 0
    SUSLDTRK_ACTIVE := 0
    IF 64-bit mode
        THEN
            RIP := fallbackRIP
        ELSE
            EIP := fallbackEIP
        FI;
    END

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XBEGIN unsigned int _xbegin( void );

SIMD Floating-Point Exceptions
None.

Protected Mode Exceptions
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
    If LOCK prefix is used.
#GP(0) If the fallback address is outside the CS segment.

Real-Address Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
    If LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
    If LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-bit Mode Exceptions

#UD  CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
     If LOCK prefix is used.

#GP(0)  If the fallback address is non-canonical.
XCHG—Exchange Register/Memory With Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>90+rw</td>
<td>XCHG AX, r16</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>90+rw</td>
<td>XCHG r16, AX</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange AX with r16.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG EAX, r32</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r32 with EAX.</td>
</tr>
<tr>
<td>REX+W + 90+rd</td>
<td>XCHG RAX, r64</td>
<td>O</td>
<td>N.E.</td>
<td></td>
<td>Exchange r64 with RAX.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG r32, EAX</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange EAX with r32.</td>
</tr>
<tr>
<td>REX+W + 90+rd</td>
<td>XCHG r64, RAX</td>
<td>O</td>
<td>N.E.</td>
<td></td>
<td>Exchange RAX with r64.</td>
</tr>
</tbody>
</table>

**Op/En** Operand 1 Operand 2 Operand 3 Operand 4

<table>
<thead>
<tr>
<th>Op/En</th>
<th>AX/EAX/RAX (r, w)</th>
<th>opcode + rd (r, w)</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>opcode + rd (r, w)</td>
<td>AX/EAX/RAX (r, w)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRM:reg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor’s locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See “Bus Locking” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.
NOTE

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation
TEMP := DEST;
DEST := SRC;
SRC := TEMP;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If either operand is in a non-writable segment.
If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.
XEND—Transaction End

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D5 XEND</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the end of an RTM code region.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

Execution of XEND outside a transactional region causes a general-protection exception (#GP). Execution of XEND while in a suspend read address tracking region causes a transactional abort.

### Operation

**XEND**

IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    IF SUSLDRTRK_ACTIVE = 1
        THEN GOTO RTM_ABORT_PROCESSING;
    FI;
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE := 0
            FI;
        FI;
    FI;

(* For any RTM abort condition encountered during RTM execution *)
**RTM_ABORT_PROCESSING:**
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT := 0
    RTM_ACTIVE := 0
    SUSLDRTRK_ACTIVE := 0
    IF 64-bit Mode
        THEN
RIP := fallbackRIP
ELSE
    EIP := fallbackEIP
    FI;
END

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XEND void _xend( void );

SIMD Floating-Point Exceptions
None.

Other Exceptions
#UD              CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
                 If LOCK prefix is used.
#GP(0)           If RTM_ACTIVE = 0.
XGETBV—Get Value of Extended Control Register

### Description
Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Use of any other value for ECX results in a general-protection (#GP) exception.

### Operation
\[
EDX:EAX := XCR[ECX];
\]

### Flags Affected
None.

### Intel C/C++ Compiler Intrinsic Equivalent
XGETBV unsigned __int64 __xgetbv( unsigned int);

### Protected Mode Exceptions
- **#GP(0)** If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.

### Real-Address Mode Exceptions
- **#GP(0)** If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.
**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**
Same exceptions as in protected mode.
**XLAT/XLATB—Table Look-up Translation**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D7</td>
<td>XLAT m8</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set AL to memory byte DS:[(E)BX + unsigned AL].</td>
</tr>
<tr>
<td>D7</td>
<td>XLATB</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Set AL to memory byte DS:[(E)BX + unsigned AL].</td>
</tr>
<tr>
<td>REX.w + D7</td>
<td>XLATB</td>
<td>ZO</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set AL to memory byte [RBX + unsigned AL].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

If AddressSize = 16
- THEN
  - AL := (DS:BX + ZeroExtend(AL));
- ELSE IF (AddressSize = 32)
  - AL := (DS:EBX + ZeroExtend(AL)); Fl;
- ELSE (AddressSize = 64)
  - AL := (RBX + ZeroExtend(AL));
- Fl;

**Flags Affected**

None.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS If a memory operand effective address is outside the SS segment limit.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.
### XOR—Logical Exclusive OR

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34 ib</td>
<td>XOR AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AL XOR imm8.</td>
</tr>
<tr>
<td>35 iw</td>
<td>XOR AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AX XOR imm16.</td>
</tr>
<tr>
<td>35 id</td>
<td>XOR EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>EAX XOR imm32.</td>
</tr>
<tr>
<td>REX.W + 35 id</td>
<td>XOR RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX XOR imm32 (sign-extended).</td>
</tr>
<tr>
<td>80 /6 ib</td>
<td>XOR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 XOR imm8.</td>
</tr>
<tr>
<td>REX + 80 /6 ib</td>
<td>XOR r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 XOR imm8.</td>
</tr>
<tr>
<td>81 /6 iw</td>
<td>XOR r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR imm16.</td>
</tr>
<tr>
<td>81 /6 id</td>
<td>XOR r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR imm32.</td>
</tr>
<tr>
<td>REX.W + 81 /6 id</td>
<td>XOR r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR imm32 (sign-extended).</td>
</tr>
<tr>
<td>83 /6 ib</td>
<td>XOR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>83 /6 id</td>
<td>XOR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>REX.W + 83 /6 ib</td>
<td>XOR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>30 /r</td>
<td>XOR r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 XOR r8.</td>
</tr>
<tr>
<td>REX + 30 /r</td>
<td>XOR r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 XOR r8.</td>
</tr>
<tr>
<td>31 /r</td>
<td>XOR r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR r16.</td>
</tr>
<tr>
<td>31 /r</td>
<td>XOR r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR r32.</td>
</tr>
<tr>
<td>REX.W + 31 /r</td>
<td>XOR r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR r64.</td>
</tr>
<tr>
<td>32 /r</td>
<td>XOR r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r8 XOR r/m8.</td>
</tr>
<tr>
<td>REX + 32 /r</td>
<td>XOR r8*, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r8 XOR r/m8.</td>
</tr>
<tr>
<td>33 /r</td>
<td>XOR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 XOR r/m16.</td>
</tr>
<tr>
<td>33 /r</td>
<td>XOR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r32 XOR r/m32.</td>
</tr>
<tr>
<td>REX.W + 33 /r</td>
<td>XOR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r64 XOR r/m64.</td>
</tr>
</tbody>
</table>

**NOTES:**
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (r, w)</td>
<td>imm8/16/32</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description
Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.
In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

\[
\text{DEST} := \text{DEST} \text{ XOR } \text{SRC};
\]

**Flags Affected**

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

**Protected Mode Exceptions**

- **#GP(0)** If the destination operand points to a non-writable segment.
- If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

- **#GP** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS** If a memory operand effective address is outside the SS segment limit.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
**XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 57/r XORPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical XOR of packed double precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 57 /r VXORPD xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed double precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 57 /r VXORPD ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed double precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 57 /r VXORPD zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical XOR of packed double precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N/A</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>B</td>
<td>N/A</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise logical XOR of the two, four or eight packed double precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**VEX.256 and EVEX.256 encoded versions:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 and EVEX.128 encoded versions:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
**Operation**

**VXORPD (EVEX Encoded Versions)**

\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]

FOR \(j := 0 \) TO \(KL-1\)

\[i := j \times 64\]

IF \(k1[j] \) OR *no writemask* \THEN

IF (EVEX.b == 1) AND (SRC2 *is memory*)

THEN \(\text{DEST}[i+63:j] := \text{SRC1}[i+63:j] \text{ BITWISE XOR SRC2}[63:0];\)

ELSE \(\text{DEST}[i+63:j] := \text{SRC1}[i+63:j] \text{ BITWISE XOR SRC2}[i+63:j];\)

FI;

ELSE

IF *merging-masking* \THEN \(\text{DEST}[i+63:j] \text{ remains unchanged};\)

ELSE *zeroing-masking* \THEN \(\text{DEST}[i+63:j] = 0;\)

FI

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

**VXORPD (VEX.256 Encoded Version)**

\(\text{DEST}[63:0] := \text{SRC1}[63:0] \text{ BITWISE XOR SRC2}[63:0]\)

\(\text{DEST}[127:64] := \text{SRC1}[127:64] \text{ BITWISE XOR SRC2}[127:64]\)


\(\text{DEST}[255:192] := \text{SRC1}[255:192] \text{ BITWISE XOR SRC2}[255:192]\)

\(\text{DEST}[\text{MAXVL}-1:128] := 0\)

**VXORPD (VEX.128 Encoded Version)**

\(\text{DEST}[63:0] := \text{SRC1}[63:0] \text{ BITWISE XOR SRC2}[63:0]\)

\(\text{DEST}[127:64] := \text{SRC1}[127:64] \text{ BITWISE XOR SRC2}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] := 0\)

**XORPD (128-bit Legacy SSE Version)**

\(\text{DEST}[63:0] := \text{DEST}[63:0] \text{ BITWISE XOR SRC}[63:0]\)

\(\text{DEST}[127:64] := \text{DEST}[127:64] \text{ BITWISE XOR SRC}[127:64]\)

\(\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}\)

**Intel C/C++ Compiler Intrinsic Equivalent**

\(\text{VXORPD } \text{__m512d } \text{__mm512_xor_pd } (\text{__m512d a, __m512d b});\)

\(\text{VXORPD } \text{__m512d } \text{__mm512_mask_xor_pd } (\text{__m512d a, __mmask8 m, __m512d b});\)

\(\text{VXORPD } \text{__m512d } \text{__mm512_maskz_xor_pd } (\text{__mmask8 m, __m512d a});\)

\(\text{VXORPD } \text{__m256d } \text{__mm256_xor_pd } (\text{__m256d a, __m256d b});\)

\(\text{VXORPD } \text{__m256d } \text{__mm256_mask_xor_pd } (\text{__m256d a, __mmask8 m, __m256d b});\)

\(\text{VXORPD } \text{__m256d } \text{__mm256_maskz_xor_pd } (\text{__mmask8 m, __m256d a});\)

\(\text{XORPD } \text{__m128d } \text{__mm128_xor_pd } (\text{__m128d a, __m128d b});\)

\(\text{VXORPD } \text{__m128d } \text{__mm128_mask_xor_pd } (\text{__m128d a, __mmask8 m, __m128d b});\)

\(\text{VXORPD } \text{__m128d } \text{__mm128_maskz_xor_pd } (\text{__mmask8 m, __m128d a});\)

**SIMD Floating-Point Exceptions**

None.
Other Exceptions
Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”
EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions.”
XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 57 /r</td>
<td>A</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.0F.WL 57 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.0F.WL 57 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.0F.W0 57 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 57 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 57 /r</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

| Instruction Operand Encoding |
|-------------------------------|---|---|---|---|
| Op/En | Tuple Type | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
| A      | N/A         | ModRMreg (r, w) | ModRMreg (r) | N/A | N/A |
| B      | N/A         | ModRMreg (w) | VEX.vvvv (r) | ModRMreg/r (r) | N/A |
| C      | Full        | ModRMreg (w) | VEX.vvvv (r) | ModRMreg/r (r) | N/A |

Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.
Operation

**VXORPS (EVEX Encoded Versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[31:0];
      ELSE DEST[i+31:i] := SRC1[i+31:i] BITWISE XOR SRC2[i+31:31];
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] = 0
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VXORPS (VEX.256 Encoded Version)**
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:256] := 0

**VXORPS (VEX.128 Encoded Version)**
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:128] := 0

**XORPS (128-bit Legacy SSE Version)**
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)
Intel C++ Compiler Intrinsic Equivalent

VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 _mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 _mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 _mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 _mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 _mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions."
EVEX-encoded instructions, see Table 2-49, "Type E4 Class Exception Conditions."
XRESLDTRK—Resume Tracking Load Addresses

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 01 E9 XRESLDTRK</td>
<td>ZO</td>
<td>V/V</td>
<td>TSXLDTRK</td>
<td>Specifies the end of an Intel TSX suspend read address tracking region.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The instruction marks the end of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a suspend load address tracking region it will end the suspend region and all following load addresses will be added to the transaction read set. If this instruction is used inside an active transaction but not in a suspend region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.


**Operation**

XRESLDTRK

IF RTM_ACTIVE = 1:
   IF SUSLDTRK_ACTIVE = 1:
      SUSLDTRK_ACTIVE := 0
   ELSE:
      RTM_ABORT
ELSE:
   NOP

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XRESLDTRK void _xresldtrk(void);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

#UD  
If CPUID.(EAX=7, ECX=0):EDX.TSXLDTRK[bit 16] = 0.  
If the LOCK prefix is used.
XRSTOR—Restore Processor Extended States

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /5 XRSTOR mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVE</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /5 XRSTOR64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVE</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.8, “Operation of XRSTOR,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[i] = 0, XRSTOR does not update state component i.¹
- If RFBM[i] = 1 and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component i.
- If RFBM[i] = 1 and XSTATE_BV[i] = 1, XRSTOR loads state component i from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

¹. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.
Operation

RFBM := XCR0 AND EDX:EAX;  /* bitwise logical AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED := RFBM AND RSTORMASK;
        TO_BE_INITIALIZED := RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                XINUSE[0] := 1;
                load x87 state from legacy region of XSAVE area;
            ELSIF TO_BE_INITIALIZED[0] = 1
                THEN
                    XINUSE[0] := 0;
                    initialize x87 state;
            FI;

            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                XINUSE[1] := 1;
                load XMM registers from legacy region of XSAVE area;  // this step does not load MXCSR
            ELSIF TO_BE_INITIALIZED[1] = 1
                THEN
                    XINUSE[1] := 0;
                    set all XMM registers to 0;  // this step does not initialize MXCSR
            FI;

        FOR i := 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset n from base of XSAVE area;
                    // n enumerated by CPUID(EAX=0DH,ECX=i):EBX
                ELSIF TO_BE_INITIALIZED[i] = 1
                    THEN
                        XINUSE[i] := 0;
                        initialize XSAVE state component i;
                FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN  /* compacted form not supported */
                #GP(0);
        FI;
    FI;
FORMAT = COMPMASK AND 7FFFFFFF_80000000;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
  THEN
    XINUSE[0] := 1;
    load x87 state from legacy region of XSAVE area;
ELSIF TO_BE_INITIALIZED[0] = 1
  THEN
    XINUSE[0] := 0;
    initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
  THEN
    XINUSE[1] := 1;
    load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
ELSIF TO_BE_INITIALIZED[1] = 1
  THEN
    set all XMM registers to 0;
    XINUSE[1] := 0;
    MXCSR := 1F80H;
FI;

NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
  IF FORMAT[i] = 1
    THEN
      IF TO_BE_RESTORED[i] = 1
        THEN
          XINUSE[i] := 1;
          load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
        FI;
      NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
  IF TO_BE_INITIALIZED[i] = 1
    THEN
      XINUSE[i] := 0;
      initialize XSAVE state component i;
  FI;
ENDFOR;
FI;

XMODIFIED := NOT RFBM:

IF in VMX non-root operation
  THEN VMXNR := 1;
  ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := (CPL,VMXRNR,LAXA,COMPMASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTOR void _xrstor( void *, unsigned __int64);
XRSTOR void _xrstor64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XCOMP_BV field of the XSAVE header is 1.
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0
and the corresponding bit in the XSTATE_BV field is 1.
If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 64-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).

Real-Address Mode Exceptions
#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand outside the effective address space from 0 to FFFFH.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XCOMP_BV field of the XSAVE header is 1.

#AC
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If a memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the
XCOMP_BV field of the XSAVE header is 1.
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0
and the corresponding bit in the XSTATE_BV field is 1.
If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 64-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).
XRSTORS—Restore Processor Extended States Supervisor

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.12, "Operation of XRSTORS," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[\(i\)] = 0, XRSTORS does not update state component \(i\).
- If RFBM[\(i\)] = 1 and bit \(i\) is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component \(i\).
- If RFBM[\(i\)] = 1 and XSTATE_BV[\(i\)] = 1, XRSTORS loads state component \(i\) from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component \(i\) for which RFBM[\(i\)] = 1 and XSTATE_BV[\(i\)] = 1; it tracks as modified any state component \(i\) for which RFBM[\(i\)] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m(r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Opcode / Instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C7 /3 XRSTORS mem</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /3 XRSTORS64 mem</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
</tbody>
</table>
Operation

RFBM := (XCR0 OR IA32_XSS) AND EDX:EAX; /* bitwise logical OR and AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
THEN
    XINUSE[0] := 1;
    load x87 state from legacy region of XSAVE area;
ELSIF TO_BE_INITIALIZED[0] = 1
THEN
    XINUSE[0] := 0;
    initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
THEN
    XINUSE[1] := 1;
    load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
ELSIF TO_BE_INITIALIZED[1] = 1
THEN
    set all XMM registers to 0;
    XINUSE[1] := 0;
    MXCSR := 1F80H;
FI;

NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF FORMAT[i] = 1
    THEN
        IF TO_BE_RESTORED[i] = 1
        THEN
            XINUSE[i] := 1;
            load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
        FI;
        NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
    THEN
        XINUSE[i] := 0;
        initialize XSAVE state component i;
    FI;
ENDFOR;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
THEN VMXNR := 1;

XRSTORS—Restore Processor Extended States Supervisor
ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := \langle CPL, VMXNR, LAXA, COMPRESS \rangle;

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTORS void _xrstors( void *, unsigned __int64);
XRSTORS64 void _xrstors64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) If CPL > 0.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
   If a bit in XCR0[IA32_XSS] is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
   If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
   If bytes 63:16 of the XSAVE header are not all zero.
   If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If any part of the operand lies outside the effective address space from 0 to FFFFH.
   If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
   If a bit in XCR0[IA32_XSS] is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
   If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
   If bytes 63:16 of the XSAVE header are not all zero.
   If attempting to write any reserved bits of the MXCSR register with 1.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If CPL > 0.
         If a memory address is in a non-canonical form.
         If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
         If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
         If a bit in XCR0[IA32_XSS] is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
         If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
         If bytes 63:16 of the XSAVE header are not all zero.
         If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code)  If a page fault occurs.

#NM  If CR0.TS[bit 3] = 1.

#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
         If CR4.OSXSAVE[bit 18] = 0.
         If the LOCK prefix is used.
**XSAVE—Save Processor Extended States**

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /4 XSAVE mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVE</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /4 XSAVE64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVE</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1:

Section 13.7, “Operation of XSAVE,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component $i$ if and only if RFBM[$i$] = 1.
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) and writes a modified value back to memory as follows. If RFBM[$i$] = 1, XSAVE writes XSTATE_BV[$i$] with the value of XINUSE[$i$]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) If RFBM[$i$] = 0, XSAVE writes XSTATE_BV[$i$] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

---

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.
**Operation**

```
RFBM := XCR0 AND EDX:EAX; /* bitwise logical AND */
OLD_BV := XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

```
XSAVE void _xsave( void *, unsigned __int64);
XSAVE void _xsave64( void *, unsigned __int64);
```

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
- **#PF(fault-code)** If a page fault occurs.
- **#NM** If CR0.TS[bit 3] = 1.
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  If CR4.OSXSAVE[bit 18] = 0.
  If the LOCK prefix is used.
- **#AC** If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC
If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSAVEC—Save Processor Extended States With Compaction

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C7 /4 XSAVEC mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEC</td>
<td>Save state components specified by EDX:EAX to mem with compaction.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /4 XSAVEC64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVEC</td>
<td>Save state components specified by EDX:EAX to mem with compaction.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.10, "Operation of XSAVEC," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component $i$ if and only if $\text{RFBM}[i] = 1$ and $\text{XINUSE}[i] = 1$.1 (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.$^{2,3}$ (See Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

---

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.
Operation

\[
\begin{align*}
\text{RFBM} & := \text{XCR0 AND EDX:EAX}; & /* \text{bitwise logical AND */} \\
\text{TO\_BE\_SAVED} & := \text{RFBM AND XINUSE}; & /* \text{bitwise logical AND */} \\
\text{IF MXCSR} & \neq 1F80H \text{ AND RFBM}[1] \\
& \quad \text{TO\_BE\_SAVED}[1] = 1; \\
& \quad \text{FI}; \\
\text{IF TO\_BE\_SAVED}[0] = 1 \\
& \quad \text{THEN store x87 state into legacy region of XSAVE area;}
\end{align*}
\]

\[
\begin{align*}
\text{FI;}
\text{IF TO\_BE\_SAVED}[1] = 1 \\
& \quad \text{THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR\_MASK} \\
& \quad \text{FI;}
\end{align*}
\]

\[
\begin{align*}
\text{NEXT\_FEATURE\_OFFSET} & = 576; & /* \text{Legacy area and XSAVE header consume 576 bytes} \\
& \text{FOR i := 2 TO 62} \\
& \quad \text{IF RFBM}[i] = 1 \\
& \quad \quad \text{THEN} \\
& \quad \quad \quad \text{IF TO\_BE\_SAVED}[i] \\
& \quad \quad \quad \quad \text{THEN save XSAVE state component i at offset NEXT\_FEATURE\_OFFSET from base of XSAVE area;}
\end{align*}
\]

\[
\begin{align*}
& \quad \quad \quad \quad \text{FI;}
& \quad \quad \quad \quad \text{NEXT\_FEATURE\_OFFSET} = \text{NEXT\_FEATURE\_OFFSET} + n (n \text{ enumerated by CPUID}(EAX=0DH,ECX=i):EAX); \\
& \quad \quad \quad \quad \text{FI;}
& \text{ENDFOR;}
\end{align*}
\]

\[
\begin{align*}
\text{XSTATE\_BV field in XSAVE header} & := \text{TO\_BE\_SAVED}; \\
\text{XCOMP\_BV field in XSAVE header} & := \text{RFBM OR 80000000_00000000H};
\end{align*}
\]

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{XSAVE} & \text{ void } _{\text{xsavec( void * , unsigned __int64);}} \\
\text{XSAVE} & \text{64 void } _{\text{xsavec64( void * , unsigned __int64);}}
\end{align*}
\]

Protected Mode Exceptions

\[
\begin{align*}
\#GP(0) & \text{ If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.} \\
& \text{ If a memory operand is not aligned on a 64-byte boundary, regardless of segment.} \\
\#SS(0) & \text{ If a memory operand effective address is outside the SS segment limit.} \\
\#PF(fault-code) & \text{ If a page fault occurs.} \\
\#NM & \text{ If CR0.TS[bit 3] = 1.} \\
\#UD & \text{ If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.} \\
& \text{ If CR4.OSXSAVE[bit 18] = 0.} \\
& \text{ If the LOCK prefix is used.}
\end{align*}
\]
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.

If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSAVEOPT—Save Processor Extended States Optimized

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F AE /6 XSAVEOPT mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEOPT</td>
<td>Save state components specified by EDX:EAX to mem, optimizing if possible.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /6 XSAVEOPT64 mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEOPT</td>
<td>Save state components specified by EDX:EAX to mem, optimizing if possible.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r, w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.9, “Operation of XSAVEOPT,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component \( i \) only if \( \text{RFBM}[i] = 1 \) and \( \text{XINUSE}[i] = 1 \).\(^1\) (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) Even if both bits are 1, XSAVEOPT may optimize and not save state component \( i \) if (1) state component \( i \) has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) and writes a modified value back to memory as follows. If \( \text{RFBM}[i] = 1 \), XSAVEOPT writes XSTATE_BV[\( i \)] with the value of XINUSE[\( i \)]. If \( \text{RFBM}[i] = 0 \), XSAVEOPT writes XSTATE_BV[\( i \)] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

---

\(^1\) There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if \( \text{RFBM}[1] = 1 \) or \( \text{RFBM}[2] = 1 \), regardless of the value of XINUSE.
See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

**Operation**

```plaintext
RFBM := XCR0 AND EDX:EAX; /* bitwise logical AND */
OLD_BV := XSTATE_BV field from XSAVE header;
TO_BE_SAVED := RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;

LAXA := linear address of XSAVE area;
IF XRSTOR_INFO = \\
    \{(CPL,VMXNR,LAXA,00000000_00000000H) \}
    THEN TO_BE_SAVED := TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i := 2 TO 62
    IF TO_BE_SAVED[i] = 1
    THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
ENDFOR;

XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XSAVEOPT void _xsaveopt( void *, unsigned __int64);
XSAVEOPT void _xsaveopt64( void *, unsigned __int64);

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD
If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC
If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM
If CR0.TS[bit 3] = 1.

#UD
If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)
If a memory address referencing the SS segment is in a non-canonical form.

#GP(0)
If the memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#PF(fault-code)
If a page fault occurs.

#NM
If CR0.TS[bit 3] = 1.

#UD
If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC
If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSAVES—Save Processor Extended States Supervisor

<table>
<thead>
<tr>
<th>Opcode / Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP C7 /5 XSAVES mem</td>
<td>M</td>
<td>V/V</td>
<td>XSS</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /5 XSAVES64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSS</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMr/m (w)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVE may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.11, “Operation of XSAVES,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component \( i \) only if RFBM[\( i \)] = 1 and XINUSE[\( i \)] = 1.\(^1\) (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) Even if both bits are 1, XSAVES may optimize and not save state component \( i \) if (1) state component \( i \) has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.\(^2\) (See Section 13.4.2, “XSAVE Header,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

---

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.

2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.
See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

**Operation**

\[ RFBM := (XCR0 \lor IA32\_XSS) \land EDX:EAX; \] /* bitwise logical OR and AND */

IF in VMX non-root operation
  THEN VMXNR := 1;
  ELSE VMXNR := 0;
FI;

LAXA := linear address of XSAVE area;

COMPMASK := RFBM OR 80000000_00000000H;

TO\_BE\_SAVED := RFBM AND XINUSE;

IF XRSTOR\_INFO = (CPL,VMXNR,LAXA,COMPMASK)
  THEN TO\_BE\_SAVED := TO\_BE\_SAVED AND XMODIFIED;
FI;

IF MXCSR \neq 1F80H AND RFBM[1]
  THEN TO\_BE\_SAVED[1] = 1;
FI;

IF TO\_BE\_SAVED[0] = 1
  THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO\_BE\_SAVED[1] = 1
  THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR\_MASK
FI;

NEXT\_FEATURE\_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes

FOR i := 2 TO 62
  IF RFBM[i] = 1
    THEN
      IF TO\_BE\_SAVED[i]
        THEN
          save XSAVE state component i at offset NEXT\_FEATURE\_OFFSET from base of XSAVE area;
          IF i = 8 // state component 8 is for PT state
            THEN IA32\_RTIT\_CTL.TraceEn[bit 0] := 0;
          FI;
        FI;
    FI;

  NEXT\_FEATURE\_OFFSET = NEXT\_FEATURE\_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
ENDFOR;

NEW\_HEADER := RFBM AND XINUSE;

IF MXCSR \neq 1F80H AND RFBM[1]
  THEN NEW\_HEADER[1] = 1;
FI;

XSTATE\_BV field in XSAVE header := NEW\_HEADER;
XCOMP\_BV field in XSAVE header := COMPMASK;

**Flags Affected**

None.
Intel C/C++ Compiler Intrinsic Equivalent

XSAVES void _xsaves( void *, unsigned __int64);
XSAVES64 void _xsaves64( void *, unsigned __int64);

Protected Mode Exceptions

#GP(0) If CPL > 0.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If CPL > 0.
   If the memory address is in a non-canonical form.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.
XSETBV—Set Extended Control Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D1</td>
<td>XSETBV</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Write the value in EDX:EAX to the XCR specified by ECX.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, “Enabling the XSAVE Feature Set and XSAVE-Enabled Features,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

**Operation**

XCR[ECX] := EDX:EAX;

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XSETBV void _xsetbv( unsigned int, unsigned __int64);

**Protected Mode Exceptions**

#GP(0) If the current privilege level is not 0.
If an invalid XCR is specified in ECX.
If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
If an attempt is made to clear bit 0 of XCR0.
If an attempt is made to set XCR0[2:1] to 10b.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
Real-Address Mode Exceptions

#GP          If an invalid XCR is specified in ECX.
If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
If an attempt is made to clear bit 0 of XCR0.
If an attempt is made to set XCR0[2:1] to 10b.

#UD          If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)       The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
XSUSLDTRK—Suspend Tracking Load Addresses

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 0F 01 E8</td>
<td>ZO</td>
<td>V/V</td>
<td>TSXLDTTRK</td>
<td>Specifies the start of an Intel TSX suspend read address tracking region.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

The instruction marks the start of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a transactional region, subsequent loads are not added to the read set of the transaction. If the instruction is used inside a suspend load address tracking region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides additional information on Intel® TSX Suspend Load Address Tracking.

**Operation**

**XSUSLDTRK**

IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 0:
        SUSLDTRK_ACTIVE := 1
    ELSE:
        RTM_ABORT
ELSE:
    NOP

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

XSUSLDTRK void _xsusldtrk(void);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

#UD If CPUID.(EAX=7, ECX=0):EDX.TSXLDTTRK[bit 16] = 0.

If the LOCK prefix is used.
XTEST—Test if in Transactional Execution

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D6 XTEST</td>
<td>ZO</td>
<td>V/V</td>
<td>0</td>
<td>Test if executing in a transactional region.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

```
XTEST
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
   THEN
      ZF := 0
   ELSE
      ZF := 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

```
XTEST int _xtest( void );
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD CPUID.(EAX=7, ECX=0):EBX.HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
If LOCK prefix is used.
7.1 OVERVIEW

This chapter describes the Safer Mode Extensions (SMX) for the Intel 64 and IA-32 architectures. Safer Mode Extensions (SMX) provide a programming interface for system software to establish a measured environment within the platform to support trust decisions by end users. The measured environment includes:

- Measured launch of a system executive, referred to as a Measured Launched Environment (MLE)\(^1\). The system executive may be based on a Virtual Machine Monitor (VMM), a measured VMM is referred to as MVMM\(^2\).
- Mechanisms to ensure the above measurement is protected and stored in a secure location in the platform.
- Protection mechanisms that allow the VMM to control attempts to modify the VMM.

The measurement and protection mechanisms used by a measured environment are supported by the capabilities of an Intel® Trusted Execution Technology (Intel® TXT) platform:

- The SMX are the processor’s programming interface in an Intel TXT platform.
- The chipset in an Intel TXT platform provides enforcement of the protection mechanisms.
- Trusted Platform Module (TPM) 1.2 in the platform provides platform configuration registers (PCRs) to store software measurement values.

7.2 SMX FUNCTIONALITY

SMX functionality is provided in an Intel 64 processor through the GETSEC instruction via leaf functions. The GETSEC instruction supports multiple leaf functions. Leaf functions are selected by the value in EAX at the time GETSEC is executed. Each GETSEC leaf function is documented separately in the reference pages with a unique mnemonic (even though these mnemonics share the same opcode, 0F 37).

7.2.1 Detecting and Enabling SMX

Software can detect support for SMX operation using the CPUID instruction. If software executes CPUID with 1 in EAX, a value of 1 in bit 6 of ECX indicates support for SMX operation (GETSEC is available), see CPUID instruction for the layout of feature flags of reported by CPUID.01H:ECX.

System software enables SMX operation by setting CR4.SMXE[Bit 14] = 1 before attempting to execute GETSEC. Otherwise, execution of GETSEC results in the processor signaling an invalid opcode exception (#UD).

If the CPUID SMX feature flag is clear (CPUID.01H.ECX[Bit 6] = 0), attempting to set CR4.SMXE[Bit 14] results in a general protection exception.

The IA32_FEATURE_CONTROL MSR (at address 03AH) provides feature control bits that configure operation of VMX and SMX. These bits are documented in Table 7-1.

---

1. See the Intel® Trusted Execution Technology Measured Launched Environment Programming Guide.
2. An MVMM is sometimes referred to as a measured launched environment (MLE). See the Intel® Trusted Execution Technology Measured Launched Environment Programming Guide.
SAFER MODE EXTENSIONS REFERENCE

7.2.2 SMX Instruction Summary

System software must first query for available GETSEC leaf functions by executing GETSEC[CAPABILITIES]. The CAPABILITIES leaf function returns a bit map of available GETSEC leaves. An attempt to execute an unsupported leaf index results in an undefined opcode (#UD) exception.

Table 7-1. Layout of IA32_FEATURE_CONTROL

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lock bit (0 = unlocked, 1 = locked). When set to ‘1’ further writes to this MSR are blocked.</td>
</tr>
<tr>
<td>1</td>
<td>Enable VMX in SMX operation.</td>
</tr>
<tr>
<td>2</td>
<td>Enable VMX outside SMX operation.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>14:8</td>
<td>SENTER Local Function Enables: When set, each bit in the field represents an enable control for a corresponding SENTER function.</td>
</tr>
<tr>
<td>15</td>
<td>SENTER Global Enable: Must be set to ‘1’ to enable operation of GETSEC[SENTER].</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
</tr>
<tr>
<td>17</td>
<td>SGX Launch Control Enable: Must be set to ‘1’ to enable runtime re-configuration of SGX Launch Control via the IA32_SGXLEPUBKEYHASHn MSR.</td>
</tr>
<tr>
<td>18</td>
<td>SGX Global Enable: Must be set to ‘1’ to enable Intel SGX leaf functions.</td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>LMCE On: When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.</td>
</tr>
<tr>
<td>63:21</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
7.2.2.1 GETSEC[CAPABILITIES]

The SMX functionality provides an architectural interface for newer processor generations to extend SMX capabilities. Specifically, the GETSEC instruction provides a capability leaf function for system software to discover the available GETSEC leaf functions that are supported in a processor. Table 7-2 lists the currently available GETSEC leaf functions.

Table 7-2. GETSEC Leaf Functions

<table>
<thead>
<tr>
<th>Index (EAX)</th>
<th>Leaf function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CAPABILITIES</td>
<td>Returns the available leaf functions of the GETSEC instruction.</td>
</tr>
<tr>
<td>1</td>
<td>Undefined</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>ENTERACCS</td>
<td>Enter</td>
</tr>
<tr>
<td>3</td>
<td>EXITAC</td>
<td>Exit</td>
</tr>
<tr>
<td>4</td>
<td>SENTER</td>
<td>Launch an MLE.</td>
</tr>
<tr>
<td>5</td>
<td>SEXIT</td>
<td>Exit the MLE.</td>
</tr>
<tr>
<td>6</td>
<td>PARAMETERS</td>
<td>Return SMX related parameter information.</td>
</tr>
<tr>
<td>7</td>
<td>SMCTRL</td>
<td>SMX mode control.</td>
</tr>
<tr>
<td>8</td>
<td>WAKEUP</td>
<td>Wake up sleeping processors in safer mode.</td>
</tr>
<tr>
<td>9 - (4G-1)</td>
<td>Undefined</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

7.2.2.2 GETSEC[ENTERACCS]

The GETSEC[ENTERACCS] leaf enables authenticated code execution mode. The ENTERACCS leaf function performs an authenticated code module load using the chipset public key as the signature verification. ENTERACCS requires the existence of an Intel® Trusted Execution Technology capable chipset since it unlocks the chipset private configuration register space after successful authentication of the loaded module. The physical base address and size of the authenticated code module are specified as input register values in EBX and ECX, respectively.

While in the authenticated code execution mode, certain processor state properties change. For this reason, the time in which the processor operates in authenticated code execution mode should be limited to minimize impact on external system events.

Upon entry into , the previous paging context is disabled (since the authenticated code module image is specified with physical addresses and can no longer rely upon external memory-based page-table structures).

Prior to executing the GETSEC[ENTERACCS] leaf, system software must ensure the logical processor issuing GETSEC[ENTERACCS] is the boot-strap processor (BSP), as indicated by IA32_APIC_BASE.BSP = 1. System software must ensure other logical processors are in a suitable idle state and not marked as BSP.

The GETSEC[ENTERACCS] leaf may be used by different agents to load different authenticated code modules to perform functions related to different aspects of a measured environment, for example system software and Intel® TXT enabled BIOS may use more than one authenticated code modules.

7.2.2.3 GETSEC[EXITAC]

GETSEC[EXITAC] takes the processor out of authenticated code execution mode. When this instruction leaf is executed, the contents of the authenticated code execution area are scrubbed and control is transferred to the non-authenticated context defined by a near pointer passed with the GETSEC[EXITAC] instruction.

The authenticated code execution area is no longer accessible after completion of GETSEC[EXITAC]. RBX (or EBX) holds the address of the near absolute indirect target to be taken.
7.2.2.4 GETSEC[SENTER]

The GETSEC[SENTER] leaf function is used by the initiating logical processor (ILP) to launch an MLE. GETSEC[SENTER] can be considered a superset of the ENTERACCS leaf, because it enters as part of the measured environment launch.

Measured environment startup consists of the following steps:

- the ILP rendezvous the responding logical processors (RLPs) in the platform into a controlled state (At the completion of this handshake, all the RLPs except for the ILP initiating the measured environment launch are placed in a newly defined SENTER sleep state).
- Load and authenticate the authenticated code module required by the measured environment, and enter authenticated code execution mode.
- Verify and lock certain system configuration parameters.
- Measure the dynamic root of trust and store into the PCRs in TPM.
- Transfer control to the MLE with interrupts disabled.

Prior to executing the GETSEC[SENTER] leaf, system software must ensure the platform’s TPM is ready for access and the ILP is the boot-strap processor (BSP), as indicated by IA32_APIC_BASE.BSP. System software must ensure other logical processors (RLPs) are in a suitable idle state and not marked as BSP.

System software launching a measurement environment is responsible for providing a proper authenticate code module address when executing GETSEC[SENTER]. The AC module responsible for the launch of a measured environment and loaded by GETSEC[SENTER] is referred to as SINIT. See Intel® Trusted Execution Technology Measured Launched Environment Programming Guide for additional information on system software requirements prior to executing GETSEC[SENTER].

7.2.2.5 GETSEC[SEXIT]

System software exits the measured environment by executing the instruction GETSEC[SEXIT] on the ILP. This instruction rendezvous the responding logical processors in the platform for exiting from the measured environment. External events (if left masked) are unmasked and Intel® TXT-capable chipset’s private configuration space is re-locked.

7.2.2.6 GETSEC[PARAMETERS]

The GETSEC[PARAMETERS] leaf function is used to report attributes, options, and limitations of SMX operation. Software uses this leaf to identify operating limits or additional options.

The information reported by GETSEC[PARAMETERS] may require executing the leaf multiple times using EBX as an index. If the GETSEC[PARAMETERS] instruction leaf or if a specific parameter field is not available, then SMX operation should be interpreted to use the default limits of respective GETSEC leaves or parameter fields defined in the GETSEC[PARAMETERS] leaf.

7.2.2.7 GETSEC[SMCTRL]

The GETSEC[SMCTRL] leaf function is used for providing additional control over specific conditions associated with the SMX architecture. An input register is supported for selecting the control operation to be performed. See the specific leaf description for details on the type of control provided.

7.2.2.8 GETSEC[WAKEUP]

Responding logical processors (RLPs) are placed in the SENTER sleep state after the initiating logical processor executes GETSEC[SENTER]. The ILP can wake up RLPs to join the measured environment by using GETSEC[WAKEUP]. When the RLPs in SENTER sleep state wake up, these logical processors begin execution at the entry point defined in a data structure held in system memory (pointed to by an chipset register LT.MLE.JOIN) in TXT configuration space.
7.2.3 Measured Environment and SMX

This section gives a simplified view of a representative life cycle of a measured environment that is launched by a system executive using SMX leaf functions. The Intel® Trusted Execution Technology Measured Launched Environment Programming Guide provides more detailed examples of using SMX and chipset resources (including chipset registers, Trusted Platform Module) to launch an MVMM.

The life cycle starts with the system executive (an OS, an OS loader, and so forth) loading the MLE and SINIT AC module into available system memory. The system executive must validate and prepare the platform for the measured launch. When the platform is properly configured, the system executive executes GETSEC[SENTER] on the initiating logical processor (ILP) to rendezvous the responding logical processors into an SENTER sleep state, the ILP then enters into using the SINIT AC module. In a multi-threaded or multi-processing environment, the system executive must ensure that other logical processors are already in an idle loop, or asleep (such as after executing HLT) before executing GETSEC[SENTER].

After the GETSEC[SENTER] rendezvous handshake is performed between all logical processors in the platform, the ILP loads the chipset authenticated code module (SINIT) and performs an authentication check. If the check passes, the processor hashes the SINIT AC module and stores the result into TPM PCR 17. It then switches execution context to the SINIT AC module. The SINIT AC module will perform a number of platform operations, including: verifying the system configuration, protecting the system memory used by the MLE from I/O devices capable of DMA, producing a hash of the MLE, storing the hash value in TPM PCR 18, and various other operations. When SINIT completes execution, it executes the GETSEC[EXITAC] instruction and transfers control the MLE at the designated entry point.

Upon receiving control from the SINIT AC module, the MLE must establish its protection and isolation controls before enabling DMA and interrupts and transferring control to other software modules. It must also wake up the RLPs from their SENTER sleep state using the GETSEC[WAKEUP] instruction and bring them into its protection and isolation environment.

While executing in a measured environment, the MVMM can access the Trusted Platform Module (TPM) in locality 2. The MVMM has complete access to all TPM commands and may use the TPM to report current measurement values or use the measurement values to protect information such that only when the platform configuration registers (PCRs) contain the same value is the information released from the TPM. This protection mechanism is known as sealing.

A measured environment shutdown is ultimately completed by executing GETSEC[SEXIT]. Prior to this step system software is responsible for scrubbing sensitive information left in the processor caches, system memory.

7.3 GETSEC LEAF FUNCTIONS

This section provides detailed descriptions of each leaf function of the GETSEC instruction. GETSEC is available only if CPUID.01H:ECX[Bit 6] = 1. This indicates the availability of SMX and the GETSEC instruction. Before GETSEC can be executed, SMX must be enabled by setting CR4.SMXE[Bit 14] = 1.

A GETSEC leaf can only be used if it is shown to be available as reported by the GETSEC[CAPABILITIES] function. Attempts to access a GETSEC leaf index not supported by the processor, or if CR4.SMXE is 0, results in the signaling of an undefined opcode exception.

All GETSEC leaf functions are available in protected mode, including the compatibility sub-mode of IA-32e mode and the 64-bit sub-mode of IA-32e mode. Unless otherwise noted, the behavior of all GETSEC functions and interactions related to the measured environment are independent of IA-32e mode. This also applies to the interpretation of register widths1 passed as input parameters to GETSEC functions and to register results returned as output parameters.

---

1. This chapter uses the 64-bit notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because processors that support SMX also support Intel 64 Architecture. The MVMM can be launched in IA-32e mode or outside IA-32e mode. The 64-bit notation of processor registers also refer to its 32-bit forms if SMX is used in 32-bit environment. In some places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
The GETSEC functions ENTERACCS, SENTER, SEXIT, and WAKEUP require an Intel® TXT capable-chipset to be present in the platform. The GETSEC[CAPABILITIES] returned bit vector in position 0 indicates an Intel® TXT-capable chipset has been sampled present\(^1\) by the processor.

The processor’s operating mode also affects the execution of the following GETSEC leaf functions: SMCTRL, ENTERACCS, EXITAC, SENTER, SEXIT, and WAKEUP. These functions are only allowed in protected mode at CPL = 0. They are not allowed while in SMM in order to prevent potential intra-mode conflicts. Further execution qualifications exist to prevent potential architectural conflicts (for example: nesting of the measured environment or authenticated code execution mode). See the definitions of the GETSEC leaf functions for specific requirements.

For the purpose of performance monitor counting, the execution of GETSEC functions is counted as a single instruction with respect to retired instructions. The response by a responding logical processor (RLP) to messages associated with GETSEC[SENTER] or GTSEC[SEXIT] is transparent to the retired instruction count on the ILP.

---

\(^1\) Sampled present means that the processor sent a message to the chipset and the chipset responded that it (a) knows about the message and (b) is capable of executing SENTER. This means that the chipset CAN support Intel® TXT, and is configured and WILLING to support it.
GETSEC[CAPABILITIES]—Report the SMX Capabilities

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37</td>
<td>GETSEC[CAPABILITIES] (EAX = 0)</td>
<td>Report the SMX capabilities. The capabilities index is input in EBX with the result returned in EAX.</td>
</tr>
</tbody>
</table>

Description

The GETSEC[CAPABILITIES] function returns a bit vector of supported GETSEC leaf functions. The CAPABILITIES leaf of GETSEC is selected with EAX set to 0 at entry. EBX is used as the selector for returning the bit vector field in EAX. GETSEC[CAPABILITIES] may be executed at all privilege levels, but the CR4.SMXE bit must be set or an undefined opcode exception (#UD) is returned.

With EBX = 0 upon execution of GETSEC[CAPABILITIES], EAX returns the a bit vector representing status on the presence of a Intel® TXT-capable chipset and the first 30 available GETSEC leaf functions. The format of the returned bit vector is provided in Table 7-3.

If bit 0 is set to 1, then an Intel® TXT-capable chipset has been sampled present by the processor. If bits in the range of 1-30 are set, then the corresponding GETSEC leaf function is available. If the bit value at a given bit index is 0, then the GETSEC leaf function corresponding to that index is unsupported and attempted execution results in a #UD.

Bit 31 of EAX indicates if further leaf indexes are supported. If the Extended Leafs bit 31 is set, then additional leaf functions are accessed by repeating GETSEC[CAPABILITIES] with EBX incremented by one. When the most significant bit of EAX is not set, then additional GETSEC leaf functions are not supported; indexing EBX to a higher value results in EAX returning zero.

Table 7-3. GETSEC Capability Result Encoding (EBX = 0)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chipset Present</td>
<td>0</td>
<td>Intel® TXT-capable chipset is present.</td>
</tr>
<tr>
<td>Undefined</td>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>ENTERACCS</td>
<td>2</td>
<td>GETSEC[ENTERACCS] is available.</td>
</tr>
<tr>
<td>EXITAC</td>
<td>3</td>
<td>GETSEC(EXITAC) is available.</td>
</tr>
<tr>
<td>SENTER</td>
<td>4</td>
<td>GETSEC[SENTER] is available.</td>
</tr>
<tr>
<td>SEXIT</td>
<td>5</td>
<td>GETSEC[SEXIT] is available.</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>6</td>
<td>GETSEC[PARAMETERS] is available.</td>
</tr>
<tr>
<td>SMCTRL</td>
<td>7</td>
<td>GETSEC[SMCTRL] is available.</td>
</tr>
<tr>
<td>WAKEUP</td>
<td>8</td>
<td>GETSEC[WAKEUP] is available.</td>
</tr>
<tr>
<td>Undefined</td>
<td>30:9</td>
<td>Reserved</td>
</tr>
<tr>
<td>Extended Leafs</td>
<td>31</td>
<td>Reserved for extended information reporting of GETSEC capabilities.</td>
</tr>
</tbody>
</table>
Operation
IF (CR4.SMXE=0)
   THEN #UD;
ELSIF (in VMX non-root operation)
   THEN VM Exit (reason="GETSEC instruction");
IF (EBX=0) THEN
   BitVector := 0;
   IF (TXT chipset present)
      BitVector[Chipset present] := 1;
   IF (ENTERACCS Available)
      THEN BitVector[ENTERACCS] := 1;
   IF (EXITAC Available)
      THEN BitVector[EXITAC] := 1;
   IF (SENTER Available)
      THEN BitVector[SENTER] := 1;
   IF (SEXIT Available)
      THEN BitVector[SEXIT] := 1;
   IF (PARAMETERS Available)
      THEN BitVector[PARAMETERS] := 1;
   IF (SMCTRL Available)
      THEN BitVector[SMCTRL] := 1;
   IF (WAKEUP Available)
      THEN BitVector[WAKEUP] := 1;
   EAX := BitVector;
ELSE
   EAX := 0;
END;

Flags Affected
None.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
NP 66/F2/F3 prefixes are not allowed.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.

Compatibility Mode Exceptions
#UD If CR4.SMXE = 0.
64-Bit Mode Exceptions

#UD If CR4.SMXE = 0.

VM-exit Condition

Reason (GETSEC) If in VMX non-root operation.
GETSEC[ENTERACCS]—Execute Authenticated Chipset Code

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37 (EAX = 2)</td>
<td>GETSEC[ENTERACCS]</td>
<td>Enter authenticated code execution mode. EBX holds the authenticated code module physical base address. ECX holds the authenticated code module size (bytes).</td>
</tr>
</tbody>
</table>

Description

The GETSEC[ENTERACCS] function loads, authenticates, and executes an authenticated code module using an Intel® TXT platform chipset's public key. The ENTERACCS leaf of GETSEC is selected with EAX set to 2 at entry. There are certain restrictions enforced by the processor for the execution of the GETSEC[ENTERACCS] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled, that is, CR0.CD and CR0.NW bits must be 0.
- For processor packages containing more than one logical processor, CR0.CD is checked to ensure consistency between enabled logical processors.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor cannot already be in authenticated code execution mode as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction without a subsequent exiting using GETSEC[EXITAC].
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_APIC_BASE.BSP (Bit 8).

Failure to conform to the above conditions results in the processor signaling a general protection exception.

Prior to execution of the ENTERACCS leaf, other logical processors, i.e., RLPs, in the platform must be:

- Idle in a wait-for-SIPI state (as initiated by an INIT assertion or through reset for non-BSP designated processors), or
- In the SENTER sleep state as initiated by a GETSEC[SENTER] from the initiating logical processor (ILP).

If other logical processor(s) in the same package are not idle in one of these states, execution of ENTERACCS signals a general protection exception. The same requirement and action applies if the other logical processor(s) of the same package do not have CR0.CD = 0.

A successful execution of ENTERACCS results in the ILP entering an authenticated code execution mode. Prior to reaching this point, the processor performs several checks. These include:

- Establish and check the location and size of the specified authenticated code module to be executed by the processor.
- Inhibit the ILP’s response to the external events: INIT, A20M, NMI, and SMI.
- Broadcast a message to enable protection of memory and I/O from other processor agents.
- Load the designated code module into an authenticated code execution area.
- Isolate the contents of the authenticated code execution area from further state modification by external agents.
- Authenticate the authenticated code module.
- Initialize the initiating logical processor state based on information contained in the authenticated code module header.
- Unlock the Intel® TXT-capable chipset private configuration space and TPM locality 3 space.
• Begin execution in the authenticated code module at the defined entry point.

The GETSEC[ENTERACCS] function requires two additional input parameters in the general purpose registers EBX and ECX. EBX holds the authenticated code (AC) module physical base address (the AC module must reside below 4 GBytes in physical address space) and ECX holds the AC module size (in bytes). The physical base address and size are used to retrieve the code module from system memory and load it into the internal authenticated code execution area. The base physical address is checked to verify it is on a modulo-4096 byte boundary. The size is verified to be a multiple of 64, that it does not exceed the internal authenticated code execution area capacity (as reported by GETSEC[CAPABILITIES]), and that the top address of the AC module does not exceed 32 bits. An error condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception.

As an integrity check for proper processor hardware operation, execution of GETSEC[ENTERACCS] will also check the contents of all the machine check status registers (as reported by the MSRs IA32_MCI_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must not be asserted, indicating that no machine check exception processing is currently in progress. These checks are performed prior to initiating the load of the authenticated code module. Any outstanding valid uncorrectable machine check error condition present in these status registers at this point will result in the processor signaling a general protection violation.

The ILP masks the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. This masking remains active until optionally unmasked by GETSEC[EXITAC] (this defined unmasking behavior assumes GETSEC[ENTERACCS] was not executed by a prior GETSEC[SENTER]). The purpose of this masking control is to prevent exposure to existing external event handlers that may not be under the control of the authenticated code module.

The ILP sets an internal flag to indicate it has entered authenticated code execution mode. The state of the A20M pin is likewise masked and forced internally to a de-asserted state so that any external assertion is not recognized during authenticated code execution mode.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) access and I/O originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Prior to launching the authenticated execution module using GETSEC[ENTERACCS] or GETSEC[SENTER], the processor’s MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (writeback). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. If the processor detected this requirement is not met, it will signal an Intel® TXT reset condition with an error code during the loading of the authenticated code module.

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default).

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

At the successful completion of GETSEC[ENTERACCS], the architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDTRBasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDTRLimit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.
A summary of the register state initialization after successful completion of GETSEC[ENTERACCS] is given for the processor in Table 7-4. The paging is disabled upon entry into authenticated code execution mode. The authenticated code module is loaded and initially executed using physical addresses. It is up to the system software after execution of GETSEC[ENTERACCS] to establish a new (or restore its previous) paging environment with an appropriate mapping to meet new protection requirements. EBP is initialized to the authenticated code module base physical address for initial execution in the authenticated environment. As a result, the authenticated code can reference EBP for relative address based references, given that the authenticated code module must be position independent.

### Table 7-4. Register State Initialization After GETSEC[ENTERACCS]

<table>
<thead>
<tr>
<th>Register State</th>
<th>Initialization Status</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR0</td>
<td>PG←0, AM←0, WP←0: Others unchanged</td>
<td>Paging, Alignment Check, Write-protection are disabled.</td>
</tr>
<tr>
<td>CR4</td>
<td>MCE←0, CET←0, PCIDE←0: Others unchanged</td>
<td>Machine Check Exceptions, Control-flow Enforcement Technology, and Process-context Identifiers disabled.</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>000000002H</td>
<td>IA-32e mode disabled.</td>
</tr>
<tr>
<td>IA32_EFER</td>
<td>0H</td>
<td>IA-32e mode disabled.</td>
</tr>
<tr>
<td>EIP</td>
<td>AC.base + EntryPoint</td>
<td>AC.base is in EBX as input to GETSEC[ENTERACCS].</td>
</tr>
<tr>
<td>ECX</td>
<td>Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel</td>
<td>Carry forward processor state across GETSEC[ENTERACCS].</td>
</tr>
<tr>
<td>[E</td>
<td>R]DX</td>
<td>Pre-ENTERACCS state: GDTR base</td>
</tr>
<tr>
<td>EBP</td>
<td>AC.base</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Sel=[SegSel], base=0, limit=FFFFFFh, G=1, D=1, AR=9BH</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>Sel=[SegSel] +8, base=0, limit=FFFFFFh, G=1, D=1, AR=93H</td>
<td></td>
</tr>
<tr>
<td>GDTR</td>
<td>Base= AC.base (EBX) + [GDTBasePtr], Limit= [GDTLimit]</td>
<td></td>
</tr>
<tr>
<td>DR7</td>
<td>000000400H</td>
<td></td>
</tr>
<tr>
<td>IA32_DEBUGCTL</td>
<td>0H</td>
<td>The number of initialized fields may change due to processor implementation.</td>
</tr>
<tr>
<td>IA32_MISC_ENABLE</td>
<td>See Table 7-5 for example.</td>
<td></td>
</tr>
</tbody>
</table>

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.
The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGS, and the MSR IA32_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

Performance related counters and counter control registers are cleared as part of execution of ENTERACCS. This implies any active performance counters at any time of ENTERACCS execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

The IA32_MISC_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (see the footnote for Table 7-5). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general-purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[EXITAC].

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended

---

**Table 7-5. IA32_MISC_ENABLE MSR Initialization**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast strings enable</td>
<td>0</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>FOPCODE compatibility mode enable</td>
<td>2</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Thermal monitor enable</td>
<td>3</td>
<td>Set to 1 if other thermal monitor capability is not enabled.²</td>
</tr>
<tr>
<td>Split-lock disable</td>
<td>4</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Bus lock on cache line splits disable</td>
<td>8</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Hardware prefetch disable</td>
<td>9</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>GV1/2 legacy enable</td>
<td>15</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>MONITOR/MWAIT s/m enable</td>
<td>18</td>
<td>Clear to 0.</td>
</tr>
<tr>
<td>Adjacent sector prefetch disable</td>
<td>19</td>
<td>Clear to 0.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The number of IA32_MISC_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended
that a new IDT context be established before this point. Until such a new IDT context is established, the
programmer must take care in not executing an INT n instruction or any other operation that would result in an
exception or trap signaling.

Prior to completion of the GETSEC[ENTERACCS] instruction and after successful authentication of the AC module,
the private configuration space of the Intel TXT chipset is unlocked. The authenticated code module alone can gain
access to this normally restricted chipset state for the purpose of securing the platform.

Once the authenticated code module is launched at the completion of GETSEC[ENTERACCS], it is free to enable
interrupts by setting EFLAGS.IF and enable NMI by execution of IRET. This presumes that it has re-established
interrupt handling support through initialization of the IDT, GDT, and corresponding interrupt handling code.

**Operation in a Uni-Processor Platform**

(* The state of the internal flag ACMODEFLAG persists across instruction boundary *)

\[
\text{IF (CR4.SMXE=0)} \\
\quad \text{THEN #UD;} \\
\text{ELSIF (in VMX non-root operation)} \\
\quad \text{THEN VM Exit (reason="GETSEC instruction");} \\
\text{ELSIF (GETSEC leaf unsupported)} \\
\quad \text{THEN #UD;} \\
\text{ELSIF ((in VMX operation) or} \\
\quad (\text{CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or} \\
\quad (\text{CPL>0) or (EFLAGS.VM=1) or} \\
\quad (\text{IA32_APIC_BASE.BSP=0) or} \\
\quad (\text{TXT chipset not present) or} \\
\quad (\text{ACMODEFLAG=1) or (IN_SMM=1))} \\
\quad \text{THEN #GP(0);} \\
\text{FI;} \\
\text{IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)} \\
\quad \text{THEN #GP(0);} \\
\text{ACBASE := EBX;} \\
\text{ACSIZE := ECX;} \\
\text{IF (((ACBASE MOD 4096) \neq 0) or ((ACSIZE MOD 64 ) \neq 0 ) or (ACSIZE < minimum module size) OR (ACSIZE > authenticated RAM \ capacity)) OR ((ACBASE+ACSIZE) > (2^32 -1))} \\
\quad \text{THEN #GP(0);} \\
\text{IF (secondary thread(s) CR0.CD = 1) or (secondary thread(s) NOT(wait-for-SIPI)) and} \\
\quad (\text{secondary thread(s) not in SENTER sleep state}) \\
\quad \text{THEN #GP(0);} \\
\text{Mask SMI, INIT, A20M, and NMI external pin events;} \\
\text{IA32_MISC_ENABLE := (IA32_MISC_ENABLE & MASK_CONST*)} \\
\text{(* The hexadecimal value of MASK CONST may vary due to processor implementations *)} \\
\text{A20M := 0;} \\
\text{IA32_DEBUGCTL := 0;} \\
\text{Invalidate processor TLB(s);} \\
\text{Drain Outgoing Transactions;} \\
\text{ACMODEFLAG := 1;} \\
\text{SignalTXTMessage(ProcessorHold);} \\
\text{Load the internal ACRAM based on the AC module size;} \\
\text{(* Ensure that all ACRAM loads hit Write Back memory space *)} \\
\text{IF (ACRAM memory type \neq WB)}
THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
(* Authenticate the AC Module and shutdown with an error if it fails *)
KEY := GETKEY(ACRAM, ACBASE);
KEYHASH := HASH(KEY);
CSKEYHASH := READ(TXT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE := DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific *)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] := SIGNATURE[I];
COMPUTEDSIGNATURE := HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] := COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL := ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF (ACRecEntryPoint > ACSIZE) OR (ACRecEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF (ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN ACEntryPoint := ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint := ACBASE+ACRAM[EntryPoint];
    IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
        THEN TXT-SHUTDOWN(#BadACMFormat);
    IF (ACRAM[GDTLimit] & FFFF0000h)
        THEN TXT-SHUTDOWN(#BadACMFormat);
    IF (ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8)
        THEN TXT-SHUTDOWN(#BadACMFormat);
    IF (ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL ≠ 0)
        THEN TXT-SHUTDOWN(#BadACMFormat);
    CR0.[PG.AM.WP] := 0;
    CR4.MCE := 0;
    EFLAGS := 00000002h;
    IA32_EFER := 0h;
    [E|R]BX := [E|R]IP of the instruction after GETSEC[ENTERACCS];
    ECX := Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
    [E|R]DX := Pre-GETSEC[ENTERACCS] GDT.base;
    EBP := ACBASE;
    GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
    GDTR.LIMIT := ACRAM[GDTLimit];
    CS.SEL := ACRAM[SegSel];
    CS.BASE := 0;
    CS.LIMIT := FFFFFh;
    CS.G := 1;
    CS.D := 1;
    CS.AR := 9Bh;
    DS.SEL := ACRAM[SegSel]+8;

GETSEC[ENTERACCS]–Execute Authenticated Chipset Code
DS.BASE := 0;
DS.LIMIT := FFFFFh;
DS.G := 1;
DS.D := 1;
DS.AR := 93h;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP := ACEntryPoint;
END;

**Flags Affected**
All flags are cleared.

**Use of Prefixes**
- **LOCK** Causes #UD.
- **REP** Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
- **Operand size** Causes #UD.
- **NP** 66/F2/F3 prefixes are not allowed.
- **Segment overrides** Ignored.
- **Address size** Ignored.
- **REX** Ignored.

**Protected Mode Exceptions**

**#UD**
If CR4.SMXE = 0.
If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].

**#GP(0)**
If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
If a Intel® TXT-capable chipset is not present.
If in VMX root operation.
If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.
If the processor is already in authenticated code execution mode.
If the processor is in SMM.
If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.
If the authenticated code base is not on a 4096 byte boundary.
If the authenticated code size > processor internal authenticated code area capacity.
If the authenticated code size is not modulo 64.
If other enabled logical processor(s) of the same package CR0.CD = 1.
If other enabled logical processor(s) of the same package are not in the wait-for-SIPI or SENTER sleep state.

**Real-Address Mode Exceptions**

**#UD**
If CR4.SMXE = 0.
If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].

**#GP(0)**
GETSEC[ENTERACCS] is not recognized in real-address mode.
Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) GETSEC[ENTERACCS] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.

#GP If AC code module does not reside in physical address below 2^32 -1.

64-Bit Mode Exceptions
All protected mode exceptions apply.

#GP If AC code module does not reside in physical address below 2^32 -1.

VM-exit Condition
Reason (GETSEC) If in VMX non-root operation.
GETSEC(EXITAC)—Exit Authenticated Code Execution Mode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37 (EAX=3)</td>
<td>GETSEC(EXITAC)</td>
<td>Exit authenticated code execution mode. RBX holds the Near Absolute Indirect jump target and EDX hold the exit parameter flags.</td>
</tr>
</tbody>
</table>

**Description**

The GETSEC(EXITAC) leaf function exits the ILP out of authenticated code execution mode established by GETSEC(EXITACCS) or GETSEC(SENTER). The EXITAC leaf of GETSEC is selected with EAX set to 3 at entry. EBX (or RBX, if in 64-bit mode) holds the near jump target offset for where the processor execution resumes upon exiting authenticated code execution mode. EDX contains additional parameter control information. Currently only an input value of 0 in EDX is supported. All other EDX settings are considered reserved and result in a general protection violation.

GETSEC(EXITAC) can only be executed if the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0. The processor must also be in authenticated code execution mode. To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it is in SMM or in VMX operation. A violation of these conditions results in a general protection violation.

Upon completion of the GETSEC(EXITAC) operation, the processor unmask responses to external event signals INIT#, NMI#, and SMI#. This unmasking is performed conditionally, based on whether the authenticated code execution mode was entered via execution of GETSEC(SENTER) or GETSEC(EXITACCS). If the processor is in authenticated code execution mode due to the execution of GETSEC(SENTER), then these external event signals will remain masked. In this case, A20M is kept disabled in the measured environment until the measured environment executes GETSEC(SEXIT). INIT# is unconditionally unmasked by EXITAC. Note that any events that are pending, but have been blocked while in authenticated code execution mode, will be recognized at the completion of the GETSEC(EXITAC) instruction if the pin event is unmasked.

The intent of providing the ability to optionally leave the pin events SMI#, and NMI# masked is to support the completion of a measured environment bring-up that makes use of VMX. In this envisioned security usage scenario, these events will remain masked until an appropriate virtual machine has been established in order to field servicing of these events in a safer manner. Details on when and how events are masked and unmasked in VMX operation are described in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C. It should be cautioned that if no VMX environment is to be activated following GETSEC(EXITAC), that these events will remain masked until the measured environment is exited with GETSEC(SEXIT). If this is not desired then the GETSEC function SMCTRL(0) can be used for unmasking SMI# in this context. NMI# can be correspondingly unmasked by execution of IRET.

A successful exit of the authenticated code execution mode requires the ILP to perform additional steps as outlined below:

- Invalidate the contents of the internal authenticated code execution area.
- Invalidate processor TLBs.
- Clear the internal processor AC Mode indicator flag.
- Re-lock the TPM locality 3 space.
- Unlock the Intel® TXT-capable chipset memory and I/O protections to allow memory and I/O activity by other processor agents.
- Perform a near absolute indirect jump to the designated instruction location.

The content of the authenticated code execution area is invalidated by hardware in order to protect it from further use or visibility. This internal processor storage area can no longer be used or relied upon after GETSEC(EXITAC). Data structures need to be re-established outside of the authenticated code execution area if they are to be referenced after EXITAC. Since addressed memory content formerly mapped to the authenticated code execution area may no longer be coherent with external system memory after EXITAC, processor TLBs in support of linear to physical address translation are also invalidated.
Upon completion of GETSEC[EXITAC] a near absolute indirect transfer is performed with EIP loaded with the contents of EBX (based on the current operating mode size). In 64-bit mode, all 64 bits of RBX are loaded into RIP if REX.W precedes GETSEC[EXITAC]. Otherwise RBX is treated as 32 bits even while in 64-bit mode. Conventional CS limit checking is performed as part of this control transfer. Any exception conditions generated as part of this control transfer will be directed to the existing IDT; thus it is recommended that an IDTR should also be established prior to execution of the EXITAC function if there is a need for fault handling. In addition, any segmentation related (and paging) data structures to be used after EXITAC should be re-established or validated by the authenticated code prior to EXITAC.

In addition, any segmentation related (and paging) data structures to be used after EXITAC need to be re-established and mapped outside of the authenticated RAM designated area by the authenticated code prior to EXITAC. Any data structure held within the authenticated RAM allocated area will no longer be accessible after completion by EXITAC.

**Operation**

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

IF (CR4.SMXE=0)
   THEN #UD;
ELSIF (in VMX non-root operation)
   THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
   THEN #UD;
ELSIF (in VMX operation) or ( (in 64-bit mode) and ( RBX is non-canonical )
   (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
   (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX ≠ 0))
   THEN #GP(0);
IF (OperandSize = 32)
   THEN tempEIP := EBX;
ELSIF (OperandSize = 64)
   THEN tempEIP := RBX;
ELSE
   tempEIP := EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
   THEN #GP(0);
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENTERFLAG=0)
   THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
   THEN Unmask SMI pin event;
ACMODEFLAG := 0;  
If IA32_EFER.LMA == 1
   THEN CR3 := R8;
EIP := tempEIP;
END;

**Flags Affected**

None.
Use of Prefixes

LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
NP 66/F2/F3 prefixes are not allowed.
Segment overrides Ignored.
Address size Ignored.
REX.W Sets 64-bit mode Operand size attribute.

Protected Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC(EXITAC) is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.PE = 0 or CPL>0 or EFLAGS.VM =1.
If in VMX root operation.
If the processor is not currently in authenticated code execution mode.
If the processor is in SMM.
If any reserved bit position is set in the EDX parameter register.

Real-Address Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC(EXITAC) is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC(EXITAC) is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC(EXITAC) is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC(EXITAC) is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.
#GP(0) If the target address in RBX is not in a canonical form.

VM-Exit Condition

Reason (GETSEC) If in VMX non-root operation.
GETSEC[SENTER]—Enter a Measured Environment

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37</td>
<td>GETSEC[SENTER]</td>
<td>Launch a measured environment.</td>
</tr>
<tr>
<td>(EAX=4)</td>
<td></td>
<td>EBX holds the SINIT authenticated code module physical base address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ECX holds the SINIT authenticated code module size (bytes).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EDX controls the level of functionality supported by the measured environment launch.</td>
</tr>
</tbody>
</table>

Description

The GETSEC[SENTER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The SENTER leaf of GETSEC is selected with EAX set to 4 at execution. The physical base address of the AC module to be loaded and authenticated is specified in EBX. The size of the module in bytes is specified in ECX. EDX controls the level of functionality supported by the measured environment launch. To enable the full functionality of the protected environment launch, EDX must be initialized to zero.

The authenticated code base address and size parameters (in bytes) are passed to the GETSEC[SENTER] instruction using EBX and ECX respectively. The ILP evaluates the contents of these registers according to the rules for the AC module address in GETSEC[ENTERACCS]. AC module execution follows the same rules, as set by GETSEC[ENTERACCS].

The launching software must ensure that the TPM.ACCESS_0.activeLocality bit is clear before executing the GETSEC[SENTER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[SENTER] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled using the CR0.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not be in authenticated code execution mode or already in a measured environment (as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[SENTER] instruction must also be designated the BSP (boot-strap processor) as defined by IA32_APIC_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PARAMETERS] leaf reports otherwise, only a value of zero is supported.

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction leaf starts the launch of a measured environment by initiating a rendezvous sequence for all logical processors in the platform. The rendezvous sequence involves the initiating logical processor sending a message (by executing GETSEC[SENTER]) and other responding logical processors (RLPs) acknowledging the message, thus synchronizing the RLP(s) with the ILP.

In response to a message signaling the completion of rendezvous, RLPs clear the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and enter an SENTER sleep state. In this sleep state, RLPs enter an idle processor condition while waiting to be activated after a measured environment has been established by the system executive. RLPs in the SENTER sleep state can only be activated by the GETSEC leaf function WAKEUP in a measured environment.
A successful launch of the measured environment results in the initiating logical processor entering the authentic-
cated code execution mode. Prior to reaching this point, the ILP performs the following steps internally:

- Inhibit processor response to the external events: INIT, A20M, NMI, and SMI.
- Establish and check the location and size of the authenticated code module to be executed by the ILP.
- Check for the existence of an Intel® TXT-capable chipset.
- Verify the current power management configuration is acceptable.
- Broadcast a message to enable protection of memory and I/O from activities from other processor agents.
- Load the designated AC module into authenticated code execution area.
- Isolate the content of authenticated code execution area from further state modification by external agents.
- Authenticate the AC module.
- Updated the Trusted Platform Module (TPM) with the authenticated code module’s hash.
- Initialize processor state based on the authenticated code module header information.
- Unlock the Intel® TXT-capable chipset private configuration register space and TPM locality 3 space.
- Begin execution in the authenticated code module at the defined entry point.

As an integrity check for proper processor hardware operation, execution of GETSEC[SENTER] will also check the
contents of all the machine check status registers (as reported by the MSRs IA32_MCI_STATUS) for any valid
uncorrectable error condition. In addition, the global machine check status register IA32_MCG_STATUS MCIP bit
must be cleared and the IERR processor package pin (or its equivalent) must be not asserted, indicating that no
machine check exception processing is currently in-progress. These checks are performed twice: once by the ILP
prior to the broadcast of the rendezvous message to RLPs, and later in response to RLPs acknowledging the rendez-
vous message. Any outstanding valid uncorrectable machine check error condition present in the machine check
status registers at the first check point will result in the ILP signaling a general protection violation. If an
outstanding valid uncorrectable machine check error condition is present at the second check point, then this will
result in the corresponding logical processor signaling the more severe TXT-shutdown condition with an error code
of 12.

Before loading and authentication of the target code module is performed, the processor also checks that the
current voltage and bus ratio encodings correspond to known good values supportable by the processor. The MSR
IA32_PERF_STATUS values are compared against either the processor supported maximum operating target
setting, system reset setting, or the thermal monitor operating target. If the current settings do not meet any of
these criteria then the SENTER function will attempt to change the voltage and bus ratio select controls in a
processor-specific manner. This adjustment may be to the thermal monitor, minimum (if different), or maximum
operating target depending on the processor.

This implies that some thermal operating target parameters configured by BIOS may be overridden by SENTER.
The measured environment software may need to take responsibility for restoring such settings that are deemed
to be safe, but not necessarily recognized by SENTER. If an adjustment is not possible when an out of range setting
is discovered, then the processor will abort the measured launch. This may be the case for chipset controlled
settings of these values or if the controllability is not enabled on the processor. In this case it is the responsibility of
the external software to program the chipset voltage ID and/or bus ratio select settings to known good values
recognized by the processor, prior to executing SENTER.

NOTE
For a mobile processor, an adjustment can be made according to the thermal monitor operating
target. For a quad-core processor the SENTER adjustment mechanism may result in a more conser-
vative but non-uniform voltage setting, depending on the pre-SENTER settings per core.

The ILP and RLPs mask the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#.
The purpose of this masking control is to prevent exposure to existing external event handlers until a protected handler
has been put in place to directly handle these events. Masked external pin events may be unmasked conditionally
or unconditionally via the GETSEC[EXITAC], GETSEC[EXIT], GETSEC[SMCTRL] or for specific VMX related opera-
tions such as a VM entry or the VMXOFF instruction (see respective GETSEC leaves and Intel® 64 and IA-32 Archi-
tectures Software Developer’s Manual, Volume 3C, for more details). The state of the A20M pin is masked and
forced internally to a de-asserted state so that external assertion is not recognized. A20M masking as set by
GETSEC[SENTER] is undone only after taking down the measured environment with the GETSEC[SEXIT] instruction or processor reset. INTR is masked by simply clearing the EFLAGS.IF bit. It is the responsibility of system software to control the processor response to INTR through appropriate management of EFLAGS.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) and I/O activities originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Once the authenticated code module has been loaded into the authenticated code execution area, it is protected against further modification from external bus snoops. There is also a requirement that the memory type for the authenticated code module address range be WB (via initialization of the MTRRs prior to execution of this instruction). If this condition is not satisfied, it is a violation of security and the processor will force a TXT system reset (after writing an error code to the chipset LT.ERRORCODE register). This action is referred to as a Intel® TXT reset condition. It is performed when it is considered unreliable to signal an error through the conventional exception reporting mechanism.

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

Once successful authentication has been completed by the ILP, the computed hash is stored in a trusted storage facility in the platform. The following trusted storage facility are supported:

- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 0, the computed hash is stored to the platform’s TPM at PCR17 after this register is implicitly reset. PCR17 is a dedicated register for holding the computed hash of the authenticated code module loaded and subsequently executed by the GETSEC[SENTER]. As part of this process, the dynamic PCRs 18-22 are reset so they can be utilized by subsequently software for registration of code and data modules.
- If the platform register FTM_INTERFACE_ID.[bits 3:0] = 1, the computed hash is stored in a firmware trusted module (FTM) using a modified protocol similar to the protocol used to write to TPM’s PCR17.

After successful execution of SENTER, either PCR17 (if FTM is not enabled) or the FTM (if enabled) contains the measurement of AC code and the SENTER launching parameters.

After authentication is completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked so that the authenticated code module and measured environment software can gain access to this normally restricted chipset state. The Intel® TXT-capable chipset private configuration space can be locked later by software writing to the chipset LT.CMD.CLOSE-PRIVATE register or unconditionally using the GETSEC[SEXIT] instruction.

The SENTER leaf function also initializes some processor architecture state for the ILP from contents held in the header of the authenticated code module. Since the authenticated code module is relocatable, all address references are relative to the base address passed in via EBX. The ILP GDTR base value is initialized to EBX + [GDTBasePtr] and GDTR limit set to [GDTLimit]. The CS selector is initialized to the value held in the AC module header field SegSel, while the DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write/accessible for DS, SS, and ES, while execute/read/accessed for CS. Execution in the authenticated code module for the ILP begins with the EIP set to EBX + [EntryPoint]. AC module defined fields used for initializing processor state are consistency checked with a failure resulting in an TXT-shutdown condition.

Table 7-6 provides a summary of processor state initialization for the ILP and RLP(s) after successful completion of GETSEC[SENTER]. For both ILP and RLP(s), paging is disabled upon entry to the measured environment. It is up to the ILP to establish a trusted paging environment, with appropriate mappings, to meet protection requirements established during the launch of the measured environment. RLP state initialization is not completed until a subsequent wake-up has been signaled by execution of the GETSEC[WAKEUP] function by the ILP.
Table 7-6. Register State Initialization After GETSEC[SENTER] and GETSEC[WAKEUP]

<table>
<thead>
<tr>
<th>Register State</th>
<th>ILP after GETSEC[SENTER]</th>
<th>RLP after GETSEC[WAKEUP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR0</td>
<td>PG ← 0, AM ← 0, WP ← 0; Others unchanged</td>
<td>PG ← 0, CD ← 0, NW ← 0, AM ← 0, WP ← 0; PE ← 1, NE ← 1</td>
</tr>
<tr>
<td>CR4</td>
<td>00004000H</td>
<td>00004000H</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>00000002H</td>
<td>00000002H</td>
</tr>
<tr>
<td>IA32_EFER</td>
<td>0H</td>
<td>0</td>
</tr>
<tr>
<td>EIP</td>
<td>[EntryPoint from MLE header^1]</td>
<td>[LT.MLE.JOIN + 12]</td>
</tr>
<tr>
<td>EBX</td>
<td>Unchanged [SINIT.BASE]</td>
<td>Unchanged</td>
</tr>
<tr>
<td>EDX</td>
<td>SENTER control flags</td>
<td>Unchanged</td>
</tr>
<tr>
<td>EBP</td>
<td>SINIT.BASE</td>
<td>Unchanged</td>
</tr>
<tr>
<td>CS</td>
<td>Sel = [SINIT SegSel], base = 0, limit = FFFFFFFH, G = 1, D = 1, AR = 9BH</td>
<td>Sel = [LT.MLE.JOIN + 8], base = 0, limit = FFFFFFFH, G = 1, D = 1, AR = 9BH</td>
</tr>
<tr>
<td>DS, ES, SS</td>
<td>Sel = [SINIT SegSel] + 8, base = 0, limit = FFFFFFFH, G = 1, D = 1, AR = 93H</td>
<td>Sel = [LT.MLE.JOIN + 8] + 8, base = 0, limit = FFFFFFFH, G = 1, D = 1, AR = 93H</td>
</tr>
<tr>
<td>GDTR</td>
<td>Base = SINIT.base (EBX) + [SINIT.GDTBasePtr], Limit = [SINIT.GDTLimit]</td>
<td>Base = [LT.MLE.JOIN + 4], Limit = [LT.MLE.JOIN]</td>
</tr>
<tr>
<td>DR7</td>
<td>00000400H</td>
<td>00000400H</td>
</tr>
<tr>
<td>IA32_DEBUGCTL</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>Performance counters and counter control registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MISC_ENABLE</td>
<td>See Table 7-5</td>
<td>See Table 7-5</td>
</tr>
<tr>
<td>IA32_SMM_MONITOR_CTL</td>
<td>Bit 2 ← 0</td>
<td>Bit 2 ← 0</td>
</tr>
</tbody>
</table>

NOTES:

Segmentation related processor state that has not been initialized by GETSEC[SENTER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENTER]. This is achieved by clearing DR7, TF in EFLAGS, and the MSR IA32_DEBUGCTL as defined in Table 7-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor CR4.MCE
must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32_MISC_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 7-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

**Effect of MSR IA32_FEATURE_CONTROL MSR**

Bits 15:8 of the IA32_FEATURE_CONTROL MSR affect the execution of GETSEC[SENTER]. These bits consist of two fields:

- Bit 15: a global enable control for execution of SENTER.
- Bits 14:8: a parameter control field providing the ability to qualify SENTER execution based on the level of functionality specified with corresponding EDX parameter bits 6:0.

The layout of these fields in the IA32_FEATURE_CONTROL MSR is shown in Table 7-1.

Prior to the execution of GETSEC[SENTER], the lock bit of IA32_FEATURE_CONTROL MSR must be bit set to affirm the settings to be used. Once the lock bit is set, only a power-up reset condition will clear this MSR. The IA32_FEATURE_CONTROL MSR must be configured in accordance to the intended usage at platform initialization. Note that this MSR is only available on SMX or VMX enabled processors. Otherwise, IA32_FEATURE_CONTROL is treated as reserved.

The Intel® Trusted Execution Technology Measured Launched Environment Programming Guide provides additional details and requirements for programming measured environment software to launch in an Intel TXT platform.

**Operation in a Uni-Processor Platform**

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

**GETSEC[SENTER] (ILP Only):**

- **IF (CR4.SMXE=0)**
  - THEN #UD;
- **ELSE IF (in VMX non-root operation)**
  - THEN VM Exit (reason="GETSEC instruction");
- **ELSE IF (GETSEC leaf unsupported)**
  - THEN #UD;
- **ELSE IF (in VMX root operation) or**
  - (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
  - (CPL>0) or (EFLAGS.VM=1) or
  - (IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
  - (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
  - (TPM interface is not present) or
  - (EDX ≠ (SENTER_EDX_support_mask & EDX)) or
  - (IA32_FEATURE_CONTROL[0]=0) or (IA32_FEATURE_CONTROL[15]=0) or
  - ((IA32_FEATURE_CONTROL[14:8] & EDX[6:0]) ≠ EDX[6:0]))
  - THEN #GP(0);
- **IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)**
  - FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    - **IF IA32_MCG[I].STATUS = uncorrectable error**
      - THEN #GP(0);
  - Fl;
- OD;

**GETSEC[SENTER]—Enter a Measured Environment**
SAFER MODE EXTENSIONS REFERENCE

FT;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN #GP(0);
ACBASE := EBX;
ACSIZE := ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0 ) or (ACSIZE < minimum module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2⁻³² - 1)))
  THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
  WHILE (no SignalSENTER message);

TXT__SENTER__MSG_EVENT (ILP & RLP):
Mask and clear SignalSENTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
  THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
  IF IA32_MC[I]_STATUS = uncorrectable error
    THEN TXT-SHUTDOWN(#UnrecovMCError);
  FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
  THEN TXT-SHUTDOWN(#UnrecovMCError);
IF (Voltage or bus ratio status are NOT at a known good state)
  THEN IF (Voltage select and bus ratio are internally adjustable)
    THEN
      Make product-specific adjustment on operating parameters;
    ELSE
      TXT-SHUTDOWN(#IllegalVIDBRatio);
    FI;
IA32_MISC_ENABLE := (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M := 0;
IA32_DEBUGCTL := 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENTERFLAG := 1;
SignalTXTMsg(SENTERAck);
IF (logical processor is not ILP)
  THEN GOTO RLP__SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
  DONE := TXT.READ(LT.STS);
  WHILE (not DONE);
  SignalTXTMsg(SENTERContinue);
  SignalTXTMsg(ProcessorHold);
  FOR I=ACBASE to ACBASE+ACSIZE-1 DO
    ACRAM[I-ACBASE].ADDR := I;
    ACRAM[I-ACBASE].DATA := LOAD(I);
  OD;

7-26  Vol. 2D  GETSEC[SENTER]—Enter a Measured Environment
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY := GETKEY(ACRAM, ACBASE);
KEYHASH := HASH(KEY);
CSKEYHASH := LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE := DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific *)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] := SIGNATURE[I];
COMPUTEDSIGNATURE := HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] := COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL := ACRAM[CodeControl];
IF ((ACMCONTROL0 = 0) and (ACMCONTROL1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[DTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    (ACRAM[DTBasePtr] + ACRAM[DTLimit]) >= ACSIZE)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL0 = 1) and (ACMCONTROL1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN ACEntrypoint := ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntrypoint := ACBASE+ACRAM[EntryPoint];
IF ((ACEntrypoint >= ACSIZE) OR (ACEntrypoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[DTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF (FTM_INTERFACE_ID.[3:0] = 1 ) (* Alternate FTM Interface has been enabled *)
    THEN (* TPM_LOC_CTRL_4 is located at 0FED400BH, TMP_DATA_BUFFER_4 is located at 0FED4008OH *)
        WRITE(TPM_LOC_CTRL_4) := 01H; (* Modified HASH.START protocol *)
        (* Write to firmware storage *)
        WRITE(TPM_DATA_BUFFER_4) := SIGNATURE_LEN_CONST + 4;
        FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
            WRITE(TPM_DATA_BUFFER_4 + 2 + I) := ACRAM[SCRATCH.I];
        WRITE(TPM_DATA_BUFFER_4 + 2 + SIGNATURE_LEN_CONST) := EDX;
        WRITE(FTM_LOC_CTRL) := 06H; (* Modified protocol combining HASH.DATA and HASH.END *)
ELSE IF (FTM_INTERFACE_ID.[3:0] = 0 ) (* Use standard TPM Interface *)
    ACRAM[SCRATCH.SIGNATURE_LENCONST] := EDX;
    WRITE(TPM.HASH.START) := 0;
    FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
        WRITE(TPM.HASH.DATA) := ACRAM[SCRATCH.I];
    WRITE(TPM.HASH.END) := 0;
FI;
ACMODEFLAG := 1;
CR0.[PG.AM.WP] := 0;
CR4 := 00004000h;
EFLAGS := 00000002h;
IA32_EFER := 0;
EBP := ACBASE;
GDTR.BASE := ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT := ACRAM[GDTLimit];
CS.SEL := ACRAM[SegSel];
CS.BASE := 0;
CS.LIMIT := FFFFFh;
CS.G := 1;
CS.D := 1;
CS.AR := 9Bh;
DS.SEL := ACRAM[SegSel]+8;
DS.BASE := 0;
DS.LIMIT := FFFFFh;
DS.G := 1;
DS.D := 1;
DS.AR := 93h;
SS := DS;
ES := DS;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP := ACEntryPoint;
END;

RLP_SENTER_ROUTINE: (RLP Only)
Mask SMI, INIT, A20M, and NMI external pin events
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP := 0;
GOTO SENTER sleep state;
END;

Flags Affected
All flags are cleared.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
NP 66/F2/F3 prefixes are not allowed.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.
Protected Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
If in VMX root operation.
If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.
If an Intel® TXT-capable chipset is not present.
If an Intel® TXT-capable chipset interface to TPM is not detected as present.
If a protected partition is already active or the processor is already in authenticated code mode.
If the processor is in SMM.
If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.
If the authenticated code base is not on a 4096 byte boundary.
If the authenticated code size > processor's authenticated code execution area storage capacity.
If the authenticated code size is not modulo 64.

Real-Address Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) GETSEC[SENTER] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.
If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].

#GP(0) GETSEC[SENTER] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP If AC code module does not reside in physical address below 2^32 -1.

64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP If AC code module does not reside in physical address below 2^32 -1.

VM-Exit Condition

Reason (GETSEC) If in VMX non-root operation.
GETSEC[SEXIT]—Exit Measured Environment

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37 (EAX=5)</td>
<td>GETSEC[SEXIT]</td>
<td>Exit measured environment.</td>
</tr>
</tbody>
</table>

Description

The GETSEC[SEXIT] instruction initiates an exit of a measured environment established by GETSEC[SENTER]. The SEXIT leaf of GETSEC is selected with EAX set to 5 at execution. This instruction leaf sends a message to all logical processors in the platform to signal the measured environment exit.

There are restrictions enforced by the processor for the execution of the GETSEC[SEXIT] instruction:

- Execution is not allowed unless the processor is in protected mode (CR0.PE = 1) with CPL = 0 and EFLAGS.VM = 0.
- The processor must be in a measured environment as launched by a previous GETSEC[SENTER] instruction, but not still in authenticated code execution mode.
- To avoid potential interoperability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To ensure consistent handling of SIPI messages, the processor executing the GETSEC[SEXIT] instruction must also be designated the BSP (bootstrap processor) as defined by the register bit IA32_APIC_BASE.BSP (bit 8).

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction initiates a sequence to rendezvous the RLPs with the ILP. It then clears the internal processor flag indicating the processor is operating in a measured environment.

In response to a message signaling the completion of rendezvous, all RLPs restart execution with the instruction that was to be executed at the time GETSEC[SEXIT] was recognized. This applies to all processor conditions, with the following exceptions:

- If an RLP executed HLT and was in this halt state at the time of the message initiated by GETSEC[SEXIT], then execution resumes in the halt state.
- If an RLP was executing MWAIT, then a message initiated by GETSEC[SEXIT] causes an exit of the MWAIT state, falling through to the next instruction.
- If an RLP was executing an intermediate iteration of a string instruction, then the processor resumes execution of the string instruction at the point which the message initiated by GETSEC[SEXIT] was recognized.
- If an RLP is still in the SENTER sleep state (never awakened with GETSEC[WAKEUP]), it will be sent to the wait-for-SIPI state after first clearing the bootstrap processor indicator flag (IA32_APIC_BASE.BSP) and any pending SIPI state. In this case, such RLPs are initialized to an architectural state consistent with having taken a soft reset using the INIT# pin.

Prior to completion of the GETSEC[SEXIT] operation, both the ILP and any active RLPs unmask the response of the external event signals INIT#, A20M, NMI#, and SMI#. This unmasking is performed unconditionally to recognize pin events which are masked after a GETSEC[SENTER]. The state of A20M is unmasked, as the A20M pin is not recognized while the measured environment is active.

On a successful exit of the measured environment, the ILP re-locks the Intel® TXT-capable chipset private configuration space. GETSEC[SEXIT] does not affect the content of any PCR.

At completion of GETSEC[SEXIT] by the ILP, execution proceeds to the next instruction. Since EFLAGS and the debug register state are not modified by this instruction, a pending trap condition is free to be signaled if previously enabled.
Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

GETSEC[SEXIT] (ILP Only):
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APB_BASE.BSP=0) or
    (TXT chipset not present) or
    (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);
    SignalTXTMsg(SEXIT);
DO
WHILE (no SignalSEXIT message);

TXT_SEXIT_MSG_EVENT (ILP & RLP):
Mask and clear SignalSEXIT event;
Clear MONITOR FSM;
Unmask SignalSENTER event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
SignalTXTMsg(SEXITAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SEXIT_ROUTINE;
    (* ILP waits for all logical processors to ACK *)
DO
    DONE := READ(LT.STS);
WHILE (NOT DONE);
SignalTXTMsg(SEXITContinue);
SignalTXTMsg(ClosePrivate);
SENTERFLAG := 0;
Unmask SMI, INIT, A20M, and NMI external pin events;
END;

RLP_SEXIT_ROUTINE (RLPs Only):
Wait for SignalSEXITContinue message;
Unmask SMI, INIT, A20M, and NMI external pin events;
IF (prior execution state = HLT)
    THEN reenter HLT state;
IF (prior execution state = SENTER sleep)
    THEN
        IA32_APB_BASE.BSP := 0;
        Clear pending SIPI state;
        Call INIT_PROCESSOR_STATE;
        Unmask SIPI event;
        GOTO WAIT-FOR-SIPI;
    FI;
END;
**Flags Affected**

ILP: None.
RLPs: All flags are modified for an RLP, returning to wait-for-SIPI state, none otherwise.

**Use of Prefixes**

- **LOCK**: Causes #UD.
- **REP***: Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
- **Operand size**: Causes #UD.
- **NP**: 66/F2/F3 prefixes are not allowed.
- **Segment overrides**: Ignored.
- **Address size**: Ignored.
- **REX**: Ignored.

**Protected Mode Exceptions**

- **#UD**: If CR4.SMXE = 0.
  
  - If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
- **#GP(0)**: If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
  
  - If in VMX root operation.
  - If the initiating processor is not designated via the MSR bit IA32_APIC_BASE.BSP.
  - If an Intel® TXT-capable chipset is not present.
  - If a protected partition is not already active or the processor is already in authenticated code mode.
  - If the processor is in SMM.

**Real-Address Mode Exceptions**

- **#UD**: If CR4.SMXE = 0.
  
  - If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
- **#GP(0)**: GETSEC[SEXIT] is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

- **#UD**: If CR4.SMXE = 0.
  
  - If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].
- **#GP(0)**: GETSEC[SEXIT] is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

All protected mode exceptions apply.

**64-Bit Mode Exceptions**

All protected mode exceptions apply.

**VM-Exit Condition**

- **Reason (GETSEC)**: If in VMX non-root operation.
GETSEC[PARAMETERS]—Report the SMX Parameters

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37</td>
<td>GETSEC[PARAMETERS]</td>
<td>Report the SMX parameters. The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.</td>
</tr>
</tbody>
</table>

Description

The GETSEC[PARAMETERS] instruction returns specific parameter information for SMX features supported by the processor. Parameter information is returned in EAX, EBX, and ECX, with the input parameter selected using EBX.

Software retrieves parameter information by searching with an input index for EBX starting at 0, and then reading the returned results in EAX, EBX, and ECX. EAX[4:0] is designated to return a parameter type field indicating if a parameter is available and what type it is. If EAX[4:0] is returned with 0, this designates a null parameter and indicates no more parameters are available.

Table 7-7 defines the parameter types supported in current and future implementations.

Table 7-7. SMX Reporting Parameters Format

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Parameter Description</th>
<th>EAX[31:5]</th>
<th>EBX[31:0]</th>
<th>ECX[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>Reserved (0 returned)</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>1</td>
<td>Supported AC module versions</td>
<td>Reserved (0 returned)</td>
<td>Version comparison mask</td>
<td>Version numbers supported</td>
</tr>
<tr>
<td>2</td>
<td>Max size of authenticated code execution area</td>
<td>Multiply by 32 for size in bytes</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>3</td>
<td>External memory types supported during AC mode</td>
<td>Memory type bit mask</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>4</td>
<td>Selective SENTER functionality control</td>
<td>EAX[14:8] correspond to available SENTER function disable controls</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
<tr>
<td>5</td>
<td>TXT extensions support</td>
<td>TXT Feature Extensions Flags (see Table)</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>6-31</td>
<td>Undefined</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
<td>Reserved (unmodified)</td>
</tr>
</tbody>
</table>
Supported AC module versions (as defined by the AC module HeaderVersion field) can be determined for a particular SMX capable processor by the type 1 parameter. Using EBX to index through the available parameters reported by GETSEC[PARAMETERS] for each unique parameter set returned for type 1, software can determine the complete list of AC module version(s) supported.

For each parameter set, EBX returns the comparison mask and ECX returns the available HeaderVersion field values supported, after AND’ing the target HeaderVersion with the comparison mask. Software can then determine if a particular AC module version is supported by following the pseudo-code search routine given below:

```plaintext
parameter_search_index = 0

do {
    EBX = parameter_search_index++
    EAX = 6
    GETSEC
    if (EAX[4:0] = 1) {
        if ((version_query & EBX) = ECX) {
            version_is_supported = 1
            break
        }
    }
} while (EAX[4:0] ≠ 0)
```

If only AC modules with a HeaderVersion of 0 are supported by the processor, then only one parameter set of type 1 will be returned, as follows: EAX = 00000001H, EBX = FFFFFFFFH and ECX = 00000000H.

The maximum capacity for an authenticated code execution area supported by the processor is reported with the parameter type of 2. The maximum supported size in bytes is determined by multiplying the returned size in EAX[31:5] by 32. Thus, for a maximum supported authenticated RAM size of 32KBytes, EAX returns with 00008002H.

Supportable memory types for memory mapped outside of the authenticated code execution area are reported with the parameter type of 3. While is active, as initiated by the GETSEC functions SENTER and ENTERACCS and terminated by EXITAC, there are restrictions on what memory types are allowed for the rest of system memory. It is the responsibility of the system software to initialize the memory type range register (MTRR) MSRs and/or the page attribute table (PAT) to only map memory types consistent with the reporting of this parameter. The reporting of supportable memory types of external memory is indicated using a bit map returned in EAX[31:8]. These bit positions correspond to the memory type encodings defined for the MTRR MSR and PAT programming. See Table 7-9.

### Table 7-8. TXT Feature Extensions Flags

<table>
<thead>
<tr>
<th>Bit</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Processor based S-CRTM support</td>
<td>Returns 1 if this processor implements a processor-rooted S-CRTM capability and 0 if not (S-CRTM is rooted in BIOS). This flag cannot be used to infer whether the chipset supports TXT or whether the processor support SMX.</td>
</tr>
<tr>
<td>6</td>
<td>Machine Check Handling</td>
<td>Returns 1 if it machine check status registers can be preserved through ENTERACCS and SENTER. If this bit is 1, the caller of ENTERACCS and SENTER is not required to clear machine check error status bits before invoking these GETSEC leaves. If this bit returns 0, the caller of ENTERACCS and SENTER must clear all machine check error status bits before invoking these GETSEC leaves.</td>
</tr>
<tr>
<td>31:7</td>
<td>Reserved</td>
<td>Reserved for future use. Will return 0.</td>
</tr>
</tbody>
</table>
The parameter type of 4 is used for enumerating the availability of selective \texttt{GETSEC[SENTER]} function disable controls. If a 1 is reported in bits 14:8 of the returned parameter EAX, then this indicates a disable control capability exists with \texttt{SENTER} for a particular function. The enumerated field in bits 14:8 corresponds to use of the EDX input parameter bits 6:0 for \texttt{SENTER}. If an enumerated field bit is set to 1, then the corresponding EDX input parameter bit of EDX may be set to 1 to disable that designated function. If the enumerated field bit is 0 or this parameter is not reported, then no disable capability exists with the corresponding EDX input parameter for \texttt{SENTER}, and EDX bit(s) must be cleared to 0 to enable execution of \texttt{SENTER}. If no selective disable capability for \texttt{SENTER} exists as enumerated, then the corresponding bits in the IA32\_FEATURE\_CONTROL MSR bits 14:8 must also be programmed to 1 if the \texttt{SENTER} global enable bit 15 of the MSR is set. This is required to enable future extensibility of \texttt{SENTER} selective disable capability with respect to potentially separate software initialization of the MSR.

### Table 7-9. External Memory Types Using Parameter 3

<table>
<thead>
<tr>
<th>EAX Bit Position</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Uncacheable (UC)</td>
</tr>
<tr>
<td>9</td>
<td>Write Combining (WC)</td>
</tr>
<tr>
<td>11:10</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Write-through (WT)</td>
</tr>
<tr>
<td>13</td>
<td>Write-protected (WP)</td>
</tr>
<tr>
<td>14</td>
<td>Write-back (WB)</td>
</tr>
<tr>
<td>31:15</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

If the \texttt{GETSEC[PARAMETERS]} leaf or specific parameter is not present for a given SMX capable processor, then default parameter values should be assumed. These are defined in Table 7-10.

### Table 7-10. Default Parameter Values

<table>
<thead>
<tr>
<th>Parameter Type EAX[4:0]</th>
<th>Default Setting</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0 only</td>
<td>Supported AC module versions.</td>
</tr>
<tr>
<td>2</td>
<td>32 KBytes</td>
<td>Authenticated code execution area size.</td>
</tr>
<tr>
<td>3</td>
<td>UC only</td>
<td>External memory types supported during AC execution mode.</td>
</tr>
<tr>
<td>4</td>
<td>None</td>
<td>Available SENTER selective disable controls.</td>
</tr>
</tbody>
</table>

### Operation

(* example of a processor supporting only a 0.0 HeaderVersion, 32K ACRAM size, memory types UC and WC *)

\[\begin{align*}
\text{IF (CR4.SMXE=0)} & \quad \text{THEN} \ #UD; \\
\text{ELSE IF (in VMX non-root operation)} & \quad \text{THEN VM Exit (reason="GETSEC instruction");} \\
\text{ELSE IF (GETSEC leaf unsupported)} & \quad \text{THEN} \ #UD; \\
\text{(* example of a processor supporting a 0.0 HeaderVersion *)} & \quad \text{IF (EBX=0) THEN} \\
& \quad \text{EAX := 00000001h;} \\
& \quad \text{EBX := FFFFFFFFh;}
\end{align*}\]
ECX := 00000000h;
ELSE IF (EBX=1)
  (* example of a processor supporting a 32K ACRAM size *)
  THEN EAX := 00000002h;
ELSE IF (EBX=2)
  (* example of a processor supporting external memory types of UC and WC *)
  THEN EAX := 00000303h;
ELSE IF (EBX= other value(s) less than unsupported index value)
  (* EAX value varies. Consult Table 7-7 and Table *)
ELSE (* unsupported index*)
  EAX := 00000000h;
END;

Flags Affected
None.

Use of Prefixes
LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
NP 66/F2/F3 prefixes are not allowed.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

Real-Address Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

Virtual-8086 Mode Exceptions
#UD If CR4.SMXE = 0.
If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].

Compatibility Mode Exceptions
All protected mode exceptions apply.

64-Bit Mode Exceptions
All protected mode exceptions apply.

VM-Exit Condition
Reason (GETSEC) If in VMX non-root operation.
**GETSEC[SMCTRL]—SMX Mode Control**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 37 (EAX = 7)</td>
<td>GETSEC[SMCTRL]</td>
<td>Perform specified SMX mode control as selected with the input EBX.</td>
</tr>
</tbody>
</table>

**Description**

The GETSEC[SMCTRL] instruction is available for performing certain SMX specific mode control operations. The operation to be performed is selected through the input register EBX. Currently only an input value in EBX of 0 is supported. All other EBX settings will result in the signaling of a general protection violation.

If EBX is set to 0, then the SMCTRL leaf is used to re-enable SMI events. SMI is masked by the ILP executing the GETSEC[SENTER] instruction (SMI is also masked in the responding logical processors in response to SENTER rendezvous messages.). The determination of when this instruction is allowed and the events that are unmasked is dependent on the processor context (See Table 7-11). For brevity, the usage of SMCTRL where EBX=0 will be referred to as GETSEC[SMCTRL(0)].

As part of support for launching a measured environment, the SMI, NMI, and INIT events are masked after GETSEC[SENTER], and remain masked after exiting authenticated execution mode. Unmasking these events should be accompanied by securely enabling these event handlers. These security concerns can be addressed in VMX operation by a MVMM.

The VM monitor can choose two approaches:

- In a dual monitor approach, the executive software will set up an SMM monitor in parallel to the executive VMM (i.e., the MVMM), see Chapter 32, “System Management Mode,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C. The SMM monitor is dedicated to handling SMI events without compromising the security of the MVMM. This usage model of handling SMI while a measured environment is active does not require the use of GETSEC[SMCTRL(0)] as event re-enabling after the VMX environment launch is handled implicitly and through separate VMX based controls.

- If a dedicated SMM monitor will not be established and SMIs are to be handled within the measured environment, then GETSEC[SMCTRL(0)] can be used by the executive software to re-enable SMI that has been masked as a result of SENTER.

Table 7-11 defines the processor context in which GETSEC[SMCTRL(0)] can be used and which events will be unmasked. Note that the events that are unmasked are dependent upon the currently operating processor context.

<table>
<thead>
<tr>
<th>ILP Mode of Operation</th>
<th>SMCTRL execution action</th>
</tr>
</thead>
<tbody>
<tr>
<td>In VMX non-root operation</td>
<td>VM exit</td>
</tr>
<tr>
<td>SENTERFLAG = 0</td>
<td>#GP(0), illegal context</td>
</tr>
<tr>
<td>In authenticated code execution mode</td>
<td>#GP(0), illegal context</td>
</tr>
<tr>
<td>(ACMODEFLAG = 1)</td>
<td></td>
</tr>
<tr>
<td>SENTERFLAG = 1, not in VMX operation, not in SMM</td>
<td>Unmask SMI</td>
</tr>
<tr>
<td>SENTERFLAG = 1, in VMX root operation, not in SMM</td>
<td>Unmask SMI if SMM monitor is not configured, otherwise #GP(0)</td>
</tr>
<tr>
<td>SENTERFLAG = 1, In VMX root operation, in SMM</td>
<td>#GP(0), illegal context</td>
</tr>
</tbody>
</table>
Operation

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) OR (EFLAGS.VM=1))
    THEN #GP(0);
ELSE IF ((EBX=0) and (SENTERFLAG=1) and (ACMODEFLAG=0) and (IN_SMM=0) and
        (((in VMX root operation) and (SMM monitor not configured)) or (not in VMX operation))
        )
    THEN unmask SMI;
ELSE
    #GP(0);
END

Flags Affected

None.

Use of Prefixes

LOCK Causes #UD.
REP* Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size Causes #UD.
NP 66/F2/F3 prefixes are not allowed.
Segment overrides Ignored.
Address size Ignored.
REX Ignored.

Protected Mode Exceptions

#UD If CR4.SMXE = 0.
    If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
    If in VMX root operation.
    If a protected partition is not already active or the processor is currently in authenticated code mode.
    If the processor is in SMM.
    If the SMM monitor is not configured.

Real-Address Mode Exceptions

#UD If CR4.SMXE = 0.
    If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SMCTRL] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.
    If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0) GETSEC[SMCTRL] is not recognized in virtual-8086 mode.
Compatibility Mode Exceptions
All protected mode exceptions apply.

64-Bit Mode Exceptions
All protected mode exceptions apply.

VM-exit Condition
Reason (GETSEC) If in VMX non-root operation.
**GETSEC[_WAKEUP]—Wake Up Sleeping Processors in Measured Environment**

### Description

The GETSEC[_WAKEUP] leaf function broadcasts a wake-up message to all logical processors currently in the SENTER sleep state. This GETSEC leaf must be executed only by the ILP, in order to wake-up the RLPs. Responding logical processors (RLPs) enter the SENTER sleep state after completion of the SENTER rendezvous sequence.

The GETSEC[_WAKEUP] instruction may only be executed:

- In a measured environment as initiated by execution of GETSEC[_SENTER].
- Outside of authenticated code execution mode.
- Execution is not allowed unless the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0.
- In addition, the logical processor must be designated as the boot-strap processor as configured by setting IA32_APIC_BASE.BSP = 1.

If these conditions are not met, attempts to execute GETSEC[_WAKEUP] result in a general protection violation.

An RLP exits the SENTER sleep state and start execution in response to a WAKEUP signal initiated by ILP’s execution of GETSEC[_WAKEUP]. The RLP retrieves a pointer to a data structure that contains information to enable execution from a defined entry point. This data structure is located using a physical address held in the Intel® TXT-capable chipset configuration register LT.MLE.JOIN. The register is publicly writable in the chipset by all processors and is not restricted by the Intel® TXT-capable chipset configuration register lock status. The format of this data structure is defined in Table 7-12.

#### Table 7-12. RLP MVM JOIN Data Structure

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GDT limit</td>
</tr>
<tr>
<td>4</td>
<td>GDT base pointer</td>
</tr>
<tr>
<td>8</td>
<td>Segment selector initializer</td>
</tr>
<tr>
<td>12</td>
<td>EIP</td>
</tr>
</tbody>
</table>

The MLE JOIN data structure contains the information necessary to initialize RLP processor state and permit the processor to join the measured environment. The GDTR, LIP, and CS, DS, SS, and ES selector values are initialized using this data structure. The CS selector index is derived directly from the segment selector initializer field; DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE = 0, LIMIT = FFFFFH, G = 1, D = 1, P = 1, S = 1; read/write/access for DS, SS, and ES; and execute/read/access for CS. It is the responsibility of external software to establish a GDT pointed to by the MLE JOIN data structure that contains descriptor entries consistent with the implicit settings initialized by the processor (see Table 7-6). Certain states from the content of Table 7-12 are checked for consistency by the processor prior to execution. A failure of any consistency check results in the RLP aborting entry into the protected environment and signaling an Intel® TXT shutdown condition. The specific checks performed are documented later in this section. After successful completion of processor consistency checks and subsequent initialization, RLP execution in the measured environment begins from the entry point at offset 12 (as indicated in Table 7-12).
Operation

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=0) or (in VMX operation) or
        (IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;

RLP_SIPI_WAKEUP_FROM_SENTER_ROUTINE: (RLP Only)

WHILE (no SignalWAKEUP event);
IF (IA32_SMM_MONITOR_CTL[0] ≠ ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
    Mask SMI pin event;
    Mask A20M, and NMI external pin events (unmask INIT);
    Mask SignalWAKEUP event;
    Invalidate processor TLB(s);
    Drain outgoing transactions;
    TempGDTRLIMIT := LOAD(LT.MLE.JOIN);
    TempGDTRBASE := LOAD(LT.MLE.JOIN+4);
    TempSegSel := LOAD(LT.MLE.JOIN+8);
    TempEIP := LOAD(LT.MLE.JOIN+12);
    IF (TempGDTLimit & FFFF0000h)
        THEN TXT-SHUTDOWN(#BadJOINFormat);
    IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
        THEN TXT-SHUTDOWN(#BadJOINFormat);
    IF ((TempSegSel.TI=1) or (TempSegSel.RPL ≠ 0))
        THEN TXT-SHUTDOWN(#BadJOINFormat);
    CR0.[PG,CD,AM,WP] := 0;
    CR0.[NE,PE] := 1;
    CR4 := 00004000h;
    EFLAGS := 00000002h;
    IA32_EFER := 0;
    GDTR.BASE := TempGDTRBASE;
    GDTR.LIMIT := TempGDTRLIMIT;
    CS.SEL := TempSegSel;
    CS.BASE := 0;
    CS.LIMIT := FFFFFh;
    CS.G := 1;
    CS.D := 1;
    CS.AR := 9Bh;
    DS.SEL := TempSegSel+8;
    DS.BASE := 0;
    DS.LIMIT := FFFFFh;
    DS.G := 1;
DS.D := 1;
DS.AR := 93h;
SS := DS;
ES := DS;
DR7 := 00000400h;
IA32_DEBUGCTL := 0;
EIP := TempEIP;
END;

Flags Affected
None.

Use of Prefixes
- **LOCK** Causes #UD.
- **REP** Causes #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
- **Operand size** Causes #UD.
- **NP** 66/F2/F3 prefixes are not allowed.
- **Segment overrides** Ignored.
- **Address size** Ignored.
- **REX** Ignored.

Protected Mode Exceptions
- **#UD** If CR4.SMXE = 0.
  - If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
- **#GP(0)** If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.
  - If in VMX operation.
  - If a protected partition is not already active or the processor is currently in authenticated code mode.
  - If the processor is in SMM.
- **#UD** If CR4.SMXE = 0.
  - If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
- **#GP(0)** GETSEC[WAKEUP] is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
- **#UD** If CR4.SMXE = 0.
  - If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].
- **#GP(0)** GETSEC[WAKEUP] is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
All protected mode exceptions apply.

64-Bit Mode Exceptions
All protected mode exceptions apply.

VM-exit Condition
- **Reason (GETSEC)** If in VMX non-root operation.
This chapter describes the instruction set that is unique to Intel® Xeon Phi™ Processors based on the Knights Landing and Knights Mill microarchitectures. The set is not supported in any other Intel processors. Included are Intel® AVX-512 instructions. For additional instructions supported on these processors, see Chapter 3, “Instruction Set Reference, A-L”; Chapter 4, “Instruction Set Reference, M-U”; Chapter 5, “Instruction Set Reference, V”; and Chapter 6, “Instruction Set Reference, W-Z.”
### PREFETCHWT1—Prefetch Vector Data Into Caches With Intent to Write and T1 Hint

#### Opcode/Description

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 0D /2 PREFETCHWT1 m8</td>
<td>M</td>
<td>V/V</td>
<td>PREFETCHWT1</td>
<td>Move data from m8 closer to the processor using T1 hint with intent to write.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRMx/m (r)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

#### Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into ‘Exclusive’ state via a request for ownership) and a locality hint:

- T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHWT1 instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 9.5, “Memory Optimization Using Prefetch” of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHWT1 instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHWT1 instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHWT1 instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHWT1 instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

#### Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by ‘mem’ into the cache level specified by ‘Level’; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

Prefetch (m8, Level = 1, EXCLUSIVE=1);

#### Flags Affected

All flags are affected.

#### C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch( char const *, int hint= _MM_HINT_ET1);
Protected Mode Exceptions
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
V4FMADDPS/V4FNMADDPS—Packed Single Precision Floating-Point Fused Multiply-Add (4-Iterations)

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.F2.0F38.W0 9A /r V4FMADDPS zmm1[k1]{z}, zmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F38.W0 AA /r V4FNMADDPS zmm1[k1]{z}, zmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply and negate packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

This instruction computes 4 sequential packed fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.
**Operation**

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```python
define NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest := dest

    // reg[] is an array representing the SIMD register file.
    FOR j := 0 to regs_loaded-1:
        FOR i := 0 to kl-1:
            IF k1[i] or *no writemask*:
                IF posneg = 0:
                ELSE:
                    tmpdest.single[i] := RoundFPControl_MXCSR(tmpdest.single[i] + reg[src_base + j].single[i] * msrc.single[j])
            ELSE IF *zeroing*:
                tmpdest.single[i] := 0
        dest := tmpdest
        dest[MAX_VL-1:VL] := 0
```

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)

KL, VL = (16,512)

regs_loaded := 4

src_base := src_reg_id & ~3 // for src1 operand

posneg := 0 if negative form, 1 otherwise

NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg)

**Intel C/C++ Compiler Intrinsic Equivalent**

V4FMADDPS __m512 _mm512_4fmadd_ps(__m512, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_mask_4fmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_maskz_4fmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_4fnmadd_ps(__m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_mask_4fnmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_maskz_4fnmadd_ps(__mmask16, __m512, __m512x4, __m128 *);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Type E2; additionally:

#UD If the EVEX broadcast bit is set to 1.

#UD If the MODRM.mod = 0b11.
**V4FMADDSS/V4FNMADDSS—Scalar Single Precision Floating-Point Fused Multiply-Add (4-Iterations)**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.F2.0F38.W0 0B /r V4FMADDSS xmm1[k1]{z}, xmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LLIG.F2.0F38.W0 AB /r V4FNMADDSS xmm1[k1]{z}, xmm2+3, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4FMAPS</td>
<td>Multiply and negate scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction computes 4 sequential scalar fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if the least significant mask bit is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

**Operation**

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```c
define NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest := dest
    // reg[] is an array representing the SIMD register file.
    IF k1[0] or *no writemask*:
        FOR j := 0 to regs_loaded - 1:
            IF posneg = 0:
                tmpdest.single[0] := RoundFPControl_MXCSR(tmpdest.single[0] - reg[src_base + j].single[0] * msrc.single[j])
            ELSE:
                tmpdest.single[0] := RoundFPControl_MXCSR(tmpdest.single[0] + reg[src_base + j].single[0] * msrc.single[j])
        ELSE IF *zeroing*:
            tmpdest.single[0] := 0
    dest := tmpdest
    dest[MAX_VL-1:VL] := 0
```

8-6 Vol. 2D V4FMADDSS/V4FNMADDSS—Scalar Single Precision Floating-Point Fused Multiply-Add (4-Iterations)
V4FMADDSS and V4FNMADDSS dest[k1], src1, msrc (AVX512)

VL = 128

regs_loaded := 4
src_base := src_reg_id & ~3 // for src1 operand
posneg := 0 if negative form, 1 otherwise
NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg)

Intel C/C++ Compiler Intrinsic Equivalent
V4FMADDSS __m128 _mm_4fmadd_ss(__m128, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_mask_4fmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_maskz_4fmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_4fnmadd_ss(__m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_mask_4fnmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_maskz_4fnmadd_ss(__mmask8, __m128, __m128x4, __m128 *);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions
See Type E2; additionally:
#UD If the EVEX broadcast bit is set to 1.
#UD If the MODRM.mod = 0b11.
VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double Precision Floating-Point Values With Less Than $2^{-23}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.w1 C8 /r VEXP2PD zmm1 [k1]{z}, zmm2/m512/m64bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed double precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

Computes the approximate base-2 exponential evaluation of the double precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than $2^{-23}$ of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


Operation

VEXP2PD

$(K, VL) = (8, 512)$

FOR $j := 0$ TO $KL-1$

\[i := j \times 64\]

IF $k1[j]$ OR "no writemask" THEN

IF (EVEX.b = 1) AND (SRC "is memory")

THEN DEST[i+63:j] := EXP2_23_DP(SRC[63:i])

ELSE DEST[i+63:j] := EXP2_23_DP(SRC[i+63:j])

FI;

ELSE

IF "merging-masking" ; merging-masking

THEN "DEST[i+63:j] remains unchanged"

ELSE ; zeroing-masking

DEST[i+63:j] := 0

FI;

FI;

ENDFOR;
Table 6-44. Special Values Behavior

<table>
<thead>
<tr>
<th>Source Input</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(src)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>+/-0</td>
<td>1.0f</td>
<td>Exact result</td>
</tr>
<tr>
<td>-∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>Integral value N</td>
<td>2^N</td>
<td>Exact result</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

```
VEXP2PD __m512d __m512d_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d __m512d_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d __m512d_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);
```

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow.

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”
VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single Precision Floating-Point Values With Less Than $2^{-23}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1[k1][z], zmm2/m512/m32bcst (sae)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than $2^{-23}$ of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


### Operation

**VEXP2PS**

(\(KL, VL\) = (16, 512))

\[i := j \times 32\]

\[\text{IF} \ k1[j] \ \text{OR} \ \text{"no writemask"} \ \text{THEN}\]

\[\text{IF} \ (EVEX.b = 1) \ \text{AND} \ (\text{SRC \ "is memory"})\]

\[\text{THEN} \ \text{DEST}[i+31:i] := \text{EXP2}_{23}SP(SRC[31:0])\]

\[\text{ELSE} \ \text{DEST}[i+31:i] := \text{EXP2}_{23}SP(SRC[i+31:i])\]

\[\text{FI;}\]

\[\text{ELSE}\]

\[\text{IF} \ \text{"merging-masking"}\]

\[\text{THEN} \ \text{DEST}[i+31:i] \ \text{remains unchanged}\]

\[\text{ELSE}\]

\[\text{DEST}[i+31:i] := 0\]

\[\text{FI;}\]

\[\text{FI;}\]

ENDFOR;
Table 6-45. Special Values Behavior

<table>
<thead>
<tr>
<th>Source Input</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(src)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>+/-0</td>
<td>1.0f</td>
<td>Exact result</td>
</tr>
<tr>
<td>-∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>Integral value N</td>
<td>$2^N$</td>
<td>Exact result</td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXP2PS__m512__mm512_exp2a23_round_ps(__m512 a, int sae);
VEXP2PS__m512__mm512_mask_exp2a23_round_ps(__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS__m512__mm512_maskz_exp2a23_round_ps(__mmask16 m, __m512 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”
VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch
Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T0 Hint

## Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /vsib VGATHERPF0DPS vm32z {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /vsib VGATHERPF0QPS vm64z {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /vsib VGATHERPF0DPD vm32y {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T0 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /vsib VGATHERPF0QPD vm64z {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T0 hint.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg(R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

## Description
The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into a location in the cache hierarchy specified by a locality hint (T0):
- T0 (temporal data)—prefetch data into the first level cache.
- [PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.
- [PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:
1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

## Operation
BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by ‘mem’ into the cache level specified by ‘Level’; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
VGATHERPF0DPS (EVEX Encoded Version)
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j]
      Fetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
   FI;
ENDFOR

VGATHERPF0DPD (EVEX Encoded Version)
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   k := j * 32
   IF k1[j]
      Fetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
   FI;
ENDFOR

VGATHERPF0QPS (EVEX Encoded Version)
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j]
      Fetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
   FI;
ENDFOR

VGATHERPF0QPD (EVEX Encoded Version)
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j]
      Fetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
   FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-62, “Type E12NP Class Exception Conditions.”
VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0.C6 /2 /vsib VGATHERPF1DPS vm32z {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0.C7 /2 /vsib VGATHERPF1QPS vm64z {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1.C6 /2 /vsib VGATHERPF1DPD vm32y {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T1 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1.C7 /2 /vsib VGATHERPF1QPD vm64z {k1}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double precision data using opmask k1 and T1 hint.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R): VSIB:base, VectorReg(R):VSIB:index</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one. Lines prefetched are loaded into a location in the cache hierarchy specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.
VINDEX stands for the memory operand vector of indices (a vector register).
SCALE stands for the memory operand scalar (1, 2, 4 or 8).
DISP is the optional 1, 2 or 4 byte displacement.
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
VGATHERPF1DPS (EVEX Encoded Version)
(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
 i := j * 32
 IF k1[j]
  Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
 FI;
ENDFOR

VGATHERPF1DPD (EVEX Encoded Version)
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
 i := j * 64
 k := j * 32
 IF k1[j]
  Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
 FI;
ENDFOR

VGATHERPF1QPS (EVEX Encoded Version)
(KL, VL) = (8, 256)
FOR j := 0 TO KL-1
 i := j * 64
 IF k1[j]
  Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
 FI;
ENDFOR

VGATHERPF1QPD (EVEX Encoded Version)
(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
 i := j * 64
 k := j * 64
 IF k1[j]
  Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
 FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent
VGATHERPF1DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-62, "Type E12NP Class Exception Conditions."
VP4DPwSSDS—Dot Product of Signed Words With Dword Accumulation and Saturation (4-Iterations)

**Description**

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation and signed saturation. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a “no masking” encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

**Operation**

(src_reg_id) is the 5 bit index of the vector register specified in the instruction as the src1 register.

\[
\text{VP4DPwSSDS} \text{ dest, src1, src2} \\
(KL,VL) = (16,512) \\
N := 4 \\
\text{ORIGDEST} := \text{DEST} \\
\text{src_base} := \text{src_reg_id} \& \sim (N-1) // for src1 operand
\]

**FOR** \(i := 0 \) to \(KL-1\):

**IF** \(k1[i]\) or *no writemask*:

\[
\begin{align*}
&\text{FOR} \ m := 0 \text{ to } N-1: \\
&t := \text{SRC2.dword}[m] \\
&p1dword := \text{reg[src_base+m].word}[2*i] * t\text{.word}[0] \\
&p2dword := \text{reg[src_base+m].word}[2*i+1] * t\text{.word}[1] \\
&\text{DEST.dword}[i] := \text{SIGNED_DWORD_SATURATE} (\text{DEST.dword}[i] + p1dword + p2dword)
\end{align*}
\]

**ELSE IF** *zeroing*:

\[
\text{DEST.dword}[i] := 0
\]

**ELSE**

\[
\text{DEST.dword}[i] := \text{ORIGDEST.dword}[i]
\]

\[
\text{DEST[MAX_VL-1:VL]} := 0
\]
Intel C/C++ Compiler Intrinsic Equivalent

VP4DPWSSDS __m512i __m512_4dpwssds_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSDS __m512i __m512_mask_4dpwssds_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSDS __m512i __m512_maskz_4dpwssds_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4; additionally:

#UD If the EVEX broadcast bit is set to 1.
#UD If the MODRM.mod = 0b11.
**VP4DPWSSD—Dot Product of Signed Words With Dword Accumulation (4-Iterations)**

**Opcode/ Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.F2.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_4VNNIW</td>
<td>Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate resulting signed dwords in zmm1.</td>
</tr>
<tr>
<td>VP4DPWSSD zmm1[k1][z], zmm2+3, m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1_4X</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation; see Figure 8-1 below. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand. This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

**Figure 8-1. Register Source-Block Dot Product of Two Signed Word Operands With Doubleword Accumulation**

NOTES:

1. For illustration purposes, one source-block dot product instance is shown out of the four.
**Operation**

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

VP4DPwSSD dest, src1, src2
(KL,VL) = (16,512)
N := 4

ORIGDEST := DEST
src_base := src_reg_id & ~ (N-1) // for src1 operand

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m := 0 to N-1:
            t := SRC2.dword[m]
            p1dword := reg[src_base+m].word[2*i] * t.word[0]
            p2dword := reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] := DEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE
        DEST.dword[i] := ORIGDEST.dword[i]

DEST[MAX_VL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VP4DPwSSD __m512i __mm512_4dpwssd_epi32(__m512i, __m512ix4, __m128i *);
VP4DPwSSD __m512i __mm512_mask_4dpwssd_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPwSSD __m512i __mm512_maskz_4dpwssd_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Type E4; additionally:

#UD If the EVEX broadcast bit is set to 1.

#UD If the MODRM.mod = 0b11.
VRCP28PD—Approximation to the Reciprocal of Packed Double Precision Floating-Point Values With Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1.CA /r VRCP28PD zmm1 [k1][z], zmm2/m512/m64bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocals ($&lt;2^{-28}$ relative error) of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


### Operation

**VRCP28PD (EVEX Encoded Versions)**

$(KL, VL) = (8, 512)$

FOR $j := 0$ TO $KL - 1$

$i := j * 64$

IF $k1[j]$ OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN $DEST[i+63:j] := RCP_28_DP(1.0/SRC[i+63:j])$;

ELSE $DEST[i+63:j] := RCP_28_DP(1.0/SRC[i+63:j])$;

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:j] remains unchanged*;

ELSE ; zeroing-masking

$DEST[i+63:j] := 0$

FI;

FI;

ENDFOR;

8-20 Vol. 2D VRCP28PD—Approximation to the Reciprocal of Packed Double Precision Floating-Point Values With Less Than $2^{-28}$ Relative Error
**Table 6-46. VRCP28PD Special Cases**

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$0 \leq X &lt; 2^{-1022}$</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>$-2^{-1022} &lt; X \leq -0$</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>$X &gt; 2^{1022}$</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>$X &lt; -2^{1022}$</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>$X = +\infty$</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>$X = -\infty$</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>$X = 2^{-n}$</td>
<td>$2^n$</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>$X = -2^{-n}$</td>
<td>$-2^n$</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28PD __m512d _mm512_rcp28_round_pd ( __m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd( __mmask8 m, __m512d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”
VRCP28SD—Approximation to the Reciprocal of Scalar Double Precision Floating-Point Value With Less Than 2^-28 Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W1 CB /r VRCP28SD xmm1 (k1)[r], xmm2, xmm3/m64 {sae}</td>
<td>A V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocal ((&lt; 2^{-28}) relative error) of the scalar double precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^−28 of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.


### Operation

**VRCP28SD ((EVEX Encoded Versions))**

IF k1[0] OR *no writemask* THEN

\[
\text{DEST}[63: 0] := \text{RCP}_28\_DP(1.0/\text{SRC2}[63: 0]);
\]

ELSE

IF *merging-masking* ; merging-masking

\[
\text{THEN *DEST}[63: 0] \text{remains unchanged*}
\]

ELSE ; zeroing-masking

\[
\text{DEST}[63: 0] := 0
\]

FI;

FI;

ENDFOR;

\[
\text{DEST}[127:64] := \text{SRC1}[127: 64]
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
Table 6-47. VRCP28SD Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^-1022</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^-1022 &lt; X ≤ -0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^1022</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2^1022</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2^n</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^n</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."
VRCP28PS—Approximation to the Reciprocal of Packed Single Precision Floating-Point Values With Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0.CA /r VRCP28PS zmm1 [k1][z], zmm2/m512/m32bcst {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocals ($&lt; 2^{-28}$ relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final results are rounded to $< 2^{-23}$ relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is $\pm\infty$, $\pm0.0$ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


**Operation**

**VRCP28PS (EVEX Encoded Versions)**

(KL, VL) = (16, 512)

FOR $j := 0$ TO KL-1
  $i := j \times 32$
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] := RCP_28_SP(1.0/SRC[31:0]);
      ELSE DEST[i+31:i] := RCP_28_SP(1.0/SRC[i+31:i]);
      Fl;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*;
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      Fl;
    Fl;
  ENDFOR;
Table 6-48. VRCP28PS Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^-126</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^-126 &lt; X ≤ 0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^126</td>
<td>+0.0f</td>
<td>INF</td>
</tr>
<tr>
<td>X &lt; -2^126</td>
<td>-0.0f</td>
<td>-INF</td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td>INF</td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td>-INF</td>
</tr>
<tr>
<td>X = 2^-n</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^-n</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS __m512_rcp28_round_ps ( __m512 a, int sae);
VRCP28PS __m512 __m512_mask_rcp28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS __m512 __m512_maskz_rcp28_round_ps( __mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."
VRCP28SS—Approximation to the Reciprocal of Scalar Single Precision Floating-Point Value With Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 CB /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocal ($&lt;2^{-28}$ relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final result is rounded to $<2^{-23}$ relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is $\pm\infty$, $\pm0.0$ is returned for that element. Also, if any source element is $\pm0.0$, $\pm\infty$ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.


### Operation

**VRCP28SS ((EVEX Encoded Versions))**

IF k1[0] OR *no writemask* THEN

DEST[31: 0] := RCP_28_SP(1.0/SRC2[31: 0]);
ELSE

IF *merging-masking* THEN *DEST[31: 0] remains unchanged*
ELSE ; zeroing-masking

DEST[31: 0] := 0
FI;
FI;
ENDFOR;
DEST[MAXVL-1:128] := 0
Table 6-49. VRCP28SS Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^-126</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^-126 ≤ X ≤ -0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^126</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2^126</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2^-n</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^-n</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 __m128_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 __m128_mask_rcp28_round_ss (__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 __m128_maskz_rcp28_round_ss (__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."
VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double Precision Floating-Point Values With Less Than 2^-28 Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.61 CC/r VRSQRT28PD zmm1 (k1)[z], zmm2/m512/m64bcst (sae)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the Reciprocal square root (&lt;2^-28 relative error) of the packed double precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


**Operation**

VRSQRT28PD (EVEX Encoded Versions)

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask* THEN

IF (EVEX.b = 1) AND (SRC *is memory*)

THEN DEST[i+63:i] := (1.0/ SQRT(SRC[63:0]));

ELSE DEST[i+63:i] := (1.0/ SQRT(SRC[i+63:i]));

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:i] remains unchanged* 

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI;

FI;

ENDFOR;
Table 6-50. VRSQRT28PD Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNaN(input)</td>
<td></td>
</tr>
<tr>
<td>X = 2^{-2n}</td>
<td>2^n</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0 or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +0 or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PD __m512d _mm512_rsqrt28_round_pd(__m512d a, int sae);
VRSQRT28PD __m512d _mm512_mask_rsqrt28_round_pd(__m512d s, __mmask8 m, __m512d a, int sae);
VRSQRT28PD __m512d _mm512_maskz_rsqrt28_round_pd(__mmask8 m, __m512d a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”
VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double Precision Floating-Point Value With Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mod Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W1 CD /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximate reciprocal square root ($&lt;2^{-28}$ relative error) of the scalar double precision floating-point value from xmm3/m64 and stores result in xmm1 with writemask k1. Also, upper double precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than $2^{-28}$ of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of $-0$ must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.


**Operation**

VRSQRT28SD (EVEX Encoded Versions)

IF k1[0] OR *no writemask* THEN

DEST[63: 0] := (1.0 / SQRT(SRC[63: 0]));

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63: 0] remains unchanged*

ELSE ; zeroing-masking

DEST[63: 0] := 0

FI;

FI;

ENDFOR;

DEST[127:64] := SRC1[127: 64]

DEST[MAXVL-1:128] := 0
Table 6-51. VRSQRT28SD Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNaN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>X = 2^{-2n}</td>
<td>2^n</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0 or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +0 or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SD __m128d_mm_rsqrt28_round_sd(__m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d_mm_mask_rsqrt28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int rounding);
VRSQRT28SD __m128d_mm_maskz_rsqrt28_round_sd(__mmask8 m, __m128d a, __m128d b, int rounding);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions.”
VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single Precision Floating-Point Values With Less Than $2^{-28}$ Relative Error

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final results is rounded to < $2^{-23}$ relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


### Operation

**VRSQRT28PS (EVEX Encoded Versions)**

$$(KL, VL) = (16, 512)$$

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] := (1.0 / SQRT(SRC[i+31:i]));
      ELSE DEST[i+31:i] := (1.0 / SQRT(SRC[i+31:i]));
      Fi;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN REST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      Fi;
    Fi;
  ENDIF;
ENDFOR;
### Table 6-52. VRSQRT28PS Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNaN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +\text{INF}$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

#### Intel C/C++ Compiler Intrinsic Equivalent

- VRSQRT28PS __m512 _mm512_rsqrt28_round_ps(__m512 a, int sae);
- VRSQRT28PS __m512 _mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
- VRSQRT28PS __m512 _mm512_maskz_rsqrt28_round_ps(__mmask16 m, __m512 a, int sae);

#### SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

#### Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”
VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single Precision Floating-Point Value With Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 [k1][z], xmm2, xmm3/m32 {sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximate reciprocal square root ($&lt;2^{-28}$ relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1 with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final result is rounded to $< 2^{-23}$ relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of $-0$ must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.


**Operation**

**VRSQRT28SS (EVEX Encoded Versions)**

IF k1[0] OR *no writemask* THEN

DEST[31:0] := (1.0/ SQRT(SRC[31:0]));

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

DEST[31:0] := 0

Fl;

ENDFOR;


DEST[MAXVL-1:128] := 0
### Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_mask_rsqrt28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int rounding);
VRSQRT28SS __m128 _mm_maskz_rsqrt28_round_ss(__mmask8 m, __m128 a, __m128 b, int rounding);

### SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero.

### Other Exceptions

See Table 2-47, "Type E3 Class Exception Conditions."

### Table 6-53. VRSQRT28SS Special Cases

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Result Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNaN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$X = 2^{-2^n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>
VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint With Intent to Write

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /vsib VSCATTERPF0DPS vm32z (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /vsib VSCATTERPF0QPS vm64z (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /vsib VSCATTERPF0DPD vm32y (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /vsib VSCATTERPF0QPD vm64z (k1)</td>
<td>A</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tuple1 Scalar</td>
<td>BaseReg (R): VSIBbase, VectorReg(R): VSIBindex</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight memory locations.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist.

VINDEX stands for the memory operand vector of indices (a vector register).

SCALE stands for the memory operand scalar (1, 2, 4 or 8).

DISP is the optional 1, 2 or 4 byte displacement.

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
**VSCATTERPF0DPS (EVEX Encoded Version)**

(KL, VL) = (16, 512)

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

**VSCATTERPF0DPD (EVEX Encoded Version)**

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

**VSCATTERPF0QPS (EVEX Encoded Version)**

(KL, VL) = (8, 256)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

**VSCATTERPF0QPD (EVEX Encoded Version)**

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERPF0DPS void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512 PREFETCH I32SCATTER_PDO(long base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask PREFETCH I32SCATTER_PDO(long base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_mask PREFETCH I64SCATTER_PDS(long base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_prefetch_i64scatter_PDS(long base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_mask PREFETCH I64SCATTER_PDS(long base, __m512i vdx, int scale, int hint);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-62, “Type E12NP Class Exception Conditions.”
VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values With Signed Dword, Signed Qword Indices Using T1 Hint With Intent to Write

**Description**

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

**Operation**

**BASE_ADDR** stands for the memory operand base address (a GPR); may not exist.

**VINDEX** stands for the memory operand vector of indices (a vector register).

**SCALE** stands for the memory operand scalar (1, 2, 4 or 8).

**DISP** is the optional 1, 2 or 4 byte displacement.

**PREFETCH(mem, Level, State)** Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
**VSCATTERPF1DPS (EVEX Encoded Version)**

(KL, VL) = (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

**VSCATTERPF1DPD (EVEX Encoded Version)**

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

**VSCATTERPF1QPS (EVEX Encoded Version)**

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

**VSCATTERPF1QPD (EVEX Encoded Version)**

(KL, VL) = (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 64
  IF k1[j]
    Fetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERPF1DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_prefetch_i32scatter_ps(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m256i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-62, “Type E12NP Class Exception Conditions.”
Use the opcode tables in this chapter to interpret IA-32 and Intel 64 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3/SSSE3/SSE4, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

**NOTE**

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

### A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined. Cells marked “Reserved-NOP” are also reserved but may behave as NOP on certain processors. Software should not use opcodes corresponding blank cells or cells marked “Reserved-NOP” nor depend on the current behavior of those opcodes.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, "Opcode Look-up Examples for One, Two, and Three-Byte Opcodes."

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating-point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

### A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

#### A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

- **A**: Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- **B**: The VEX.vvvv field of the VEX prefix selects a general purpose register.
**Opcode Map**

C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F EFLAGS/RFLAGS Register.

G The reg field of the ModR/M byte selects a general register (for example, AX (000)).

H The VEX.vvvv field of the VEX prefix selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. For legacy SSE encodings this operand does not exist, changing the instruction to destructive form.

I Immediate data: the operand value is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).

L The upper 4 bits of the 8-bit immediate selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type. (the MSB is ignored in 32-bit mode)

M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).

N The R/M field of the ModR/M byte selects a packed-quaddword, MMX technology register.

O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).

P The reg field of the ModR/M byte selects a packed quadword MMX technology register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).

S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).

U The R/M field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

V The reg field of the ModR/M byte selects a 128-bit XMM register or a 256-bit YMM register, determined by operand type.

W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register, a 256-bit YMM register (determined by operand type), or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).

Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

### A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

- **a** Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).

- **b** Byte, regardless of operand-size attribute.

- **c** Byte or word, depending on operand-size attribute.

- **d** Doubleword, regardless of operand-size attribute.
Vol. 2D A-3

OPCODE MAP

dq  Double-quadword, regardless of operand-size attribute.
p  32-bit, 48-bit, or 80-bit pointer, depending on operand-size attribute.
pd 128-bit or 256-bit packed double precision floating-point data.
pi Quadword MMX technology register (for example: mm0).
ps 128-bit or 256-bit packed single-precision floating-point data.
q  Quadword, regardless of operand-size attribute.
qq Quad-Quadword (256-bits), regardless of operand-size attribute.
s  6-byte or 10-byte pseudo-descriptor.
sd Scalar element of a 128-bit double precision floating data.
ss Scalar element of a 128-bit single-precision floating data.
si Doubleword integer register (for example: eax).
v  Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
w  Word, regardless of operand-size attribute.
x dq or qq based on the operand-size attribute.
y Doubleword or quadword (in 64-bit mode), depending on operand-size attribute.
z  Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

A.2.3  Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding "/x" to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

A.2.4  Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

A.2.4.1  One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

• Instruction mnemonics and operand types using the notations listed in Section A.2
• Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

• A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. Operand types are listed according to notations listed in Section A.2.
A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.

Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSH A; 06H, PUSH ES).

Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg(opcode) bits in the ModR/M byte as an opcode extension (refer to Section A.4).

A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (0FH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, “Instruction Format,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  — Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  — Gv: The reg field of the ModR/M byte selects a general-purpose register.
  — Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).
• By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

• A ModR/M byte is required and is interpreted according to the abbreviations listed in A.1 and Chapter 2, "Instruction Format," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. The operand types are listed according to notations listed in Section A.2.

Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

• 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  — Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.
  — Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.
  — Ib: Immediate data is encoded in the subsequent byte of the instruction.

• The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.

• The last byte is the immediate byte (08H).

• By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

A.2.4.4 VEX Prefix Instructions

Instructions that include a VEX prefix are organized relative to the 2-byte and 3-byte opcode maps, based on the VEX.mmmm field encoding of implied 0F, 0F38H, 0F3AH, respectively. Each entry in the opcode map of a VEX-encoded instruction is based on the value of the opcode byte, similar to non-VEX-encoded instructions.

A VEX prefix includes several bit fields that encode implied 66H, F2H, F3H prefix functionality (VEX.pp) and operand size(opcode information (VEX.L). See chapter 4 for details.

Opcode tables A2-A6 include both instructions with a VEX prefix and instructions without a VEX prefix. Many entries are only made once, but represent both the VEX and non-VEX forms of the instruction. If the VEX prefix is present all the operands are valid and the mnemonic is usually prefixed with a “v”. If the VEX prefix is not present the VEX.vvvv operand is not available and the prefix “v” is dropped from the mnemonic.

A few instructions exist only in VEX form and these are marked with a superscript “v”.

Operand size of VEX prefix instructions can be determined by the operand type code. 128-bit vectors are indicated by 'dq', 256-bit vectors are indicated by 'qq', and instructions with operands supporting either 128 or 256-bit, determined by VEX.L, are indicated by 'x'. For example, the entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported.
A.2.5  Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

<table>
<thead>
<tr>
<th>Superscript Symbol</th>
<th>Meaning of Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A</td>
<td>Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, “Opcode Extensions For One-Byte And Two-byte Opcodes”).</td>
</tr>
<tr>
<td>1B</td>
<td>Use the 0F0B opcode (UD2 instruction), the 0FB9H opcode (UD1 instruction), or the 0FFFH opcode (UD0 instruction) when deliberately trying to generate an invalid opcode exception (#UD).</td>
</tr>
<tr>
<td>1C</td>
<td>Some instructions use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6.</td>
</tr>
<tr>
<td>i64</td>
<td>The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).</td>
</tr>
<tr>
<td>o64</td>
<td>Instruction is only available when in 64-bit mode.</td>
</tr>
<tr>
<td>d64</td>
<td>When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.</td>
</tr>
<tr>
<td>f64</td>
<td>The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).</td>
</tr>
<tr>
<td>v</td>
<td>VEX form only exists. There is no legacy SSE form of the instruction. For Integer GPR instructions it means VEX prefix required.</td>
</tr>
<tr>
<td>v1</td>
<td>VEX128 &amp; SSE forms only exist (no VEX256), when can’t be inferred from the data size.</td>
</tr>
</tbody>
</table>

A.3  ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.
### Table A-2. One-byte Opcode Map: (00H — F7H) *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>Gb, Eb</td>
<td>GV, EV</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>PUSHE5264</td>
</tr>
<tr>
<td>1</td>
<td>ADC</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>PUSHE5264</td>
</tr>
<tr>
<td>2</td>
<td>AND</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, EV</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG=ES (Prefix)</td>
</tr>
<tr>
<td>3</td>
<td>XOR</td>
<td>EB, GB</td>
<td>EV, GV</td>
<td>GB, EB</td>
<td>GV, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG=SS (Prefix)</td>
</tr>
<tr>
<td>4</td>
<td>INC general register / REX5264 Prefixes</td>
<td>EB, Ib</td>
<td>EB</td>
<td>EB</td>
<td>EB</td>
<td>EB</td>
<td>EB</td>
<td>EB</td>
</tr>
<tr>
<td>5</td>
<td>PUSH general register</td>
<td>rAX/r8</td>
<td>rCX/r9</td>
<td>rDX/r10</td>
<td>rBX/r11</td>
<td>rSP/r12</td>
<td>rBP/r13</td>
<td>rSI/r14</td>
</tr>
<tr>
<td>6</td>
<td>POP general register</td>
<td>rAX/r8</td>
<td>rCX/r9</td>
<td>rDX/r10</td>
<td>rBX/r11</td>
<td>rSP/r12</td>
<td>rBP/r13</td>
<td>rSI/r14</td>
</tr>
<tr>
<td>7</td>
<td>PUSHAD / PUSHA immediate byte into byte register</td>
<td>rAX/r8</td>
<td>rCX/r9</td>
<td>rDX/r10</td>
<td>rBX/r11</td>
<td>rSP/r12</td>
<td>rBP/r13</td>
<td>rSI/r14</td>
</tr>
<tr>
<td>8</td>
<td>MOV immediate byte into byte register</td>
<td>AL, Ob</td>
<td>rAX, Ox</td>
<td>Ob, AL</td>
<td>Ox, rAX</td>
<td>MOVs/B</td>
<td>MOVs/W/D/Q</td>
<td>CMPS/B</td>
</tr>
<tr>
<td>9</td>
<td>MOV immediate byte into byte register</td>
<td>AL, Ob</td>
<td>rAX, Ox</td>
<td>Ob, AL</td>
<td>Ox, rAX</td>
<td>MOVs/B</td>
<td>MOVs/W/D/Q</td>
<td>CMPS/B</td>
</tr>
<tr>
<td>10</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>11</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>12</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>13</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>14</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>15</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>16</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>17</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>18</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>19</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>20</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>21</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>22</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>23</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>24</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>25</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>26</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>27</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>28</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>29</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>30</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
<tr>
<td>31</td>
<td>Shift Grp 2A</td>
<td>EB, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
</tr>
</tbody>
</table>

Note: * denotes additional prefixes or suffixes for specific opcodes.
### Table A-2. One-byte Opcode Map: (08H — FFH) *

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>OR</td>
<td></td>
<td>AL,</td>
<td>rAX,</td>
<td>PUSH</td>
<td>CS64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ib</td>
<td>Iz</td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>SBB</td>
<td></td>
<td></td>
<td>PUSHD</td>
<td>DS64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>POPDS64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td></td>
<td></td>
<td>SEG=CS</td>
<td>(Prefix)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DAS64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>CMP</td>
<td></td>
<td></td>
<td>SEG=DS</td>
<td>(Prefix)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AAS64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>DEC64</td>
<td></td>
<td></td>
<td>POP64</td>
<td>(general register / REX64 Prefixes)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>POP64</td>
<td>(Prefix)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>PUSH</td>
<td></td>
<td></td>
<td>IMUL</td>
<td>Gv, Ev, Iz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IMUL</td>
<td>Gv, Ev, Ib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INS/</td>
<td>INSB</td>
<td>INSD</td>
<td>OUTS/</td>
<td>OUTSB</td>
</tr>
<tr>
<td>6</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>EAX</td>
<td></td>
<td></td>
<td>IMUL</td>
<td>Gv, Ev, Ib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INS/</td>
<td>INSB</td>
<td>INSD</td>
<td>OUTS/</td>
<td>OUTSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INS/</td>
<td>INSB</td>
<td>INSD</td>
<td>OUTS/</td>
<td>OUTSB</td>
</tr>
<tr>
<td>7</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>JE</td>
<td></td>
<td></td>
<td>Jcc</td>
<td>f64,</td>
<td>Jb-</td>
<td>Short displacement jump on condition</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td></td>
<td></td>
<td>MOV</td>
<td>Ev,</td>
<td>Sw</td>
<td>LEA</td>
<td>Gv, M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>CBW/</td>
<td>CWDE/</td>
<td>CDQ/</td>
<td>CDQE</td>
<td>far CALL</td>
<td>Ap</td>
<td>FWAIT/</td>
<td>WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>TEST</td>
<td></td>
<td></td>
<td>MOV</td>
<td>immediate word or double into word, double, or quad register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LEA</td>
<td>Gv, M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>ENTER</td>
<td></td>
<td></td>
<td>LEAVE</td>
<td>far RET</td>
<td>far RET</td>
<td>INT3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>LEAVE</td>
<td></td>
<td></td>
<td>LEAVE</td>
<td>far RET</td>
<td>far RET</td>
<td>INT3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td>ESC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Escape to coprocessor instruction set)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL,</td>
<td>Ib</td>
<td>rAX,</td>
<td>Iz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STC</td>
<td></td>
<td>CLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STI</td>
<td></td>
<td>STI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLD</td>
<td></td>
<td>STD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STD</td>
<td></td>
<td>INC/DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Grp 51A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
### Table A-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH)

<table>
<thead>
<tr>
<th>pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Grp 6A</td>
<td>Grp 7A</td>
<td>LAR</td>
<td>Gv, Ew</td>
<td>LSL</td>
<td>Gv, Ew</td>
<td>SYSCALL&lt;sup&gt;64&lt;/sup&gt;</td>
<td>CLTS</td>
</tr>
<tr>
<td>1</td>
<td>vmovups Vp2, Vp2</td>
<td>vmovups Wp2, Wp2</td>
<td>vmovups Vq, Hq, Mq</td>
<td>vmovups Vq, Hq, Uq</td>
<td>vmovups Mq, Vq</td>
<td>vmovups Mq, Vq</td>
<td>vmovups Mq, Vq</td>
<td>vmovups Mq, Vq</td>
</tr>
<tr>
<td>66</td>
<td>vmovupd Vpd, Wpd</td>
<td>vmovupd Wpd, Vpd</td>
<td>vmovupd Vq, Hq, Mq</td>
<td>vmovupd Vq, Hq, Uq</td>
<td>vmovupd Mq, Vq</td>
<td>vmovupd Mq, Vq</td>
<td>vmovupd Mq, Vq</td>
<td>vmovupd Mq, Vq</td>
</tr>
<tr>
<td>F3</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
<td>VMovo Ws, Hs, Vs</td>
</tr>
<tr>
<td>F2</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
<td>MOV Vx, Hx, Wx</td>
</tr>
<tr>
<td>2</td>
<td>WRMSR</td>
<td>RDTSC</td>
<td>RDMSR</td>
<td>RDPMC</td>
<td>SYSENTER</td>
<td>SYSEXIT</td>
<td>GETSEC</td>
<td>CMOVCc, (Gv, Ev) - Conditional Move</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>NO</td>
<td>B/C/NAE</td>
<td>AE/NB/NC</td>
<td>E/Z</td>
<td>NE/NZ</td>
<td>BE/NA</td>
<td>A/NBE</td>
</tr>
<tr>
<td>5</td>
<td>vmovmskps Gy, Up2</td>
<td>vsqrtps Vps, Wps</td>
<td>vrcpps Vps, Wps</td>
<td>vandps Vps, Wps</td>
<td>vandps Vps, Wps</td>
<td>vorps Vps, Wps</td>
<td>vorps Vps, Wps</td>
<td>vxorps Vps, Wps</td>
</tr>
<tr>
<td>66</td>
<td>vmovmskpd Gy,Up2d</td>
<td>vsqrtpd Vpd, Wpd</td>
<td>vrdpps Vpd, Wpd</td>
<td>vandrdps Vpd, Wpd</td>
<td>vandrdps Vpd, Wpd</td>
<td>vorpd Vpd, Wpd</td>
<td>vorpd Vpd, Wpd</td>
<td>vxorpd Vpd, Wpd</td>
</tr>
<tr>
<td>F3</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
<td>vsqrtps Vps, Wps</td>
</tr>
<tr>
<td>6</td>
<td>vpuncpkkbw Vq, Qd</td>
<td>vpuncpkkld Vq, Qd</td>
<td>vpuncpkkltq Vq, Qd</td>
<td>packsswb Vq, Qd</td>
<td>packsswb Vq, Qd</td>
<td>pcmpgtb Vq, Qd</td>
<td>pcmpgtb Vq, Qd</td>
<td>packuswb Vq, Qd</td>
</tr>
<tr>
<td>66</td>
<td>vpuncpkkbwd Vx, Hx, Wx</td>
<td>vpuncpkkldw Vx, Hx, Wx</td>
<td>vpuncpkkldq Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
</tr>
<tr>
<td>F3</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td>vpacksswb Vx, Hx, Wx</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>vpsuhf Pq, Qq, Ib</td>
<td>vpsuhf (Grp 12&lt;sup&gt;65&lt;/sup&gt;)</td>
<td>vpsuhf (Grp 13&lt;sup&gt;65&lt;/sup&gt;)</td>
<td>vpsuhf (Grp 14&lt;sup&gt;65&lt;/sup&gt;)</td>
<td>vpsuhf (Grp 14&lt;sup&gt;65&lt;/sup&gt;)</td>
<td>vpsuhf (Grp 14&lt;sup&gt;65&lt;/sup&gt;)</td>
<td>vpsuhf (Grp 14&lt;sup&gt;65&lt;/sup&gt;)</td>
<td>vpsuhf (Grp 14&lt;sup&gt;65&lt;/sup&gt;)</td>
</tr>
<tr>
<td>66</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
</tr>
<tr>
<td>F3</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
</tr>
<tr>
<td>F2</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
<td>vpsuhf Vx, Wx, Ib</td>
</tr>
</tbody>
</table>
### Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH) *

<table>
<thead>
<tr>
<th>px</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INVD</td>
<td>WBINVD</td>
<td>2-byte Illegal Opcodes (UD2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Prefetch C' (Grp 16)</td>
<td>Reserved-NOP</td>
<td>bndtx</td>
<td>bndtx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved-NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>vmovaps</td>
<td>vmovaps</td>
<td>cvtp2ps</td>
<td>cvtlps2pi</td>
<td>vcmiiss</td>
<td>vcmiiss</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vps, Vps</td>
<td>Vps, Qpi</td>
<td>Vps, Wps</td>
<td>Vps, Qpi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>vmovapd</td>
<td>vmovapd</td>
<td>cvtp2pd</td>
<td>cvtp2pd</td>
<td>vcmiiss</td>
<td>vcmiiss</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wpd, Vpd</td>
<td>Vpd, Qpi</td>
<td>Mpd, Vpd</td>
<td>Mpd, Qpi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>vcvtsi2ds</td>
<td>vcvtsi2ds</td>
<td>Vss, Hss, Ey</td>
<td>Vss, Hss, Ey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>vcvtsi2ds</td>
<td>vcvtsi2ds</td>
<td>Vsd, Hsd, Ey</td>
<td>Vsd, Hsd, Ey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3-byte escape (Table A-4)</td>
<td>3-byte escape (Table A-5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td>NS</td>
<td>PIPE</td>
<td>NP/PO</td>
<td>LN/GE</td>
<td>NL/GE</td>
<td>LE/NG</td>
<td>NLE/G</td>
</tr>
<tr>
<td>5</td>
<td>vaddps</td>
<td>vmulps</td>
<td>cvtp2ps</td>
<td>vsubps</td>
<td>vmulp</td>
<td>vmulp</td>
<td>vmasxps</td>
<td>vmasxps</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>vaddpd</td>
<td>vmulpd</td>
<td>cvtp2ps</td>
<td>vsubpd</td>
<td>vmulp</td>
<td>vmulp</td>
<td>vmasxps</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>vaddss</td>
<td>vaddss</td>
<td>vaddss</td>
<td>vaddss</td>
<td>vaddss</td>
<td>vaddss</td>
<td>vaddss</td>
</tr>
<tr>
<td></td>
<td>Vss, Hss, Wss</td>
<td>Vss, Hss, Wss</td>
<td>Vpd, Wps</td>
<td>Vpd, Wps</td>
<td>Vpd, Wps</td>
<td>Vpd, Wps</td>
<td>Vpd, Wps</td>
<td>Vpd, Wps</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>vaddsd</td>
<td>vaddsd</td>
<td>vaddsd</td>
<td>vaddsd</td>
<td>vaddsd</td>
<td>vaddsd</td>
<td>vaddsd</td>
</tr>
<tr>
<td>6</td>
<td>punpckhbw</td>
<td>punpckhbw</td>
<td>vpacksd</td>
<td>vpacksd</td>
<td>vpunpckhbw</td>
<td>vpunpckhbw</td>
<td>vpunpckhbw</td>
<td>vpunpckhbw</td>
</tr>
<tr>
<td></td>
<td>Pq, Qd</td>
<td>Pq, Qd</td>
<td>Packsdw</td>
<td>Packsdw</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
</tr>
<tr>
<td></td>
<td>66</td>
<td>vpnunpckhbw</td>
<td>vpnunpckhbw</td>
<td>vpnunpckhbw</td>
<td>vpnunpckhbw</td>
<td>vpnunpckhbw</td>
<td>vpnunpckhbw</td>
<td>vpnunpckhbw</td>
</tr>
<tr>
<td></td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VMREAD</td>
<td>VMWRITE</td>
<td>vmovd/q</td>
<td>vmovd/q</td>
<td>vmovd/q</td>
<td>vmovd/q</td>
<td>vmovd/q</td>
<td>vmovd/q</td>
</tr>
<tr>
<td></td>
<td>Ey, Gy</td>
<td>Ey, Gy</td>
<td>Ed, Ey</td>
<td>Ed, Ey</td>
<td>Ed, Ey</td>
<td>Ed, Ey</td>
<td>Ed, Ey</td>
<td>Ed, Ey</td>
</tr>
<tr>
<td>66</td>
<td>vhaddpd</td>
<td>vhsubpd</td>
<td>vhaddpd</td>
<td>vhaddpd</td>
<td>vhaddpd</td>
<td>vhaddpd</td>
<td>vhaddpd</td>
<td>vhaddpd</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>vmovq</td>
<td>vmovq</td>
<td>vmovq</td>
<td>vmovq</td>
<td>vmovq</td>
<td>vmovq</td>
<td>vmovq</td>
</tr>
<tr>
<td></td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
<td>Vq, Wq</td>
</tr>
<tr>
<td>2</td>
<td>vshaddps</td>
<td>vshubps</td>
<td>vshaddps</td>
<td>vshaddps</td>
<td>vshaddps</td>
<td>vshaddps</td>
<td>vshaddps</td>
<td>vshaddps</td>
</tr>
<tr>
<td>pfx</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>NO</td>
<td>B/C/NAE</td>
<td>AE/NB/NC</td>
<td>E/Z</td>
<td>NE/NZ</td>
<td>BE/NA</td>
<td>A/NBE</td>
</tr>
<tr>
<td>A</td>
<td>O</td>
<td>NO</td>
<td>B/C/NAE</td>
<td>AE/NB/NC</td>
<td>E/Z</td>
<td>NE/NZ</td>
<td>BE/NA</td>
<td>A/NBE</td>
</tr>
<tr>
<td></td>
<td>SETcc, Eb - Byte Set on condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>PUSH64 FS</td>
<td>POP64 FS</td>
<td>CPUID</td>
<td>BT Ev, Gv</td>
<td>SHLD Ev, Gv, Ib</td>
<td>SHLD Ev, Gv, CL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>CMPXCHG Eb, Gb</td>
<td>Ev, Gv</td>
<td>LSS Gv, Mp</td>
<td>BTR Ev, Gv</td>
<td>LFS Gv, Mp</td>
<td>LGS Gv, Mp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVZX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>XADD Eb, Gb</td>
<td>XADD Ev, Gv</td>
<td>Vcmpss Vps,Hps,Wps,lb</td>
<td>MOVNTi Pq, Ry/Mw,lb</td>
<td>Vpsnw Pq, Ry/Mw,lb</td>
<td>Vpsnw Gd, Udq, Ib</td>
<td>Vshufps Vps,Hps,Wps,lb</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vcmpsd Vpd,Hpd,Wpd,lb</td>
<td>Vpsnw Vpd,Hpd,Wpd,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>Vcmpss Vss,Hss,Wss,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>Vcmpsd Vsd,Hsd,Wsd,lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Pslr Pq, Qq</td>
<td>Pslrd Pq, Qq</td>
<td>Pslrq Pq, Qq</td>
<td>Paddq Pq, Qq</td>
<td>Pmulw Pq, Qq</td>
<td>Vmoveq Wq, Qq</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vpsrlw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>Vpsrad Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>Vpsrlq Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vpsrad Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>Vpsrlq Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Pavevb Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vpsraw Vx, Hx, Wx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Jcc64, Jz - Long-displacement jump on condition
Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) *

<table>
<thead>
<tr>
<th>pfx</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>S</td>
<td>NS</td>
<td>PE</td>
<td>NP/PO</td>
<td>L</td>
<td>NL/GE</td>
<td>LE/NG</td>
<td>NLE/G</td>
</tr>
<tr>
<td>C</td>
<td>S</td>
<td>NS</td>
<td>PE</td>
<td>NP/PO</td>
<td>L</td>
<td>NL/GE</td>
<td>LE/NG</td>
<td>NLE/G</td>
</tr>
<tr>
<td>D</td>
<td>PUShp64</td>
<td>POp64</td>
<td>RSM</td>
<td>BTS</td>
<td>SHRD</td>
<td>SHRD</td>
<td>(Grp 15)</td>
<td>IMUL</td>
</tr>
<tr>
<td>E</td>
<td>JMPE</td>
<td>Grp 10</td>
<td>Grp 8</td>
<td>BTC</td>
<td>BF</td>
<td>Gv</td>
<td>Ev</td>
<td>MOVSX</td>
</tr>
<tr>
<td>F</td>
<td>POPCNT</td>
<td>Gv</td>
<td>Ev</td>
<td>TZCNT</td>
<td>L</td>
<td>Gv</td>
<td>Ev</td>
<td></td>
</tr>
</tbody>
</table>

| C   | RAX/EAX/ | RCX/ECX/ | RDX/EDX/ | RBX/EBX/ | RSP/ESP/ | RBP/EBP/ | RSI/ESI/ | RDI/EDI/ |
|-----| R8/R8D |
| D   | Psubsb | Psubsw | Pminub | Pand | Paddusb | Paddusw | Pmaxub | Pandn |
| E   | 66 | Vpsubsb | Vpsubsw | Vpminub | Vpand | Vpadusb | Vpadusw | Vpmaxub | Vpandn |
| F   | F3 |
| F2 |

NOTES:
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
### Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) *

<table>
<thead>
<tr>
<th>Prefix (Fx)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pshufb</td>
<td>phaddw</td>
<td>phadd</td>
<td>phadd</td>
<td>pmad.ddw</td>
<td>pshub</td>
<td>pshub</td>
<td>pshub</td>
</tr>
<tr>
<td></td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
<td>Pq, Qq</td>
</tr>
<tr>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vpshufb</td>
<td>vphaddw</td>
<td>vphadd</td>
<td>vphadd</td>
<td>vphadd</td>
<td>vphadd</td>
<td>vphadd</td>
<td>vphadd</td>
</tr>
<tr>
<td></td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pbldvd</td>
<td>pbldvd</td>
<td>pbldvd</td>
<td>pbldvd</td>
<td>pbldvd</td>
<td>pbldvd</td>
<td>pbldvd</td>
<td>pbldvd</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
</tr>
<tr>
<td></td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
<td>vpmovsdb</td>
</tr>
<tr>
<td></td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
<td>Vx, Ux/Mq</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vpmovsd</td>
<td>vpmovsd</td>
<td>vpmovsd</td>
<td>vpmovsd</td>
<td>vpmovsd</td>
<td>vpmovsd</td>
<td>vpmovsd</td>
<td>vpmovsd</td>
</tr>
<tr>
<td></td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
<td>Vx, Ux/Mw</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Values indicated by stars are currently unsupported or not specified.
**Table A-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H)**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>px</td>
<td>p8</td>
<td>p9</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>psignb</td>
<td>Vx, Hx, Wx</td>
<td>vpabsb</td>
<td>Vx, Hx, Wx</td>
<td>pabsw</td>
</tr>
<tr>
<td>66</td>
<td>8</td>
<td>p8</td>
<td>Qq</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
<td>Vx, Hx, Wx</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>vbroadcastss⁷</td>
<td>Vx, Wd</td>
<td>vpabsb</td>
<td>Vx, Hx, Wx</td>
<td>pabsw</td>
</tr>
<tr>
<td>66</td>
<td>7</td>
<td>vbroadcastsd⁷</td>
<td>Vx, Wq</td>
<td>vpabsb</td>
<td>Vx, Wq</td>
<td>Pq</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>vpmuldq</td>
<td>Vx, Hx, Wx</td>
<td>vpmuldq</td>
<td>Vx, Hx, Wx</td>
<td>vpmuldq</td>
</tr>
<tr>
<td>66</td>
<td>3</td>
<td>vpmulnds</td>
<td>Vx, Hx, Wx</td>
<td>vpmulnds</td>
<td>Vx, Hx, Wx</td>
<td>vpmulnds</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>vpmulnd</td>
<td>Vx, Hx, Wx</td>
<td>vpmulnd</td>
<td>Vx, Hx, Wx</td>
<td>vpmulnd</td>
</tr>
<tr>
<td>66</td>
<td>5</td>
<td>vpmulndo</td>
<td>Vx, Hx, Wx</td>
<td>vpmulndo</td>
<td>Vx, Hx, Wx</td>
<td>vpmulndo</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>vpmulndf</td>
<td>Vx, Hx, Wx</td>
<td>vpmulndf</td>
<td>Vx, Hx, Wx</td>
<td>vpmulndf</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>vfmadd132ps/d</td>
<td>Vx, Hx, Wx</td>
<td>vfmadd132ps/d</td>
<td>Vx, Hx, Wx</td>
<td>vfmadd132ps/d</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>vfmadd132ps/d</td>
<td>Vx, Hx, Wx</td>
<td>vfmadd132ps/d</td>
<td>Vx, Hx, Wx</td>
<td>vfmadd132ps/d</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>sha1nexte</td>
<td>Vdq, Wdq</td>
<td>sha1nexte</td>
<td>Vdq, Wdq</td>
<td>sha1nexte</td>
</tr>
<tr>
<td>66</td>
<td>D</td>
<td>VAESIMC</td>
<td>Vdq, Wdq</td>
<td>VAESIMC</td>
<td>Vdq, Wdq</td>
<td>VAESIMC</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>F3</td>
<td>F2</td>
<td>F3</td>
<td>F2</td>
<td>F3</td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are OF 3AH) *

<table>
<thead>
<tr>
<th>pfx</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>66</td>
<td>vpermqVq Wqq, Ib</td>
<td>vpermqWq Vqq, Wqq, Ib</td>
<td>vblendvVx Hx, Wx, Ib</td>
<td>vpermilpsVx Wx, Ib</td>
<td>vpermilpdVx Wx, Ib</td>
<td>vperm2f128Vq Wqq, Wqq, Ib</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>66</td>
<td>vpinsrbVdq Hdq, Ry/Mb, Ib</td>
<td>vinsertpsVdq Hdq, Udq/Md, Ib</td>
<td>vpermrdqVdq Hdq, Ey, Ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>vpinsrbVdq Hdq, Ry/Mb, Ib</td>
<td>vinsertpsVdq Hdq, Udq/Md, Ib</td>
<td>vpermrdqVdq Hdq, Ey, Ib</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>66</td>
<td>vdpdpVx Hx, Wx, Ib</td>
<td>vdpdpVdq Hdq, Wdq, Ib</td>
<td>vmpsadbwVx Hx, Wx, Ib</td>
<td>vpermulsdqVdq Hdq, Wdq, Ib</td>
<td>vperm2128Vq Wqq, Wqq, Ib</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>66</td>
<td>vcmpestrmVdq, Wdq, Ib</td>
<td>vcmpestrmVdq, Wdq, Ib</td>
<td>vcmpestrmVdq, Wdq, Ib</td>
<td>vcmpestrmVdq, Wdq, Ib</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* The first two bytes are 0F 3AH.
### Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH) *

<table>
<thead>
<tr>
<th>pfX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>66</td>
<td>vroundps</td>
<td>Vx,Wx,lb</td>
<td>vroundpd</td>
<td>Vx,Wx,lb</td>
<td>vroundss</td>
<td>Vss,Wss,lb</td>
<td>vroundsd</td>
<td>Vsd,Wsd,lb</td>
<td>vbldndps</td>
<td>Vx,Hx,Wx,lb</td>
<td>vbldndp</td>
<td>Vx,Hx,Wx,lb</td>
<td>vblenntd</td>
<td>Vx,Hx,Wx,lb</td>
<td>vblendw</td>
</tr>
<tr>
<td>1</td>
<td>66</td>
<td>vinsertf128</td>
<td>Vqq,Hqq,Wqq,lb</td>
<td>vextractf128</td>
<td>Wdq,Vqq,lb</td>
<td>vinsertl128</td>
<td>Vqq,Hqq,Wqq,lb</td>
<td>vextracr128</td>
<td>Wdq,Vqq,lb</td>
<td>vblendps</td>
<td>Vblenpvps</td>
<td>Vx,Hx,Wx,lb</td>
<td>vblendpd</td>
<td>Vx,Hx,Wx,lb</td>
<td>vblendpd</td>
<td>Vx,Hx,Wx,lb</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td>vinsertf128</td>
<td>Vqq,Hqq,Wqq,lb</td>
<td>vextractf128</td>
<td>Wdq,Vqq,lb</td>
<td>vinsertl128</td>
<td>Vqq,Hqq,Wqq,lb</td>
<td>vextracr128</td>
<td>Wdq,Vqq,lb</td>
<td>vblendps</td>
<td>Vblenpvps</td>
<td>Vx,Hx,Wx,lb</td>
<td>vblendpd</td>
<td>Vx,Hx,Wx,lb</td>
<td>vblendpd</td>
<td>Vx,Hx,Wx,lb</td>
</tr>
<tr>
<td>4</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

<table>
<thead>
<tr>
<th>mod</th>
<th>nnn</th>
<th>R/M</th>
</tr>
</thead>
</table>

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-4. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:
- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example A-5. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3, and Table A-6:
- 0F indicates that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

A.4.2 Opcode Extension Tables

See Table A-6 below.
## Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Group</th>
<th>Mod 7,6</th>
<th>pfx</th>
<th>Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80-83</td>
<td>1</td>
<td>mem, 11B</td>
<td></td>
<td>ADD, OR, ADC, SBB, AND, SUB, XOR, CMP</td>
</tr>
<tr>
<td>8F</td>
<td>1A</td>
<td>mem, 11B</td>
<td>POP</td>
<td></td>
</tr>
<tr>
<td>C0,C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL</td>
<td>2</td>
<td>mem, 11B</td>
<td></td>
<td>ROL, ROR, RCL, RCR, SHL/SAL, SHR, SAR</td>
</tr>
<tr>
<td>F6, F7</td>
<td>3</td>
<td>mem, 11B</td>
<td>TEST</td>
<td>NOT, NEG, MUL, AL/AX, IMUL, AL/AX, DIV, AL/HAX, IDIV, AL/HAX</td>
</tr>
<tr>
<td>FE</td>
<td>4</td>
<td>mem, 11B</td>
<td>INC Ebb</td>
<td>DEC Eb</td>
</tr>
</tbody>
</table>
| FF     | 5     | mem, 11B | INC Ev | DEC Ev, near CALL  
|        |       |         |      | far CALL, near JMP  
|        |       |         |      | far JMP, PUSHRd |
| 0F 00  | 6     | mem, 11B | SLDT Rv/Mw, STR Rv/Mw, LLDT Ev, LTR Ev, VERR Ew, VERW Ew |
| 0F 01  | 7     | 11B | VMCALL (001), VMLAUNCH (010), VMRESUME (011), VMXOFF (100)  
|        |       |      | MONITOR (000), MWAIT (010), CLAC (010), STAC (011), ENCLS (111)  
|        |       |      | XGETBV (000), KSETBV (001), VMFUNC (100), XEND (101), XTEST (110), ENCLU (111)  
|        |       |      | SWAPGS (000), RDTSCP (001) |
| 0F 0A  | 8     | mem, 11B | CMPXCHSB Md, CMPXCHG16B Mdq, CMPXCHG16B Msq |
| C6     | 11B   | mem  | MOV Eb, Ib |
| C7     | 11B   | mem  | MOV Ev, Iz |
| 0F 71  | 12    | 11B | pslw Nq, Ib, vpsrlw Hx,Ux,lb |
| 0F 72  | 13    | 11B | pslid Nq, Ib, vpsrlid Hx,Ux,lb |
| 0F 73  | 14    | 11B | psllw Nq, Ib, vpsllw Hx,Ux,lb |

*Groups 1, 1A, 1B, 1C are not shown.*
### Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number * (Contd.)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Group</th>
<th>Mod 7,6</th>
<th>prefix</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AE</td>
<td>15</td>
<td>mem</td>
<td>fxsave</td>
<td>fxrstor</td>
<td>ldmxcsr</td>
<td>strmxcsr</td>
<td>XSAVE</td>
<td>XRSTOR</td>
<td>XSAVEOPT</td>
<td>clflush</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
<td></td>
<td>F3</td>
<td>RDFSBASE Ry</td>
<td>RDGSBASE Ry</td>
<td>WRFSBASE Ry</td>
<td>WRGSBASE Ry</td>
<td>llence</td>
<td>mlfence</td>
<td>sfence</td>
</tr>
<tr>
<td>0F 18</td>
<td>16</td>
<td>mem</td>
<td>prefetch</td>
<td>NTA</td>
<td>prefetch</td>
<td>T0</td>
<td>prefetch</td>
<td>T1</td>
<td>prefetch</td>
<td>T2</td>
<td>Reserved NOP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.0F38 F3</td>
<td>17</td>
<td>mem</td>
<td></td>
<td></td>
<td>BLSR* By, Ey</td>
<td>BLSMSK* By, Ey</td>
<td>BLSI* By, Ey</td>
<td>Reserved NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1-and 2-byte opcodes (see A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example A-6. Opcode with ModR/M Byte in the 00H through BFH Range
DD0504000000H can be interpreted as follows:

• The instruction encoded with this opcode can be located in Section . Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).

• The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example A-7. Opcode with ModR/M Byte outside the 00H through BFH Range
D8C1H can be interpreted as follows:

• This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.

• In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte (refer to Figure A.4)</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD single-real</td>
<td>FMUL single-real</td>
<td>FCOM single-real</td>
<td>FCOMP single-real</td>
<td>FSUB single-real</td>
<td>FSUBR single-real</td>
<td>FDIV single-real</td>
<td>FDIVR single-real</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>FADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>FCOM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>FSUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td>FDIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000B</td>
<td>001B</td>
<td>010B</td>
<td>011B</td>
<td>100B</td>
<td>101B</td>
<td>110B</td>
<td>111B</td>
<td></td>
</tr>
<tr>
<td>FLD single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP single-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDENV 14/28 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDCW 2 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTENV 14/28 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTCW 2 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

#### A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIADD</td>
<td>dword-integer</td>
<td>FIMUL</td>
<td>dword-integer</td>
<td>FICOM</td>
<td>dword-integer</td>
<td>FICOMP</td>
<td>dword-integer</td>
<td>FISUB</td>
</tr>
</tbody>
</table>

### NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-12 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

<table>
<thead>
<tr>
<th>C</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>CMOVB</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
<tr>
<td>D</td>
<td>FCMOVBE</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTTP</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIST</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FISTP</td>
<td>dword-integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLD</td>
<td>extended-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP</td>
<td>extended-real</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>FCMOVNB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>FCMOVNBE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>FCLEX</td>
<td>FINIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>FCOMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
</tbody>
</table>

**NOTES:**
- All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.5  Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td>FCMOVNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>FCMOVNU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>FUCOMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
</tbody>
</table>

**NOTES:**
- All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD</td>
<td>FSTTP</td>
<td>FST</td>
<td>FSTP</td>
<td>FRSTOR</td>
<td>FSAVE</td>
<td>FSTSW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double-real</td>
<td>integer64</td>
<td>double-real</td>
<td>double-real</td>
<td>98/108bytes</td>
<td>98/108bytes</td>
<td>2 bytes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FFREE</td>
</tr>
<tr>
<td></td>
<td>ST(0)</td>
<td>ST(1)</td>
<td>ST(2)</td>
<td>ST(3)</td>
<td>ST(4)</td>
<td>ST(5)</td>
<td>ST(6)</td>
<td>ST(7)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FST</td>
</tr>
<tr>
<td></td>
<td>ST(0)</td>
<td>ST(1)</td>
<td>ST(2)</td>
<td>ST(3)</td>
<td>ST(4)</td>
<td>ST(5)</td>
<td>ST(6)</td>
<td>ST(7)</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FUCOM</td>
</tr>
<tr>
<td></td>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FIMUL</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FICOM</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FICOMP</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FISUB</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FISUBR</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FIDIV</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
<tr>
<td>FIDIVR</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td>word-integer</td>
<td></td>
</tr>
</tbody>
</table>

### NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

#### A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.
Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

### Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FADDP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
<td></td>
</tr>
<tr>
<td><strong>D</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>E</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
<td></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST(0),ST(0)</td>
<td>ST(1),ST(0)</td>
<td>ST(2),ST(0)</td>
<td>ST(3),ST(0)</td>
<td>ST(4),ST(0)</td>
<td>ST(5),ST(0)</td>
<td>ST(6),ST(0)</td>
<td>ST(7),ST(0)</td>
<td></td>
</tr>
</tbody>
</table>

### Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD</td>
<td>word-integer</td>
<td>FISTTP</td>
<td>word-integer</td>
<td>FIST</td>
<td>word-integer</td>
<td>FBLD</td>
<td>word-integer</td>
<td>FBLD</td>
</tr>
<tr>
<td>FISTTP</td>
<td>word-integer</td>
<td>FIST</td>
<td>word-integer</td>
<td>FISTP</td>
<td>word-integer</td>
<td>FBSTP</td>
<td>word-integer</td>
<td>FBSTP</td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

### A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

### Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

<table>
<thead>
<tr>
<th>nnn Field of ModR/M Byte</th>
<th>000B</th>
<th>001B</th>
<th>010B</th>
<th>011B</th>
<th>100B</th>
<th>101B</th>
<th>110B</th>
<th>111B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILD</td>
<td>word-integer</td>
<td>FISTTP</td>
<td>word-integer</td>
<td>FIST</td>
<td>word-integer</td>
<td>FBLD</td>
<td>word-integer</td>
<td>FBLD</td>
</tr>
<tr>
<td>FISTTP</td>
<td>word-integer</td>
<td>FIST</td>
<td>word-integer</td>
<td>FISTP</td>
<td>word-integer</td>
<td>FBSTP</td>
<td>word-integer</td>
<td>FBSTP</td>
</tr>
</tbody>
</table>

**NOTES:**

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

<table>
<thead>
<tr>
<th>Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>FUCOMIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>ST(0),ST(0)</td>
<td>ST(0),ST(1)</td>
<td>ST(0),ST(2)</td>
<td>ST(0),ST(3)</td>
<td>ST(0),ST(4)</td>
<td>ST(0),ST(5)</td>
<td>ST(0),ST(6)</td>
<td>ST(0),ST(7)</td>
</tr>
</tbody>
</table>

NOTES:
* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.
This page was intentionally left blank.
This appendix provides machine instruction formats and encodings of IA-32 instructions. The first section describes the IA-32 architecture’s machine instruction format. The remaining sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, x87 FPU instructions, and VMX instructions. Those instruction formats also apply to Intel 64 architecture. Instruction formats used in 64-bit mode are provided as supersets of the above.

B.1 MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of:

- an opcode
- a register and/or address mode specifier consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte (if required)
- a displacement and an immediate data field (if required)

![Figure B-1. General Machine Instruction Format](image)

The following sections discuss this format.

B.1.1 Legacy Prefixes

The legacy prefixes noted in Figure B-1 include 66H, 67H, F2H, and F3H. They are optional, except when F2H, F3H, and 66H are used in instruction extensions. Legacy prefixes must be placed before REX prefixes.

B.1.2  REX Prefixes

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.


B.1.3  Opcode Fields

The primary opcode for an instruction is encoded in one to three bytes of the instruction. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed.

Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte following the opcode. This byte, the ModR/M byte, consists of the mod field (2 bits), the reg field (3 bits; this field is sometimes an opcode extension), and the R/M field (3 bits). Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. Possible sizes are 8, 16, or 32 bits. If the instruction specifies an immediate value, the immediate value follows any displacement bytes. The immediate, if specified, is always the last field of the instruction.


B.1.4  Special Fields

Table B-1 lists bit fields that appear in certain instructions, sometimes within the opcode bytes. All of these fields (except the d bit) occur in the general-purpose instruction formats in Table B-13.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Number of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>General-register specifier (see Table B-4 or B-5).</td>
<td>3</td>
</tr>
<tr>
<td>w</td>
<td>Specifies if data is byte or full-sized, where full-sized is 16 or 32 bits (see Table B-6).</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>Specifies sign extension of an immediate field (see Table B-7).</td>
<td>1</td>
</tr>
<tr>
<td>sreg2</td>
<td>Segment register specifier for CS, SS, DS, ES (see Table B-8).</td>
<td>2</td>
</tr>
<tr>
<td>sreg3</td>
<td>Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-8).</td>
<td>3</td>
</tr>
<tr>
<td>eee</td>
<td>Specifies a special-purpose (control or debug) register (see Table B-9).</td>
<td>3</td>
</tr>
<tr>
<td>ttttn</td>
<td>For conditional instructions, specifies a condition asserted or negated (see Table B-12).</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>Specifies direction of data operation (see Table B-11).</td>
<td>1</td>
</tr>
</tbody>
</table>
B.1.4.1  Reg Field (reg) for Non-64-Bit Modes

The reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-3 shows the encoding of the reg field when the w bit is present.

Table B-2. Encoding of reg Field When w Field is Not Present in Instruction

<table>
<thead>
<tr>
<th>reg Field</th>
<th>Register Selected during 16-Bit Data Operations</th>
<th>Register Selected during 32-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>SP</td>
<td>ESP</td>
</tr>
<tr>
<td>101</td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>110</td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>DI</td>
<td>EDI</td>
</tr>
</tbody>
</table>

Table B-3. Encoding of reg Field When w Field is Present in Instruction

<table>
<thead>
<tr>
<th>reg</th>
<th>Function of w Field</th>
<th>When w = 0</th>
<th>When w = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AL</td>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>CX</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>DX</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>BP</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>DI</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reg</th>
<th>Function of w Field</th>
<th>When w = 0</th>
<th>When w = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AL</td>
<td>EAX</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>ECX</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>EDX</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>EBX</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>EDI</td>
<td></td>
</tr>
</tbody>
</table>
B.1.4.2  Reg Field (reg) for 64-Bit Mode

Just like in non-64-bit modes, the reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence of and state of the w bit in an encoding (refer to Section B.1.4.3). Table B-4 shows the encoding of the reg field when the w bit is not present in an encoding; Table B-5 shows the encoding of the reg field when the w bit is present.

Table B-4. Encoding of reg Field When w Field is Not Present in Instruction

<table>
<thead>
<tr>
<th>reg Field</th>
<th>Register Selected during 16-Bit Data Operations</th>
<th>Register Selected during 32-Bit Data Operations</th>
<th>Register Selected during 64-Bit Data Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AX</td>
<td>EAX</td>
<td>RAX</td>
</tr>
<tr>
<td>001</td>
<td>CX</td>
<td>ECX</td>
<td>RCX</td>
</tr>
<tr>
<td>010</td>
<td>DX</td>
<td>EDX</td>
<td>RDX</td>
</tr>
<tr>
<td>011</td>
<td>BX</td>
<td>EBX</td>
<td>RBX</td>
</tr>
<tr>
<td>100</td>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
</tr>
<tr>
<td>101</td>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
</tr>
<tr>
<td>110</td>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
</tr>
<tr>
<td>111</td>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
</tr>
</tbody>
</table>

Table B-5. Encoding of reg Field When w Field is Present in Instruction

<table>
<thead>
<tr>
<th>reg</th>
<th>Function of w Field</th>
<th>Function of w Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>When w = 0</td>
<td>When w = 1</td>
</tr>
<tr>
<td>000</td>
<td>AL</td>
<td>AX</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>CX</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>DX</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>BX</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>SP</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>BP</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>SI</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>DI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reg</th>
<th>Function of w Field</th>
<th>Function of w Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>When w = 0</td>
<td>When w = 1</td>
</tr>
<tr>
<td>000</td>
<td>AL</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>CL</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>DL</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>BL</td>
<td>EBP</td>
</tr>
<tr>
<td>100</td>
<td>AH</td>
<td>ESP</td>
</tr>
<tr>
<td>101</td>
<td>CH</td>
<td>EBX</td>
</tr>
<tr>
<td>110</td>
<td>DH</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>BH</td>
<td>EDI</td>
</tr>
</tbody>
</table>

NOTES:
1. AH, CH, DH, BH can not be encoded when REX prefix is used. Such an expression defaults to the low byte.

B.1.4.3  Encoding of Operand Size (w) Bit

The current operand-size attribute determines whether the processor is performing 16-bit, 32-bit or 64-bit operations. Within the constraints of the current operand-size attribute, the operand-size bit (w) can be used to indicate operations on 8-bit operands or the full operand size specified with the operand-size attribute. Table B-6 shows the encoding of the w bit depending on the current operand-size attribute.

Table B-6. Encoding of Operand Size (w) Bit

<table>
<thead>
<tr>
<th>w Bit</th>
<th>Operand Size When Operand-Size Attribute is 16 Bits</th>
<th>Operand Size When Operand-Size Attribute is 32 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 Bits</td>
<td>8 Bits</td>
</tr>
<tr>
<td>1</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
</tbody>
</table>
B.1.4.4 Sign-Extend (s) Bit

The sign-extend (s) bit occurs in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. See Table B-7.

<table>
<thead>
<tr>
<th>s</th>
<th>Effect on 8-Bit Immediate Data</th>
<th>Effect on 16- or 32-Bit Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Sign-extend to fill 16-bit or 32-bit destination</td>
<td>None</td>
</tr>
</tbody>
</table>

B.1.4.5 Segment Register (sreg) Field

When an instruction operates on a segment register, the reg field in the ModR/M byte is called the sreg field and is used to specify the segment register. Table B-8 shows the encoding of the sreg field. This field is sometimes a 2-bit field (sreg2) and other times a 3-bit field (sreg3).

<table>
<thead>
<tr>
<th>2-Bit sreg2 Field</th>
<th>Segment Register Selected</th>
<th>3-Bit sreg3 Field</th>
<th>Segment Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ES</td>
<td>000</td>
<td>ES</td>
</tr>
<tr>
<td>01</td>
<td>CS</td>
<td>001</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>SS</td>
<td>010</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>DS</td>
<td>011</td>
<td>DS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>FS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td>GS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>Reserved¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

NOTES:
1. Do not use reserved encodings.

B.1.4.6 Special-Purpose Register (eee) Field

When control or debug registers are referenced in an instruction they are encoded in the eee field, located in bits 5 though 3 of the ModR/M byte (an alternate encoding of the sreg field). See Table B-9.

<table>
<thead>
<tr>
<th>eee</th>
<th>Control Register</th>
<th>Debug Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Reserved¹</td>
<td>DR0</td>
</tr>
<tr>
<td>001</td>
<td>Reserved¹</td>
<td>DR1</td>
</tr>
<tr>
<td>010</td>
<td>CR2</td>
<td>DR2</td>
</tr>
<tr>
<td>011</td>
<td>CR3</td>
<td>DR3</td>
</tr>
<tr>
<td>100</td>
<td>CR4</td>
<td>Reserved</td>
</tr>
<tr>
<td>101</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>110</td>
<td>Reserved</td>
<td>DR6</td>
</tr>
<tr>
<td>111</td>
<td>Reserved</td>
<td>DR7</td>
</tr>
</tbody>
</table>

NOTES:
1. Do not use reserved encodings.
B.1.4.7 Condition Test (tttn) Field

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) is encoded for the condition being tested. The ttt part of the field gives the condition to test and the n part indicates whether to use the condition \( n = 0 \) or its negation \( n = 1 \).

- For 1-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the opcode byte.
- For 2-byte primary opcodes, the tttn field is located in bits 3, 2, 1, and 0 of the second opcode byte.

Table B-10 shows the encoding of the tttn field.

<table>
<thead>
<tr>
<th>t t t n</th>
<th>Mnemonic</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>O</td>
<td>Overflow</td>
</tr>
<tr>
<td>0001</td>
<td>NO</td>
<td>No overflow</td>
</tr>
<tr>
<td>0010</td>
<td>B, NAE</td>
<td>Below, Not above or equal</td>
</tr>
<tr>
<td>0011</td>
<td>NB, AE</td>
<td>Not below, Above or equal</td>
</tr>
<tr>
<td>0100</td>
<td>E, Z</td>
<td>Equal, Zero</td>
</tr>
<tr>
<td>0101</td>
<td>NE, NZ</td>
<td>Not equal, Not zero</td>
</tr>
<tr>
<td>0110</td>
<td>BE, NA</td>
<td>Below or equal, Not above</td>
</tr>
<tr>
<td>0111</td>
<td>NBE, A</td>
<td>Not below or equal, Above</td>
</tr>
<tr>
<td>1000</td>
<td>S</td>
<td>Sign</td>
</tr>
<tr>
<td>1001</td>
<td>NS</td>
<td>Not sign</td>
</tr>
<tr>
<td>1010</td>
<td>P, PE</td>
<td>Parity, Parity Even</td>
</tr>
<tr>
<td>1011</td>
<td>NP, PO</td>
<td>Not parity, Parity Odd</td>
</tr>
<tr>
<td>1100</td>
<td>L, NGE</td>
<td>Less than, Not greater than or equal to</td>
</tr>
<tr>
<td>1101</td>
<td>NL, GE</td>
<td>Not less than, Greater than or equal to</td>
</tr>
<tr>
<td>1110</td>
<td>LE, NG</td>
<td>Less than or equal to, Not greater than</td>
</tr>
<tr>
<td>1111</td>
<td>NLE, G</td>
<td>Not less than or equal to, Greater than</td>
</tr>
</tbody>
</table>

B.1.4.8 Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. See Table B-11.

- When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. Note that this bit does not appear as the symbol "d" in Table B-13; the actual encoding of the bit as 1 or 0 is given.
- When used for floating-point instructions (in Table B-16), the d bit is shown as bit 2 of the first byte of the primary opcode.

<table>
<thead>
<tr>
<th>d</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reg Field</td>
<td>ModR/M or SIB Byte</td>
</tr>
<tr>
<td>1</td>
<td>ModR/M or SIB Byte</td>
<td>reg Field</td>
</tr>
</tbody>
</table>

B.1.5 Other Notes

Table B-12 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.
B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES

Table B-13 shows machine instruction formats and encodings for general purpose instructions in non-64-bit modes.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA – ASCII Adjust after Addition</td>
<td>0011 0111</td>
</tr>
<tr>
<td>AAD – ASCII Adjust AX before Division</td>
<td>1101 0101 : 0000 1010</td>
</tr>
<tr>
<td>AAM – ASCII Adjust AX after Multiply</td>
<td>1101 0100 : 0000 1010</td>
</tr>
<tr>
<td>AAS – ASCII Adjust AL after Subtraction</td>
<td>0011 1111</td>
</tr>
<tr>
<td>ADC – ADD with Carry</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0001 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0001 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0001 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0001 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 010 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 010 r/m : immediate data</td>
</tr>
<tr>
<td>ADD – Add</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0000 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td>AND – Logical AND</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0010 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0010 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0010 001w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0010 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 100 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 100 r/m : immediate data</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARPL – Adjust RPL Field of Selector</strong></td>
<td></td>
</tr>
<tr>
<td>from register</td>
<td>0110 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from memory</td>
<td>0110 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BOUND – Check Array Against Bounds</strong></td>
<td>0110 0010 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BSF – Bit Scan Forward</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1100 : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 1100 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BSR – Bit Scan Reverse</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1101 : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 1101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BSWAP – Byte Swap</strong></td>
<td>0000 1111 : 1100 1 reg</td>
</tr>
<tr>
<td><strong>BT – Bit Test</strong></td>
<td></td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : 11 100 reg : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1010 : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1010 0011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1010 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTC – Bit Test and Complement</strong></td>
<td></td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : 11 111 reg : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1010 : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTR – Bit Test and Reset</strong></td>
<td></td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : 11 110 reg : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1010 : mod 110 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 0011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1011 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>BTS – Bit Test and Set</strong></td>
<td></td>
</tr>
<tr>
<td>register, immediate</td>
<td>0000 1111 : 1011 1010 : 11 101 reg : imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0000 1111 : 1011 1010 : mod 101 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1010 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1010 1011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>CALL – Call Procedure (in same segment)</strong></td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td>1110 1000 : full displacement</td>
</tr>
<tr>
<td>register indirect</td>
<td>1111 1111 : 11 010 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>1111 1111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>CALL – Call Procedure (in other segment)</strong></td>
<td></td>
</tr>
<tr>
<td>direct</td>
<td>1001 1010 : unsigned full offset, selector</td>
</tr>
<tr>
<td>indirect</td>
<td>1111 1111 : mod 011 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>CBW – Convert Byte to Word</td>
<td>1001 1000</td>
</tr>
<tr>
<td>CDQ - Convert Doubleword to Qword</td>
<td>1001 1001</td>
</tr>
<tr>
<td>CLC – Clear Carry Flag</td>
<td>1111 1000</td>
</tr>
<tr>
<td>CLD - Clear Direction Flag</td>
<td>1111 1100</td>
</tr>
<tr>
<td>CLI – Clear Interrupt Flag</td>
<td>1111 1010</td>
</tr>
<tr>
<td>CLTS – Clear Task-Switched Flag in CR0</td>
<td>0000 1111 : 0000 0110</td>
</tr>
<tr>
<td>CMC – Complement Carry Flag</td>
<td>1111 0101</td>
</tr>
</tbody>
</table>

**CMP – Compare Two Operands**

- register1 with register2 | 0011 100w : 11 reg1 reg2 |
- register2 with register1 | 0011 101w : 11 reg1 reg2 |
- memory with register | 0011 100w : mod reg r/m |
- register with memory | 0011 101w : mod reg r/m |
- immediate with register | 1000 00sw : 11 111 reg : immediate data |
- immediate with AL, AX, or EAX | 0011 110w : immediate data |
- immediate with memory | 1000 00sw : mod 111 r/m : immediate data |

**CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands** | 1010 011w |

**CMPXCHG – Compare and Exchange**

- register1, register2 | 0000 1111 : 1011 000w : 11 reg2 reg1 |
- memory, register | 0000 1111 : 1011 000w : mod reg r/m |

**CPUID – CPU Identification** | 0000 1111 : 1010 0010 |

**CWD – Convert Word to Doubleword** | 1001 1001 |

**CWDE – Convert Word to Doubleword** | 1001 1000 |

**DAA – Decimal Adjust AL after Addition** | 0010 0111 |

**DAS – Decimal Adjust AL after Subtraction** | 0010 1111 |

**DEC – Decrement by 1**

- register | 1111 111w : 11 001 reg |
- register (alternate encoding) | 0100 1 reg |
- memory | 1111 111w : mod 001 r/m |

**DIV – Unsigned Divide**

- AL, AX, or EAX by register | 1111 011w : 11 110 reg |
- AL, AX, or EAX by memory | 1111 011w : mod 110 r/m |

**HLT – Halt** | 1111 0100 |

**IDIV – Signed Divide**

- AL, AX, or EAX by register | 1111 011w : 11 111 reg |
- AL, AX, or EAX by memory | 1111 011w : mod 111 r/m |
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IMUL – Signed Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>1111 011w : 11 101 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>1111 011w : mod 101 reg</td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0000 1111 : 1010 1111 : 11 : reg1 reg2</td>
</tr>
<tr>
<td>register with memory</td>
<td>0000 1111 : 1010 1111 : mod reg r/m</td>
</tr>
<tr>
<td>register1 with immediate to register2</td>
<td>0110 10s1 : 11 reg1 reg2 : immediate data</td>
</tr>
<tr>
<td>memory with immediate to register</td>
<td>0110 10s1 : mod reg r/m : immediate data</td>
</tr>
<tr>
<td><strong>IN – Input From Port</strong></td>
<td></td>
</tr>
<tr>
<td>fixed port</td>
<td>1110 010w : port number</td>
</tr>
<tr>
<td>variable port</td>
<td>1110 110w</td>
</tr>
<tr>
<td><strong>INC – Increment by 1</strong></td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>1111 111w : 11 000 reg</td>
</tr>
<tr>
<td>reg (alternate encoding)</td>
<td>0100 0 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 111w : mod 000 r/m</td>
</tr>
<tr>
<td><strong>INS – Input from DX Port</strong></td>
<td>0110 110w</td>
</tr>
<tr>
<td><strong>INT n – Interrupt Type n</strong></td>
<td>1100 1101 : type</td>
</tr>
<tr>
<td><strong>INT – Single-Step Interrupt 3</strong></td>
<td>1100 1100</td>
</tr>
<tr>
<td><strong>INTO – Interrupt 4 on Overflow</strong></td>
<td>1100 1110</td>
</tr>
<tr>
<td><strong>INVD – Invalidate Cache</strong></td>
<td>0000 1111 : 0000 1000</td>
</tr>
<tr>
<td><strong>INVLPG – Invalidate TLB Entry</strong></td>
<td>0000 1111 : 0000 0001 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>INVP CID – Invalidate Process-Context Identifier</strong></td>
<td>0110 0110:0000 1111:0011 1000:1000 0010: mod reg r/m</td>
</tr>
<tr>
<td><strong>IRET/IRETD – Interrupt Return</strong></td>
<td>1100 1111</td>
</tr>
<tr>
<td><strong>Jcc – Jump if Condition is Met</strong></td>
<td></td>
</tr>
<tr>
<td>8-bit displacement</td>
<td>0111 tttn : 8-bit displacement</td>
</tr>
<tr>
<td>full displacement</td>
<td>0000 1111 : 1000 tttn : full displacement</td>
</tr>
<tr>
<td><strong>JCXZ/JECXZ – Jump on CX/ECX Zero</strong></td>
<td></td>
</tr>
<tr>
<td>Address-size prefix differentiates JCXZ and JECXZ</td>
<td>1110 0011 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>JMP – Unconditional Jump (to same segment)</strong></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>1110 1011 : 8-bit displacement</td>
</tr>
<tr>
<td>direct</td>
<td>1110 1001 : full displacement</td>
</tr>
<tr>
<td>register indirect</td>
<td>1111 1111 : 11 100 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>1111 1111 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>JMP – Unconditional Jump (to other segment)</strong></td>
<td></td>
</tr>
<tr>
<td>direct intersegment</td>
<td>1110 1010 : unsigned full offset, selector</td>
</tr>
<tr>
<td>indirect intersegment</td>
<td>1111 1111 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>LAHF – Load Flags into AHRegister</strong></td>
<td>1001 1111</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>LAR – Load Access Rights Byte</strong></td>
<td>from register: 0000 1111 : 0000 0010 : 11 reg1 reg2</td>
</tr>
<tr>
<td></td>
<td>from memory: 0000 1111 : 0000 0010 : mod reg r/m</td>
</tr>
<tr>
<td><strong>LDS – Load Pointer to DS</strong></td>
<td>1100 0101 : modA,B reg r/m</td>
</tr>
<tr>
<td><strong>LEA – Load Effective Address</strong></td>
<td>1000 1101 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LEAVE – High Level Procedure Exit</strong></td>
<td>1100 1001</td>
</tr>
<tr>
<td><strong>LES – Load Pointer to ES</strong></td>
<td>1100 0100 : modA,B reg r/m</td>
</tr>
<tr>
<td><strong>LFS – Load Pointer to FS</strong></td>
<td>0000 1111 : 1011 0100 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LGDT – Load Global Descriptor Table Register</strong></td>
<td>0000 1111 : 0000 0001 : modA 010 r/m</td>
</tr>
<tr>
<td><strong>LGS – Load Pointer to GS</strong></td>
<td>0000 1111 : 1011 0101 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LIDT – Load Interrupt Descriptor Table Register</strong></td>
<td>0000 1111 : 0000 0001 : modA 011 r/m</td>
</tr>
<tr>
<td><strong>LLDT – Load Local Descriptor Table Register</strong></td>
<td>from register: 0000 1111 : 0000 0010 : 11 010 reg</td>
</tr>
<tr>
<td></td>
<td>from memory: 0000 1111 : 0000 0000 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>LMSW – Load Machine Status Word</strong></td>
<td>from register: 0000 1111 : 0000 0001 : 11 110 reg</td>
</tr>
<tr>
<td></td>
<td>from memory: 0000 1111 : 0000 0001 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>LOCK – Assert LOCK# Signal Prefix</strong></td>
<td>1111 0000</td>
</tr>
<tr>
<td><strong>LODS/LODSB/LODSW/LODSD – Load String Operand</strong></td>
<td>1010 110w</td>
</tr>
<tr>
<td><strong>LOOP – Loop Count</strong></td>
<td>1110 0010 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOPZ/LOOPE – Loop Count while Zero/Equal</strong></td>
<td>1110 0001 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>LOOPNZ/LOOPNE – Loop Count while not Zero/Equal</strong></td>
<td>1110 0000 : 8-bit displacement</td>
</tr>
<tr>
<td><strong>LSL – Load Segment Limit</strong></td>
<td>from register: 0000 1111 : 0000 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td></td>
<td>from memory: 0000 1111 : 0000 0011 : mod reg r/m</td>
</tr>
<tr>
<td><strong>LSS – Load Pointer to SS</strong></td>
<td>0000 1111 : 1011 0010 : modA reg r/m</td>
</tr>
<tr>
<td><strong>LTR – Load Task Register</strong></td>
<td>from register: 0000 1111 : 0000 0000 : 11 011 reg</td>
</tr>
<tr>
<td></td>
<td>from memory: 0000 1111 : 0000 0000 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>MOV – Move Data</strong></td>
<td>register1 to register2: 1000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td></td>
<td>register2 to register1: 1000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td></td>
<td>memory to reg: 1000 101w : mod reg r/m</td>
</tr>
<tr>
<td></td>
<td>reg to memory: 1000 100w : mod reg r/m</td>
</tr>
<tr>
<td></td>
<td>immediate to register: 1100 011w : 11 000 reg : immediate data</td>
</tr>
<tr>
<td></td>
<td>immediate to register (alternate encoding): 1011 w reg : immediate data</td>
</tr>
<tr>
<td></td>
<td>immediate to memory: 1100 011w : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td></td>
<td>memory to AL, AX, or EAX: 1010 000w : full displacement</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL, AX, or EAX to memory</td>
<td>1010 001w : full displacement</td>
</tr>
<tr>
<td><strong>MOV - Move to/from Control Registers</strong></td>
<td></td>
</tr>
<tr>
<td>CR0 from register</td>
<td>0000 1111 : 0010 0010 : -- 000 reg</td>
</tr>
<tr>
<td>CR2 from register</td>
<td>0000 1111 : 0010 0010 : -- 010reg</td>
</tr>
<tr>
<td>CR3 from register</td>
<td>0000 1111 : 0010 0010 : -- 011 reg</td>
</tr>
<tr>
<td>CR4 from register</td>
<td>0000 1111 : 0010 0010 : -- 100 reg</td>
</tr>
<tr>
<td>register from CR0-CR4</td>
<td>0000 1111 : 0010 0000 : -- eee reg</td>
</tr>
<tr>
<td><strong>MOV - Move to/from Debug Registers</strong></td>
<td></td>
</tr>
<tr>
<td>DR0-DR3 from register</td>
<td>0000 1111 : 0010 0011 : -- eee reg</td>
</tr>
<tr>
<td>DR4-DR5 from register</td>
<td>0000 1111 : 0010 0011 : -- eee reg</td>
</tr>
<tr>
<td>DR6-DR7 from register</td>
<td>0000 1111 : 0010 0011 : -- eee reg</td>
</tr>
<tr>
<td>register from DR6-DR7</td>
<td>0000 1111 : 0010 0001 : -- eee reg</td>
</tr>
<tr>
<td>register from DR4-DR5</td>
<td>0000 1111 : 0010 0001 : -- eee reg</td>
</tr>
<tr>
<td>register from DR0-DR3</td>
<td>0000 1111 : 0010 0001 : -- eee reg</td>
</tr>
<tr>
<td><strong>MOV - Move to/from Segment Registers</strong></td>
<td></td>
</tr>
<tr>
<td>register to segment register</td>
<td>1000 1110 : 11 sreg3 reg</td>
</tr>
<tr>
<td>register to SS</td>
<td>1000 1110 : 11 sreg3 reg</td>
</tr>
<tr>
<td>memory to segment reg</td>
<td>1000 1110 : mod sreg3 r/m</td>
</tr>
<tr>
<td>memory to SS</td>
<td>1000 1110 : mod sreg3 r/m</td>
</tr>
<tr>
<td>segment register to register</td>
<td>1000 1100 : 11 sreg3 reg</td>
</tr>
<tr>
<td>segment register to memory</td>
<td>1000 1100 : mod sreg3 r/m</td>
</tr>
<tr>
<td><strong>MOVBE - Move data after swapping bytes</strong></td>
<td></td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111 : 0011 1011 0000 : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 1111 : 0011 1000 : mod reg r/m</td>
</tr>
<tr>
<td><strong>MOVMSB/MOVSD/MOVSD - Move Data from String to String</strong></td>
<td>1010 010w</td>
</tr>
<tr>
<td><strong>MOVSX - Move with Sign-Extend</strong></td>
<td></td>
</tr>
<tr>
<td>memory to reg</td>
<td>0000 1111 : 1011 111w : mod reg r/m</td>
</tr>
<tr>
<td><strong>MOVZX - Move with Zero-Extend</strong></td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 1111 : 1011 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111 : 1011 011w : mod reg r/m</td>
</tr>
<tr>
<td><strong>MUL - Unsigned Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>1111 011w : 11 100 reg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>1111 011w : mod 100 r/m</td>
</tr>
<tr>
<td><strong>NEG - Two's Complement Negation</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>1111 011w : 11 011 reg</td>
</tr>
<tr>
<td>memory</td>
<td>1111 011w : mod 011 r/m</td>
</tr>
<tr>
<td><strong>NOP - No Operation</strong></td>
<td>1001 0000</td>
</tr>
</tbody>
</table>
### Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
</table>
| **NOP – Multi-byte No Operation**

| register | 0000 1111 0001 1111 : 11 000 reg |
| memory | 0000 1111 0001 1111 : mod 000 r/m |

| **NOT – One's Complement Negation**

| register | 1111 011w : 11 010 reg |
| memory | 1111 011w : mod 010 r/m |

| **OR – Logical Inclusive OR**

| register1 to register2 | 0000 100w : 11 reg1 reg2 |
| register2 to register1 | 0000 101w : 11 reg1 reg2 |
| memory to register | 0000 101w : mod reg r/m |
| register to memory | 0000 100w : mod reg r/m |
| immediate to register | 1000 00sw : 11 001 reg : immediate data |
| immediate to AL, AX, or EAX | 0000 110w : immediate data |
| immediate to memory | 1000 00sw : mod 001 r/m : immediate data |

| **OUT – Output to Port**

| fixed port | 1110 011w : port number |
| variable port | 1110 111w |

| **OUTS – Output to DX Port** | 0110 111w |

| **POP – Pop a Word from the Stack**

| register | 1000 1111 : 11 000 reg |
| register (alternate encoding) | 0101 1 110 reg |
| memory | 1000 1111 : mod 000 r/m |

| **POP – Pop a Segment Register from the Stack** (Note: CS cannot be sreg2 in this usage.)

| segment register DS, ES | 000 sreg2 111 |
| segment register SS | 000 sreg2 111 |
| segment register FS, GS | 0000 1111 : 10 sreg3 001 |

| **POPA/POPAD – Pop All General Registers** | 0110 0001 |

| **POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register** | 1001 1101 |

| **PUSH – Push Operand onto the Stack**

| register | 1111 1111 : 11 110 reg |
| register (alternate encoding) | 0101 0 110 reg |
| memory | 1111 1111 : mod 110 r/m |
| immediate | 0110 10s0 : immediate data |

| **PUSH – Push Segment Register onto the Stack**

| segment register CS,DS,ES,SS | 000 sreg2 110 |
| segment register FS,GS | 0000 1111 : 10 sreg3 000 |

| **PUSHA/PUSHAD – Push All General Registers** | 0110 0000 |
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSHF/PUSHFD – Push Flags Register onto the Stack</td>
<td>1001 1100</td>
</tr>
</tbody>
</table>

**RCL – Rotate thru Carry Left**
- register by 1: 1101 000w: 11 010 \_reg
- memory by 1: 1101 000w: mod 010 \_r/m
- register by CL: 1101 001w: 11 010 \_reg
- memory by CL: 1101 001w: mod 010 \_r/m
- register by immediate count: 1100 000w: 11 010 \_reg: imm8 \_data
- memory by immediate count: 1100 000w: mod 010 \_r/m: imm8 \_data

**RCR – Rotate thru Carry Right**
- register by 1: 1101 000w: 11 011 \_reg
- memory by 1: 1101 000w: mod 011 \_r/m
- register by CL: 1101 001w: 11 011 \_reg
- memory by CL: 1101 001w: mod 011 \_r/m
- register by immediate count: 1100 000w: 11 011 \_reg: imm8 \_data
- memory by immediate count: 1100 000w: mod 011 \_r/m: imm8 \_data

**RDMSR – Read from Model-Specific Register**
0000 1111 : 0011 0010

**RDPMC – Read Performance Monitoring Counters**
0000 1111 : 0011 0011

**RDTSC – Read Time-Stamp Counter**
0000 1111 : 0011 0001

**RDTSCP – Read Time-Stamp Counter and Processor ID**
0000 1111 : 0000 0001: 1111 1001

**REP INS – Input String**
1111 0011 : 0110 110w

**REP LODS – Load String**
1111 0011 : 1010 110w

**REP MOVS – Move String**
1111 0011 : 1010 010w

**REP OUTS – Output String**
1111 0011 : 0110 111w

**REP STOS – Store String**
1111 0011 : 0110 1101w

**REPE CMPS – Compare String**
1111 0011 : 1010 011w

**REPE SCAS – Scan String**
1111 0011 : 1010 111w

**REPNE CMPS – Compare String**
1111 0010 : 1010 011w

**REPNE SCAS – Scan String**
1111 0010 : 1010 111w

**RET – Return from Procedure (to same segment)**
- no argument: 1100 0011
- adding immediate to SP: 1100 0010: 16-bit displacement

**RET – Return from Procedure (to other segment)**
- intersegment: 1100 1011
- adding immediate to SP: 1100 1010: 16-bit displacement
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROL - Rotate Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 000 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 000 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 000 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 000 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 000 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 000 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>ROR - Rotate Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 001 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 001 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 001 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 001 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 001 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 001 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>RSM - Resume from System Management Mode</strong></td>
<td>0000 1111 : 1010 1010</td>
</tr>
<tr>
<td><strong>SAHF - Store AH into Flags</strong></td>
<td>1001 1110</td>
</tr>
<tr>
<td><strong>SAL - Shift Arithmetic Left</strong></td>
<td>same instruction as SHL</td>
</tr>
<tr>
<td><strong>SAR - Shift Arithmetic Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101 000w : 11 111 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101 000w : mod 111 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101 001w : 11 111 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101 001w : mod 111 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 111 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 111 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SBB - Integer Subtraction with Borrow</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0001 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0001 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0001 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0001 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 011 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 011 r/m : immediate data</td>
</tr>
<tr>
<td><strong>SCAS/SCASB/SCASW/SCASD - Scan String</strong></td>
<td>1010 111w</td>
</tr>
<tr>
<td><strong>SETcc - Byte Set on Condition</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 1001 tttt : 11 000 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 1001 tttt : mod 000 r/m</td>
</tr>
<tr>
<td><strong>SGDT - Store Global Descriptor Table Register</strong></td>
<td>0000 1111 : 0000 0001 : modA 000 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td><strong>SHL – Shift Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101000w : 11 100 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101000w : mod 100 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101001w : 11 100 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101001w : mod 100 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100000w : 11 100 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100000w : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SHLD – Double Precision Shift Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>00001111 : 1010 0100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>00001111 : 1010 0100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>00001111 : 1010 0101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>00001111 : 1010 0101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>SHR – Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>1101000w : 11 101 reg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>1101000w : mod 101 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>1101001w : 11 101 reg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>1101001w : mod 101 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100000w : 11 101 reg : imm8 data</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100000w : mod 101 r/m : imm8 data</td>
</tr>
<tr>
<td><strong>SHRD – Double Precision Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>00001111 : 1010 1100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>00001111 : 1010 1100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>00001111 : 1010 1101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>00001111 : 1010 1101 : mod reg r/m</td>
</tr>
<tr>
<td><strong>SIDT – Store Interrupt Descriptor Table Register</strong></td>
<td>00001111 : 0000 0001 : modΑ 001 r/m</td>
</tr>
<tr>
<td><strong>SLDT – Store Local Descriptor Table Register</strong></td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>00001111 : 0000 0000 : 11 000 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>00001111 : 0000 0000 : mod 000 r/m</td>
</tr>
<tr>
<td><strong>SMSW – Store Machine Status Word</strong></td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>00001111 : 0000 0001 : 11 100 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>00001111 : 0000 0001 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>STC – Set Carry Flag</strong></td>
<td>1111 1001</td>
</tr>
<tr>
<td><strong>STD – Set Direction Flag</strong></td>
<td>1111 1101</td>
</tr>
<tr>
<td><strong>STI – Set Interrupt Flag</strong></td>
<td>1111 1011</td>
</tr>
<tr>
<td><strong>STOS/STOSB/STOSw/STOSD – Store String Data</strong></td>
<td>1010 101w</td>
</tr>
<tr>
<td><strong>STR – Store Task Register</strong></td>
<td></td>
</tr>
<tr>
<td>to register</td>
<td>00001111 : 0000 0000 : 11 001 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>00001111 : 0000 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>SUB – Integer Subtraction</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0010 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0010 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0010 101w : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0010 100w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 101 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 110w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 101 r/m : immediate data</td>
</tr>
<tr>
<td><strong>TEST – Logical Compare</strong></td>
<td></td>
</tr>
<tr>
<td>register1 and register2</td>
<td>1000 010w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory and register</td>
<td>1000 010w : mod reg r/m</td>
</tr>
<tr>
<td>immediate and register</td>
<td>1111 011w : 11 000 reg : immediate data</td>
</tr>
<tr>
<td>immediate and AL, AX, or EAX</td>
<td>1010 100w : immediate data</td>
</tr>
<tr>
<td>immediate and memory</td>
<td>1111 011w : mod 000 r/m : immediate data</td>
</tr>
<tr>
<td><strong>UD0 – Undefined instruction</strong></td>
<td>0000 1111 : 1111 1111</td>
</tr>
<tr>
<td><strong>UD1 – Undefined instruction</strong></td>
<td>0000 1111 : 0000 1011</td>
</tr>
<tr>
<td><strong>UD2 – Undefined instruction</strong></td>
<td>0000 FFFF : 0000 1011</td>
</tr>
<tr>
<td><strong>VERR – Verify a Segment for Reading</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 0000 0000 : 11 100 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 0000 0000 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>VERW – Verify a Segment for Writing</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0000 1111 : 0000 0000 : 11 101 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0000 1111 : 0000 0000 : mod 101 r/m</td>
</tr>
<tr>
<td><strong>WAIT – Wait</strong></td>
<td>1001 1011</td>
</tr>
<tr>
<td><strong>WBINVD – Writeback and Invalidate Data Cache</strong></td>
<td>0000 1111 : 0000 1001</td>
</tr>
<tr>
<td><strong>WRMSR – Write to Model-Specific Register</strong></td>
<td>0000 1111 : 0011 0000</td>
</tr>
<tr>
<td><strong>XADD – Exchange and Add</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1100 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0000 1111 : 1100 000w : mod reg r/m</td>
</tr>
<tr>
<td><strong>XCHG – Exchange Register/Memory with Register</strong></td>
<td></td>
</tr>
<tr>
<td>register1 with register2</td>
<td>1000 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>AX or EAX with reg</td>
<td>1001 0 reg</td>
</tr>
<tr>
<td>memory with reg</td>
<td>1000 011w : mod reg r/m</td>
</tr>
<tr>
<td><strong>XLAT/XLATB – Table Look-up Translation</strong></td>
<td>1101 0111</td>
</tr>
<tr>
<td><strong>XOR – Logical Exclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0011 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0011 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0011 001w : mod reg r/m</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register to memory</td>
<td>0011 000w : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 110 reg : immediate data</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0011 010w : immediate data</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 110 r/m : immediate data</td>
</tr>
</tbody>
</table>

Prefix Bytes

<table>
<thead>
<tr>
<th>Address size</th>
<th>0110 0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK</td>
<td>1111 0000</td>
</tr>
<tr>
<td>Operand size</td>
<td>0110 0110</td>
</tr>
<tr>
<td>CS segment override</td>
<td>0010 1110</td>
</tr>
<tr>
<td>DS segment override</td>
<td>0011 1110</td>
</tr>
<tr>
<td>ES segment override</td>
<td>0010 0110</td>
</tr>
<tr>
<td>FS segment override</td>
<td>0110 0100</td>
</tr>
<tr>
<td>GS segment override</td>
<td>0110 0101</td>
</tr>
<tr>
<td>SS segment override</td>
<td>0011 0110</td>
</tr>
</tbody>
</table>

NOTES:
1. The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation.

B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode

Table B-15 shows machine instruction formats and encodings for general purpose instructions in 64-bit mode.

Table B-14. Special Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>If the value of REX.W. is 1, it overrides the presence of 66H.</td>
</tr>
<tr>
<td>w</td>
<td>The value of bit W. in REX is has no effect.</td>
</tr>
</tbody>
</table>

Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC – ADD with Carry</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B : 0001 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B : 0001 0001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0001 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B : 0001 0011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0001 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory to qwordregister</td>
<td>0100 1RXB : 0001 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0001 000w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory</td>
<td>0100 1RXB : 0001 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 000B : 1000 00sw : 11 010 reg : immediate</td>
</tr>
<tr>
<td>immediate to qwordregister</td>
<td>0100 100B : 1000 0001 : 11 010 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate to qwordregister</td>
<td>0100 1R0B : 1000 0011 : 11 010 qwordreg : imm8</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0001 010w : immediate data</td>
</tr>
<tr>
<td>immediate to RAX</td>
<td>0100 1000 : 0000 0101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1000 00sw : mod 010 r/m : immediate</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB : 1000 0001 : mod 010 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB : 1000 0031 : mod 010 r/m : imm8</td>
</tr>
<tr>
<td>ADD – Add</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B : 0000 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0000 0000 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0000 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0000 0010 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB : 0000 0000 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0000 000w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB : 0000 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 0000B : 1000 00sw : 10 00 r/m : immediate data</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B : 1000 0001 : 11 qwordreg : imm</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 010w : immediate8</td>
</tr>
<tr>
<td>immediate to RAX</td>
<td>0100 1000 : 0000 0101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1000 00sw : mod 000 r/m : immediate</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB : 1000 0001 : mod 010 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB : 1000 0011 : mod 010 r/m : imm8</td>
</tr>
<tr>
<td>AND – Logical AND</td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B 0010 000w : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0010 0001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B 0010 001w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0100 1R0B 0010 0011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0010 001w : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB : 0010 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0010 000w : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB : 0010 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 0000B : 1000 00sw : 11 00 reg : immediate</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B : 1000 0001 : 11 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0010 010w : immediate</td>
</tr>
<tr>
<td>immediate32 to RAX</td>
<td>0100 1000 0010 1001 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1000 00sw : mod 100 r/m : immediate</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB : 1000 0001 : mod 100 r/m : immediate32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB : 1000 0011 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td>BSF – Bit Scan Forward</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 1100 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 1100 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 1100 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 1100 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BSR – Bit Scan Reverse**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 1101 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 1101 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 1101 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 1101 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BSWAP – Byte Swap**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111 : 1100 1 reg</td>
<td>0100 100B 0000 1111 : 1100 1 qwordreg</td>
</tr>
</tbody>
</table>

**BT – Bit Test**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 100 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 1111 : 1011 1010 : 11 100 qwordreg: imm8 data</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 100 r/m : imm8 data</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1010 0011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1010 0011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, reg</td>
<td>0100 0RXB 0000 1111 : 1100 0011 : mod reg r/m</td>
</tr>
<tr>
<td>memory, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1100 0011 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BTC – Bit Test and Complement**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 111 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 1111 : 1011 1010 : 11 111 qwordreg: imm8</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 1011 : 11 reg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 1011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 1011 : mod reg r/m</td>
</tr>
<tr>
<td>memory, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1011 1011 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**BTR – Bit Test and Reset**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
<td>0100 000B 0000 1111 : 1011 1010 : 11 110 reg: imm8</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
<td>0100 100B 1111 : 1011 1010 : 11 110 qwordreg: imm8</td>
</tr>
<tr>
<td>memory, immediate</td>
<td>0100 00XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8</td>
</tr>
<tr>
<td>memory64, immediate8</td>
<td>0100 10XB 0000 1111 : 1011 1010 : mod 110 r/m : imm8</td>
</tr>
<tr>
<td>register1, register2</td>
<td>0100 0R0B 0000 1111 : 1011 0011 : 11 reg2 reg1</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 1R0B 0000 1111 : 1011 0011 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0100 0RXB 0000 1111 : 1011 0011 : mod reg r/m</td>
</tr>
<tr>
<td>memory64, qwordreg</td>
<td>0100 1RXB 0000 1111 : 1011 0011 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>BTS – Bit Test and Set</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>register, immediate</td>
</tr>
<tr>
<td>qwordregister, immediate8</td>
</tr>
<tr>
<td>memory, immediate</td>
</tr>
<tr>
<td>memory64, immediate8</td>
</tr>
<tr>
<td>register1, register2</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
</tr>
<tr>
<td>memory, register</td>
</tr>
<tr>
<td>memory64, qwordreg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CALL – Call Procedure (in same segment)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>direct</td>
</tr>
<tr>
<td>register indirect</td>
</tr>
<tr>
<td>memory indirect</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CALL – Call Procedure (in other segment)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>indirect</td>
</tr>
<tr>
<td>indirect</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CBW – Convert Byte to Word</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CDQ – Convert Doubleword to Qword+</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 1001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CDQE – RAX, Sign-Extend of EAX</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 1000 1001 1001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CLC – Clear Carry Flag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CLD – Clear Direction Flag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CLI – Clear Interrupt Flag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CLTS – Clear Task-Switched Flag in CR0</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1111 : 0000 0110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CMC – Complement Carry Flag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CMP – Compare Two Operands</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 with register2</td>
</tr>
<tr>
<td>qwordregister1 with qwordregister2</td>
</tr>
<tr>
<td>register2 with register1</td>
</tr>
<tr>
<td>qwordregister2 with qwordregister1</td>
</tr>
<tr>
<td>memory with register</td>
</tr>
<tr>
<td>memory64 with qwordregister</td>
</tr>
<tr>
<td>register with memory</td>
</tr>
<tr>
<td>qwordregister with memory64</td>
</tr>
<tr>
<td>immediate with register</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate32 with qwordregister</td>
<td>0100 100B 1000 0001 : 11 111 qwordreg : imm64</td>
</tr>
<tr>
<td>immediate with AL, AX, or EAX</td>
<td>0011 110w : imm</td>
</tr>
<tr>
<td>immediate32 with RAX</td>
<td>0100 1000 0011 1101 : imm32</td>
</tr>
<tr>
<td>immediate with memory</td>
<td>0100 00XB 1000 00sw : mod 111 r/m : imm</td>
</tr>
<tr>
<td>immediate32 with memory64</td>
<td>0100 1RXB 1000 0001 : mod 111 r/m : imm64</td>
</tr>
<tr>
<td>immediate8 with memory64</td>
<td>0100 1RXB 1000 0011 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td><strong>CMPS/CMPSB/CMPSW/CMPSD/CMPSQ – Compare String Operands</strong></td>
<td></td>
</tr>
<tr>
<td>compare string operands [ X at DS:(E)SI with Y at ES:(E)DI ]</td>
<td>1010 011w</td>
</tr>
<tr>
<td>qword at address RSI with qword at address RDI</td>
<td>0100 1000 1010 0111</td>
</tr>
<tr>
<td><strong>CMPXCHG – Compare and Exchange</strong></td>
<td></td>
</tr>
<tr>
<td>register1, register2</td>
<td>0000 1111 : 1011 000w : 11 reg2 reg1</td>
</tr>
<tr>
<td>byteregister1, byteregister2</td>
<td>0100 000B 0000 1111 : 1011 0000 : 11 bytereg2 reg1</td>
</tr>
<tr>
<td>qwordregister1, qwordregister2</td>
<td>0100 100B 0000 1111 : 1011 0001 : 11 qwordreg2 reg1</td>
</tr>
<tr>
<td>memory, register</td>
<td>0000 1111 : 1011 000w : mod reg r/m</td>
</tr>
<tr>
<td>memory8, byteregister</td>
<td>0100 00XB 0000 1111 : 1011 0000 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64, qwordregister</td>
<td>0100 10XB 0000 1111 : 1011 0001 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>CPUID – CPU Identification</strong></td>
<td>0000 1111 : 1010 0010</td>
</tr>
<tr>
<td><strong>CQO – Sign-Extend RAX</strong></td>
<td>0100 1000 1001 1001</td>
</tr>
<tr>
<td><strong>CWD – Convert Word to Doubleword</strong></td>
<td>1001 1001</td>
</tr>
<tr>
<td><strong>CWDE – Convert Word to Doubleword</strong></td>
<td>1001 1000</td>
</tr>
<tr>
<td><strong>DEC – Decrement by 1</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B 1111 1111w : 11 001 reg</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 100B 1111 1111 : 11 001 qwordreg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB 1111 1111w : mod 001 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 10XB 1111 1111 : mod 001 r/m</td>
</tr>
<tr>
<td><strong>DIV – Unsigned Divide</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>0100 000B 1111 011w : 11 110 reg</td>
</tr>
<tr>
<td>Divide RDX:RAX by qwordregister</td>
<td>0100 100B 1111 0111 : 11 110 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>0100 00XB 1111 0111w : mod 110 r/m</td>
</tr>
<tr>
<td>Divide RDX:RAX by memory64</td>
<td>0100 10XB 1111 0111 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>ENTER – Make Stack Frame for High Level Procedure</strong></td>
<td>1010 011w</td>
</tr>
<tr>
<td><strong>HLT – Halt</strong></td>
<td>1111 0100</td>
</tr>
<tr>
<td><strong>IDIV – Signed Divide</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX by register</td>
<td>0100 000B 1111 011w : 11 111 reg</td>
</tr>
<tr>
<td>RDX:RAX by qwordregister</td>
<td>0100 100B 1111 0111 : 11 111 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX by memory</td>
<td>0100 00XB 1111 0111w : mod 111 r/m</td>
</tr>
<tr>
<td>RDX:RAX by memory64</td>
<td>0100 10XB 1111 0111 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>IMUL – Signed Multiply</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL, AX, or EAX with register</td>
<td>0100 000B 1111 011w : 11 101 reg</td>
</tr>
<tr>
<td>RDX:RAX := RAX with qwordregister</td>
<td>0100 100B 1111 0111 : 11 101 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>0100 00XB 1111 011w : mod 101 r/m</td>
</tr>
<tr>
<td>RDX:RAX := RAX with memory64</td>
<td>0100 10XB 1111 0111 : mod 101 r/m</td>
</tr>
<tr>
<td>register1 with register2</td>
<td>0000 1111 : 1010 1111 : 11 : reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 := qwordregister1 with qwordregister2</td>
<td>0100 10OB 0000 1111 : 1010 1111 : 11 : qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register with memory</td>
<td>0100 0RXB 0000 1111 : 1010 1111 : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister := qwordregister with memory64</td>
<td>0100 1RXB 0000 1111 : 1010 1111 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register1 with immediate to register2</td>
<td>0100 0RXB 0110 10s1 : 11 reg1 reg2 : imm</td>
</tr>
<tr>
<td>qwordregister1 := qwordregister2 with sign-extended immediate8</td>
<td>0100 10OB 0110 1011 : 11 qwordreg1 qwordreg2 : imm8</td>
</tr>
<tr>
<td>qwordregister1 := qwordregister2 with immediate32</td>
<td>0100 10OB 0110 1001 : 11 qwordreg1 qwordreg2 : imm32</td>
</tr>
<tr>
<td>memory with immediate to register</td>
<td>0100 0RXB 0110 10s1 : mod reg r/m : imm</td>
</tr>
<tr>
<td>qwordregister := memory64 with sign-extended immediate8</td>
<td>0100 1RXB 0110 1011 : mod qwordreg r/m : imm8</td>
</tr>
<tr>
<td>qwordregister := memory64 with immediate32</td>
<td>0100 1RXB 0110 1001 : mod qwordreg r/m : imm32</td>
</tr>
</tbody>
</table>

**IN – Input From Port**

- fixed port | 1110 010w : port number |
- variable port | 1110 110w |

**INC – Increment by 1**

- reg | 0100 000B 1111 111w : 11 000 reg |
- qwordreg | 0100 100B 1111 1111 : 11 000 qwordreg |
- memory | 0100 00XB 1111 111w : mod 000 r/m |
- memory64 | 0100 10XB 1111 1111 : mod 000 r/m |

**INS – Input from DX Port**

0110 110w

**INT n – Interrupt Type n**

1100 1101 : type

**INT – Single-Step Interrupt 3**

1100 1100

**INTO – Interrupt 4 on Overflow**

1100 1110

**INVD – Invalidate Cache**

0000 1111 : 0000 1000

**INVLPD – Invalidate TLB Entry**

0000 1111 : 0000 0001 : mod 111 r/m

**INVPCID – Invalidate Process-Context Identifier**

0110 0110:0000 1111:0011 1000:1000 0010:mod reg r/m

**IRETO – Interrupt Return**

1100 1111

**Jcc – Jump if Condition is Met**

8-bit displacement | 0111 tttn : 8-bit displacement |

displacements (excluding 16-bit relative offsets) | 0000 1111 : 1000 tttn : displacement32

**JCXZ/JECXZ – Jump on CX/ECX Zero**

Address-size prefix differentiates JCXZ and JECXZ | 1110 0011 : 8-bit displacement |

**JMP – Unconditional Jump (to same segment)**

short | 1110 1011 : 8-bit displacement |
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct</td>
<td>1110 1001: displacement32</td>
</tr>
<tr>
<td>register indirect</td>
<td>0100 W00B: 1111 1111: 11 100 reg</td>
</tr>
<tr>
<td>memory indirect</td>
<td>0100 W0XB: 1111 1111: mod 100 r/m</td>
</tr>
</tbody>
</table>

**JMP – Unconditional Jump (to other segment)**

| indirect intersegment                       | 0100 00XB: 1111 1111: mod 101 r/m                                     |
| 64-bit indirect intersegment                | 0100 10XB: 1111 1111: mod 101 r/m                                     |

**LAR – Load Access Rights Byte**

| from register                               | 0100 0R0B: 0000 1111: 0000 0010: 11 reg1 reg2                          |
| from dwordregister to qwordregister, masked by 00FxFF00H | 0100 WR0B: 0000 1111: 0000 0010: 11 qwordreg1 dwordreg2              |
| from memory                                 | 0100 0RXB: 0000 1111: 0000 0010: mod reg r/m                           |
| from memory32 to qwordregister, masked by 00FxFF00H | 0100 WRXB 0000 1111: 0000 0010: mod r/m                                |

**LEA – Load Effective Address**

| in wordregister/dwordregister                | 0100 ORXB: 1000 1101: modA reg r/m                                     |
| in qwordregister                             | 0100 1RXB: 1000 1101: modA qwordreg r/m                                |

**LEAVE – High Level Procedure Exit**

|                                               | 1100 1001                                                              |

**LFS – Load Pointer to FS**

| FS:r16/r32 with far pointer from memory       | 0100 ORXB: 0000 1111: 1011 0100: modA reg r/m                          |
| FS:r64 with far pointer from memory           | 0100 1RXB: 0000 1111: 1011 0101: modA qwordreg r/m                     |

**LGDT – Load Global Descriptor Table Register**

|                                               | 0100 10XB: 0000 1111: 0000 0001: modA 010 r/m                           |

**LGS – Load Pointer to GS**

| GS:r16/r32 with far pointer from memory       | 0100 ORXB: 0000 1111: 1011 0101: modA reg r/m                          |
| GS:r64 with far pointer from memory           | 0100 1RXB: 0000 1111: 1011 0101: modA qwordreg r/m                     |

**LIDT – Load Interrupt Descriptor Table Register**

|                                               | 0100 10XB: 0000 1111: 0000 0001: modA 011 r/m                           |

**LLDT – Load Local Descriptor Table Register**

| LDTR from register                           | 0100 000B: 0000 1111: 0000 0000: 11 010 reg                             |
| LDTR from memory                             | 0100 00XB: 0000 1111: 0000 0000: mod 010 r/m                           |

**LMSW – Load Machine Status Word**

| from register                                | 0100 000B: 0000 1111: 0000 0001: 11 110 reg                             |
| from memory                                  | 0100 00XB: 0000 1111: 0000 0001: mod 110 r/m                           |

**LOCK – Assert LOCK# Signal Prefix**

|                                               | 1111 0000                                                              |

**LODS/LODSB/LODSW/LODSD/LODSQ – Load String Operand**

| at DS:(E)SI to AL/EAX/EAX                     | 1010 110w                                                              |
| at (R)SI to RAX                               | 0100 1000 1010 1101                                                     |

**LOOP – Loop Count**

| if count ≠ 0, 8-bit displacement             | 1110 0010                                                              |
| if count ≠ 0, RIP + 8-bit displacement sign-extended to 64-bits | 0100 1000 1110 0010                                                     |

**LOOPE – Loop Count while Zero/Equal**
### INSTRUCTION FORMATS AND ENCODINGS

#### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>if count ≠ 0 &amp; ZF = 1, 8-bit displacement</td>
<td>1110 0001</td>
</tr>
<tr>
<td>if count ≠ 0 &amp; ZF = 1, RIP + 8-bit displacement sign-extended to 64-bits</td>
<td>0100 1000 1110 0001</td>
</tr>
</tbody>
</table>

**LOOPNE/LOOPNZ – Loop Count while not Zero/Equal**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>if count ≠ 0 &amp; ZF = 0, 8-bit displacement</td>
<td>1110 0000</td>
</tr>
<tr>
<td>if count ≠ 0 &amp; ZF = 0, RIP + 8-bit displacement sign-extended to 64-bits</td>
<td>0100 1000 1110 0000</td>
</tr>
</tbody>
</table>

**LSL – Load Segment Limit**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>from register</td>
<td>0000 1111 : 0000 0011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>from qwordregister</td>
<td>0100 1R00 0000 1111 : 0000 0011 : 11 qwordreg1 reg2</td>
</tr>
<tr>
<td>from memory16</td>
<td>0000 1111 : 0000 0011 : mod reg r/m</td>
</tr>
<tr>
<td>from memory64</td>
<td>0100 1RXB 0000 1111 : 0000 0011 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**LSS – Load Pointer to SS**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS:r16/r32 with far pointer from memory</td>
<td>0100 0RXB : 0000 1111 : 1011 0010 : mod reg r/m</td>
</tr>
<tr>
<td>SS:r64 with far pointer from memory</td>
<td>0100 1wXB : 0000 1111 : 1011 0010 : mod qwordreg r/m</td>
</tr>
</tbody>
</table>

**LTR – Load Task Register**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>from register</td>
<td>0100 0R00 : 0000 1111 : 0000 0000 : 11 011 reg</td>
</tr>
<tr>
<td>from memory</td>
<td>0100 00XB : 0000 1111 : 0000 0000 : mod 011 r/m</td>
</tr>
</tbody>
</table>

**MOV – Move Data**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B : 1000 1000 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 1000 1001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 1000 1011 : 11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister2 to qwordregister1</td>
<td>0100 1R0B 1000 1011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to reg</td>
<td>0100 0RXB : 1000 1011 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB 1000 1011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>reg to memory</td>
<td>0100 0RXB : 1000 1000 : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB 1000 1001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 000B : 1100 0111 : 11 000 reg : imm</td>
</tr>
<tr>
<td>immediate32 to qwordregister (zero extend)</td>
<td>0100 100B 1100 0111 : 11 000 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate to register (alternate encoding)</td>
<td>0100 000B : 1011 w reg : imm</td>
</tr>
<tr>
<td>immediate64 to qwordregister (alternate encoding)</td>
<td>0100 100B 1011 1000 reg : imm64</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 00XB : 1100 0111 : mod 000 r/m : imm</td>
</tr>
<tr>
<td>immediate32 to memory64 (zero extend)</td>
<td>0100 10XB 1100 0111 : mod 000 r/m : imm32</td>
</tr>
<tr>
<td>memory to AL, AX, or EAX</td>
<td>0100 0000 : 1010 0000 : displacement</td>
</tr>
<tr>
<td>memory64 to RAX</td>
<td>0100 1000 1010 0001 : displacement64</td>
</tr>
<tr>
<td>AL, AX, or EAX to memory</td>
<td>0100 0000 : 1010 0011 : displacement</td>
</tr>
<tr>
<td>RAX to memory64</td>
<td>0100 1000 1010 0011 : displacement64</td>
</tr>
</tbody>
</table>

**MOV – Move to/from Control Registers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR0-CR4 from register</td>
<td>0100 0R0B : 0000 1111 : 0010 0010 : 11 eee reg (eee = CR#)</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRx from qwordregister</td>
<td>0100 1R0B : 0000 1111 : 0010 0010 : 11 eee qwordreg (Reee = CR#)</td>
</tr>
<tr>
<td>register from CR0-CR4</td>
<td>0100 0R0B : 0000 1111 : 0010 0011 : 11 eee reg (eee = CR#)</td>
</tr>
<tr>
<td>qwordregister from CRx</td>
<td>0100 1R0B 0000 1111 : 0010 0000 : 11 eee qwordreg (Reee = CR#)</td>
</tr>
<tr>
<td>MOV – Move to/from Debug Registers</td>
<td></td>
</tr>
<tr>
<td>DRO-DR7 from register</td>
<td>0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)</td>
</tr>
<tr>
<td>DRO-DR7 from quadregister</td>
<td>0100 100B 0000 1111 : 0010 0011 : 11 eee reg (eee = DR#)</td>
</tr>
<tr>
<td>register from DRO-DR7</td>
<td>0000 1111 : 0010 0001 : 11 eee reg (eee = DR#)</td>
</tr>
<tr>
<td>quadregister from DRO-DR7</td>
<td>0100 100B 0000 1111 : 0010 0001 : 11 eee quadreg (eee = DR#)</td>
</tr>
<tr>
<td>MOV – Move to/from Segment Registers</td>
<td></td>
</tr>
<tr>
<td>register to segment register</td>
<td>0100 W00B : 1000 1110 : 11 sreg reg</td>
</tr>
<tr>
<td>register to SS</td>
<td>0100 000B : 1000 1110 : 11 sreg reg</td>
</tr>
<tr>
<td>memory to segment register</td>
<td>0100 00XB : 1000 1110 : mod sreg r/m</td>
</tr>
<tr>
<td>memory64 to segment register (lower 16 bits)</td>
<td>0100 10XB 1000 1110 : mod sreg r/m</td>
</tr>
<tr>
<td>memory to SS</td>
<td>0100 00XB : 1000 1110 : mod sreg r/m</td>
</tr>
<tr>
<td>segment register to register</td>
<td>0100 000B : 1000 1100 : 11 sreg reg</td>
</tr>
<tr>
<td>segment register to qwordregister (zero extended)</td>
<td>0100 100B 1000 1100 : 11 sreg qwordreg</td>
</tr>
<tr>
<td>segment register to memory</td>
<td>0100 00XB : 1000 1100 : mod sreg r/m</td>
</tr>
<tr>
<td>segment register to memory64 (zero extended)</td>
<td>0100 10XB 1000 1100 : mod sreg3 r/m</td>
</tr>
<tr>
<td>MOVBE – Move data after swapping bytes</td>
<td></td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 1111 : 0011 1000:1111 0000 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB : 0000 1111 : 0011 1000:1111 0000 : mod reg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0100 0RXB : 0000 1111 : 0011 1000:1111 0001 : mod reg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB : 0000 1111 : 0011 1000:1111 0001 : mod reg r/m</td>
</tr>
<tr>
<td>MOVS/MOVSB/MOVSW/MOVSD/MOVSQ – Move Data from String to String</td>
<td></td>
</tr>
<tr>
<td>Move data from string to string</td>
<td>1010 010w</td>
</tr>
<tr>
<td>Move data from string to string (qword)</td>
<td>0100 1000 1010 0101</td>
</tr>
<tr>
<td>MOVSX/MOVSXD – Move with Sign-Extend</td>
<td></td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0000 1111 : 1011 111w : 11 reg1 reg2</td>
</tr>
<tr>
<td>bytregister2 to qwordregister1 (sign-extend)</td>
<td>0100 1R0B 0000 1111 : 1011 1110 : 11 quadreg1 bytereg2</td>
</tr>
<tr>
<td>wordregister2 to qwordregister1</td>
<td>0100 1R0B 0000 1111 : 1011 1111 : 11 quadreg1 wordreg2</td>
</tr>
<tr>
<td>dwordregister2 to qwordregister1</td>
<td>0100 1R0B 0110 0011 : 11 quadreg1 dwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 1111 : 1011 111w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to qwordregister (sign-extend)</td>
<td>0100 1RXB 0000 1111 : 1011 1110 : mod qwordreg r/m</td>
</tr>
<tr>
<td>memory16 to qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 1111 : mod qwordreg r/m</td>
</tr>
<tr>
<td>memory32 to qwordregister</td>
<td>0100 1RXB 0110 0011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>MOVZX – Move with Zero-Extend</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B : 0000 1111 : 1011 011w : 11 reg1 reg2</td>
</tr>
<tr>
<td>dwordregister2 to qwordregister1</td>
<td>0100 1R0B 0000 1111 : 1011 0111 : 11 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB : 0000 1111 : 1011 011w : mod reg r/m</td>
</tr>
<tr>
<td>memory32 to qwordregister</td>
<td>0100 1RXB 0000 1111 : 1011 0111 : mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>MUL - Unsigned Multiply</strong></td>
<td></td>
</tr>
<tr>
<td>AL, AX, or EAX with register</td>
<td>0100 000B : 1111 011w : 11 100 reg</td>
</tr>
<tr>
<td>RAX with qwordregister (to RDX:RAX)</td>
<td>0100 100B 1111 0111 : 11 100 qwordreg</td>
</tr>
<tr>
<td>AL, AX, or EAX with memory</td>
<td>0100 00XB : 1111 011w : mod 100 r/m</td>
</tr>
<tr>
<td>RAX with memory64 (to RDX:RAX)</td>
<td>0100 10XB 1111 0111 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>NEG - Two's Complement Negation</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B : 1111 011w : 11 011 reg</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 100B 1111 0111 : 11 011 qwordreg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB : 1111 011w : mod 011 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 10XB 1111 0111 : mod 011 r/m</td>
</tr>
<tr>
<td><strong>NOP - No Operation</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1001 0000</td>
</tr>
<tr>
<td><strong>NOT - One's Complement Negation</strong></td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B : 1111 011w : 11 010 reg</td>
</tr>
<tr>
<td>qwordregister</td>
<td>0100 000B 1111 0111 : 11 010 qwordreg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 00XB : 1111 011w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64</td>
<td>0100 1RXB 1111 0111 : mod 010 r/m</td>
</tr>
<tr>
<td><strong>OR - Logical Inclusive OR</strong></td>
<td></td>
</tr>
<tr>
<td>register1 to register2</td>
<td>0000 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 to byteregister2</td>
<td>0100 0R0B 0000 1000 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister1 to qwordregister2</td>
<td>0100 1R0B 0000 1001 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0000 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 0R0B 0000 1010 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister2 to qwordregister1</td>
<td>0100 1R0B 0000 1011 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to byteregister</td>
<td>0100 0RXB 0000 1010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory8 to qwordregister</td>
<td>0100 0RXB 0000 1011 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register to memory</td>
<td>0000 100w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister to memory8</td>
<td>0100 0RXB 0000 1000 : mod bytereg r/m</td>
</tr>
<tr>
<td>qwordregister to memory64</td>
<td>0100 1RXB 0000 1001 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>1000 00sw : 11 001 reg : imm</td>
</tr>
<tr>
<td>immediate8 to byteregister</td>
<td>0100 000B 1000 0000 : 11 001 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 000B 1000 0001 : 11 001 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate8 to qwordregister</td>
<td>0100 000B 1000 0011 : 11 001 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0000 110w : imm</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate64 to RAX</td>
<td>0100 1000 0000 1101 : imm64</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>1000 00sw : mod 001 r/m : imm</td>
</tr>
<tr>
<td>immediate8 to memory8</td>
<td>0100 00XB 1000 0000 : mod 001 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 00XB 1000 0001 : mod 001 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 00XB 1000 0011 : mod 001 r/m : imm8</td>
</tr>
</tbody>
</table>

**OUT – Output to Port**

- fixed port 1110 011w : port number
- variable port 1110 111w

**OUTS – Output to DX Port**

- output to DX Port 0110 111w

**POP – Pop a Value from the Stack**

- wordregister 0101 0101 : 0100 000B : 1000 1111 : 11 000 reg16
- qwordregister 0100 W00B : 1000 1111 : 11 000 reg64
- wordregister (alternate encoding) 0101 0101 : 0100 000B : 0101 1 reg16
- qwordregister (alternate encoding) 0100 W00B : 0101 1 reg64
- memory64 0100 W0XB : 1000 1111 : mod 000 r/m
- memory16 0101 0101 : 0100 00XB 1000 1111 : mod 000 r/m

**POP – Pop a Segment Register from the Stack**

(Note: CS cannot be sreg2 in this usage.)

- segment register FS, GS 0000 1111: 10 sreg3 001

**POPF/POPFQ – Pop Stack into FLAGS/RFLAGS Register**

- pop stack to FLAGS register 0101 0101 : 1001 1101
- pop Stack to RFLAGS register 0100 1000 1001 1101

**PUSH – Push Operand onto the Stack**

- wordregister 0101 0101 : 0100 000B : 1111 1111 : 11 110 reg16
- qwordregister 0100 W00B : 1111 1111 : 11 110 reg64
- wordregister (alternate encoding) 0101 0101 : 0100 000B : 0101 0 reg16
- qwordregister (alternate encoding) 0100 W00B : 0101 0 reg64
- memory16 0101 0101 : 0100 000B : 1111 1111 : mod 110 r/m
- memory64 0100 W00B : 1111 1111 : mod 110 r/m
- immediate8 0110 1010 : imm8
- immediate16 0101 0101 : 0110 1000 : imm16
- immediate64 0110 1000 : imm64

**PUSH – Push Segment Register onto the Stack**

- segment register FS,GS 0000 1111: 10 sreg3 000

**PUSHF/PUSHFD – Push Flags Register onto the Stack**

- 1001 1100

**RCL – Rotate thru Carry Left**

- register by 1 0100 000B : 1101 000w : 11 010 reg
- qwordregister by 1 0100 100B 1101 0001 : 11 010 qwordreg
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory by 1</td>
<td>0100 00XB : 1101 000w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 010 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B : 1101 001w : 11 010 reg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 010 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 000B : 1101 001w : mod 010 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 100B 1101 0011 : mod 010 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B : 1100 000w : 11 010 reg : imm</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 010 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 000B : 1100 000w : mod 010 r/m : imm</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 100B 1100 0001 : mod 010 r/m : imm8</td>
</tr>
</tbody>
</table>

**RCR - Rotate thru Carry Right**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register by 1</td>
<td>0100 000B : 1101 000w : 11 011 reg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0011 : 11 011 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 000B : 1101 000w : mod 011 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 100B 1101 0011 : mod 011 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B : 1101 001w : 11 011 reg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0010 : 11 011 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 000B : 1101 001w : mod 011 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 100B 1101 0011 : mod 011 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B : 1100 000w : 11 011 reg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 011 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 000B : 1100 000w : mod 011 r/m : imm</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 100B 1100 0001 : mod 011 r/m : imm8</td>
</tr>
</tbody>
</table>

**RDSMR - Read from Model-Specific Register**

- load ECX-specified register into EDX:EAX: 0000 1111 : 0011 0010

**RDPMC - Read Performance Monitoring Counters**

- load ECX-specified performance counter into EDX:EAX: 0000 1111 : 0011 0011

**RDTSC - Read Time-Stamp Counter**

- read time-stamp counter into EDX:EAX: 0000 1111 : 0011 0001

**RDTSCP - Read Time-Stamp Counter and Processor ID**

- 0000 1111 : 0000 0001 : 1111 1001

**REP INS - Input String**

**REP LODS - Load String**

**REP MOVS - Move String**

**REP OUTS - Output String**

**REP STOS - Store String**

**REPE CMPS - Compare String**

**REPE SCAS - Scan String**

**REPNE CMPS - Compare String**
**Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REPNE SCAS – Scan String</strong></td>
<td></td>
</tr>
<tr>
<td>RET – Return from Procedure (to same segment)</td>
<td></td>
</tr>
<tr>
<td>no argument</td>
<td>1100 0011</td>
</tr>
<tr>
<td>adding immediate to SP</td>
<td>1100 0010 : 16-bit displacement</td>
</tr>
<tr>
<td>RET – Return from Procedure (to other segment)</td>
<td></td>
</tr>
<tr>
<td>intersegment</td>
<td>1100 1011</td>
</tr>
<tr>
<td>adding immediate to SP</td>
<td>1100 1010 : 16-bit displacement</td>
</tr>
<tr>
<td><strong>ROL – Rotate Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 000 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 000 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 000 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 000B 1101 0000 : mod 000 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 000B 1101 0000 : mod 000 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 100B 1101 0001 : mod 000 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 000 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 000 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 000 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 000B 1101 001w : mod 000 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 000B 1101 0010 : mod 000 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 100B 1101 0011 : mod 000 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>1100 000w : 11 000 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 000 bytereg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 000 bytereg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>1100 000w : mod 000 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 000B 1100 0000 : mod 000 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 100B 1100 0001 : mod 000 r/m : imm8</td>
</tr>
<tr>
<td><strong>ROR – Rotate Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 001 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 001 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 001 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 000B 1101 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 000B 1101 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 100B 1101 0001 : mod 001 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 001 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 001 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 001 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 000B 1101 001w : mod 001 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 000B 1101 0010 : mod 001 r/m</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

#### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 001 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 001 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 001 reg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 001 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 001 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 001 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 001 r/m : imm8</td>
</tr>
</tbody>
</table>

RSM – Resume from System Management Mode

SAR – Shift Arithmetic Right

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 111 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 111 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 111 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB 1101 000w : mod 111 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 10XB 1101 0000 : mod 111 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 111 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 111 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 111 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 111 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 1101 001w : mod 111 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 00XB 1101 0010 : mod 111 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 00XB 1101 0011 : mod 111 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 111 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 111 bytereg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 111 qwordreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 111 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 00XB 1100 0001 : mod 111 r/m : imm8</td>
</tr>
</tbody>
</table>

SBB – Integer Subtraction with Borrow

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 to register2</td>
<td>0100 0R0B 0001 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 to byteregister2</td>
<td>0100 0R0B 0001 1000 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>quadregister1 to quadregister2</td>
<td>0100 1R0B 0001 1001 : 11 quadreg1 quadreg2</td>
</tr>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B 0001 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 0R0B 0001 1010 : 11 reg1 bytereg2</td>
</tr>
<tr>
<td>byteregister2 to byteregister1</td>
<td>0100 1R0B 0001 1011 : 11 reg1 bytereg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0001 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 to byteregister</td>
<td>0100 0RXB 0001 1010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64 to byteregister</td>
<td>0100 1RXB 0001 1011 : mod quadreg r/m</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register to memory</td>
<td>0100 0RXB 0001 100w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister to memory8</td>
<td>0100 0RXB 0001 1000 : mod reg r/m</td>
</tr>
<tr>
<td>quadregister to memory64</td>
<td>0100 1RXB 0001 1001 : mod reg r/m</td>
</tr>
<tr>
<td>immediate to register</td>
<td>0100 000B 1000 00sw : 11 011 r/m : imm</td>
</tr>
<tr>
<td>immediate8 to byteregister</td>
<td>0100 000B 1000 0000 : 11 011 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 to qwordregister</td>
<td>0100 100B 1000 0001 : 11 011 qwordreg : imm32</td>
</tr>
<tr>
<td>immediate8 to qwordregister</td>
<td>0100 100B 1000 0011 : 11 011 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate to AL, AX, or EAX</td>
<td>0100 000B 0001 110w : imm</td>
</tr>
<tr>
<td>immediate32 to RAL</td>
<td>0100 1000 0001 1101 : imm32</td>
</tr>
<tr>
<td>immediate to memory</td>
<td>0100 000B 1000 00sw : mod 011 r/m : imm</td>
</tr>
<tr>
<td>immediate8 to memory8</td>
<td>0100 000B 1000 0000 : mod 011 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 to memory64</td>
<td>0100 100B 1000 0001 : mod 011 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 100B 1000 0011 : mod 011 r/m : imm8</td>
</tr>
<tr>
<td>SCAS/SCASB/SCASW/SCASD – Scan String</td>
<td></td>
</tr>
<tr>
<td>scan string</td>
<td>1010 111w</td>
</tr>
<tr>
<td>scan string (compare AL with byte at RDI)</td>
<td>0100 1000 1010 1110</td>
</tr>
<tr>
<td>scan string (compare RAX with qword at RDI)</td>
<td>0100 1000 1010 1111</td>
</tr>
<tr>
<td>SETcc – Byte Set on Condition</td>
<td></td>
</tr>
<tr>
<td>register</td>
<td>0100 000B 0000 1111 : 1001 tttn: 11 000 reg</td>
</tr>
<tr>
<td>register</td>
<td>0100 0000 0000 1111 : 1001 tttn: 11 000 reg</td>
</tr>
<tr>
<td>memory</td>
<td>0100 000B 0000 1111 : 1001 tttn: mod 000 r/m</td>
</tr>
<tr>
<td>memory</td>
<td>0100 0000 0000 1111 : 1001 tttn: mod 000 r/m</td>
</tr>
<tr>
<td>SGD – Store Global Descriptor Table Register</td>
<td>0000 1111 : 0000 0001 : modA 000 r/m</td>
</tr>
<tr>
<td>SHL – Shift Left</td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 100 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 100 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 100 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 000B 1101 000w : mod 100 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 000B 1101 0000 : mod 100 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 100B 1101 0001 : mod 100 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 100 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 100 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 100 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 000B 1101 001w : mod 100 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 000B 1101 0010 : mod 100 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 100B 1101 0011 : mod 100 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 100 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 100 bytereg : imm8</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>quadregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 100 quadreg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 100 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 100 r/m : imm8</td>
</tr>
<tr>
<td><strong>SHLD – Double Precision Shift Left</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 0R0B 0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate8</td>
<td>0100 1R0B 0000 1111 : 1010 0100 : 11 qwordreg2 qwordreg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 0RXB 0000 1111 : 1010 0100 : mod reg r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate8</td>
<td>0100 1RXB 0000 1111 : 1010 0100 : mod qwordreg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 0R0B 0000 1111 : 1010 0101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>quadregister by CL</td>
<td>0100 1R0B 0000 1111 : 1010 0101 : 11 quadreg2 quadreg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 0000 1111 : 1010 0101 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 0000 1111 : 1010 0101 : mod quadreg r/m</td>
</tr>
<tr>
<td><strong>SHR – Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by 1</td>
<td>0100 000B 1101 000w : 11 101 reg</td>
</tr>
<tr>
<td>byteregister by 1</td>
<td>0100 000B 1101 0000 : 11 101 bytereg</td>
</tr>
<tr>
<td>qwordregister by 1</td>
<td>0100 100B 1101 0001 : 11 101 qwordreg</td>
</tr>
<tr>
<td>memory by 1</td>
<td>0100 00XB 1101 000w : mod 101 r/m</td>
</tr>
<tr>
<td>memory8 by 1</td>
<td>0100 00XB 1101 0000 : mod 101 r/m</td>
</tr>
<tr>
<td>memory64 by 1</td>
<td>0100 10XB 1101 0001 : mod 101 r/m</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 1101 001w : 11 101 reg</td>
</tr>
<tr>
<td>byteregister by CL</td>
<td>0100 000B 1101 0010 : 11 101 bytereg</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 100B 1101 0011 : 11 101 qwordreg</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0100 00XB 1101 001w : mod 101 r/m</td>
</tr>
<tr>
<td>memory8 by CL</td>
<td>0100 00XB 1101 0010 : mod 101 r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 10XB 1101 0011 : mod 101 r/m</td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 000B 1100 000w : 11 101 reg : imm8</td>
</tr>
<tr>
<td>byteregister by immediate count</td>
<td>0100 000B 1100 0000 : 11 101 reg : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate count</td>
<td>0100 100B 1100 0001 : 11 101 reg : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 00XB 1100 000w : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>memory8 by immediate count</td>
<td>0100 00XB 1100 0000 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>memory64 by immediate count</td>
<td>0100 10XB 1100 0001 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td><strong>SHRD – Double Precision Shift Right</strong></td>
<td></td>
</tr>
<tr>
<td>register by immediate count</td>
<td>0100 0R0B 0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8</td>
</tr>
<tr>
<td>qwordregister by immediate8</td>
<td>0100 1R0B 0000 1111 : 1010 1100 : 11 qwordreg2 qwordreg1 : imm8</td>
</tr>
<tr>
<td>memory by immediate count</td>
<td>0100 0RXB 0000 1111 : 1010 1100 : mod reg r/m : imm8</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory64 by immediate8</td>
<td>0100 1RXB 0000 1111 : 1010 1100 : mod qwordreg r/m : imm8</td>
</tr>
<tr>
<td>register by CL</td>
<td>0100 000B 0000 1111 : 1010 1101 : 11 reg2 reg1</td>
</tr>
<tr>
<td>qwordregister by CL</td>
<td>0100 1R0B 0000 1111 : 1010 1101 : 11 qwordreg2 qwordreg1</td>
</tr>
<tr>
<td>memory by CL</td>
<td>0000 1111 : 1010 1101 : mod reg r/m</td>
</tr>
<tr>
<td>memory64 by CL</td>
<td>0100 1RXB 0000 1111 : 1010 1101 : mod qwordreg r/m</td>
</tr>
<tr>
<td>SIDT – Store Interrupt Descriptor Table Register</td>
<td>0000 1111 : 0000 0001 : modA 001 r/m</td>
</tr>
<tr>
<td>SLDT – Store Local Descriptor Table Register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 000 reg</td>
</tr>
<tr>
<td>to register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 100 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0100 000B 0000 1111 : 0000 0000 : mod 100 r/m</td>
</tr>
<tr>
<td>SMSW – Store Machine Status Word</td>
<td>1111 1011</td>
</tr>
<tr>
<td>to register</td>
<td>0100 000B 0000 1111 : 0000 0001 : 11 100 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0100 000B 0000 1111 : 0000 0001 : mod 100 r/m</td>
</tr>
<tr>
<td>STC – Set Carry Flag</td>
<td>1111 1001</td>
</tr>
<tr>
<td>STD – Set Direction Flag</td>
<td>1111 1101</td>
</tr>
<tr>
<td>STI – Set Interrupt Flag</td>
<td>1111 1101</td>
</tr>
<tr>
<td>STOS/STOSB/STOSW/STOSD/STOSQ – Store String Data</td>
<td>1010 101w</td>
</tr>
<tr>
<td>store string data</td>
<td>0100 1000 1010 1011</td>
</tr>
<tr>
<td>store string data (RAX at address RDI)</td>
<td>0100 1000 1010 1011</td>
</tr>
<tr>
<td>STR – Store Task Register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 001 reg</td>
</tr>
<tr>
<td>to register</td>
<td>0100 000B 0000 1111 : 0000 0000 : 11 000 reg</td>
</tr>
<tr>
<td>to memory</td>
<td>0100 000B 0000 1111 : 0000 0000 : mod 001 r/m</td>
</tr>
<tr>
<td>SUB – Integer Subtraction</td>
<td>0100 0R0B 0010 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>register1 from register2</td>
<td>0100 0R0B 0010 100w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister1 from byteregister2</td>
<td>0100 0R0B 0010 1000 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister1 from qwordregister2</td>
<td>0100 1R0B 0010 1000 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>register2 from register1</td>
<td>0100 0R0B 0010 101w : 11 reg1 reg2</td>
</tr>
<tr>
<td>byteregister2 from byteregister1</td>
<td>0100 0R0B 0010 1010 : 11 bytereg1 bytereg2</td>
</tr>
<tr>
<td>qwordregister2 from qwordregister1</td>
<td>0100 1R0B 0010 1101 : 11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory from register</td>
<td>0100 0RXB 0010 101w : mod reg r/m</td>
</tr>
<tr>
<td>memory8 from byteregister</td>
<td>0100 0RXB 0010 1010 : mod bytereg r/m</td>
</tr>
<tr>
<td>memory64 from qwordregister</td>
<td>0100 1RXB 0010 1101 : mod qwordreg r/m</td>
</tr>
<tr>
<td>register from memory</td>
<td>0100 0RXB 0010 100w : mod reg r/m</td>
</tr>
<tr>
<td>byteregister from memory8</td>
<td>0100 0RXB 0010 1000 : mod bytereg r/m</td>
</tr>
<tr>
<td>qwordregister from memory8</td>
<td>0100 1RXB 0010 1000 : mod qwordreg r/m</td>
</tr>
<tr>
<td>immediate from register</td>
<td>0100 000B 1000 00sw : 11 101 reg : imm</td>
</tr>
<tr>
<td>immediate8 from byteregister</td>
<td>0100 000B 1000 0000 : 11 101 bytereg : imm8</td>
</tr>
<tr>
<td>immediate32 from qwordregister</td>
<td>0100 100B 1000 0001 : 11 101 qwordreg : imm32</td>
</tr>
</tbody>
</table>
### Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate8 from qwordregister</td>
<td>0100 100B 1000 0011 : 11 101 qwordreg : imm8</td>
</tr>
<tr>
<td>immediate from AL, AX, or EAX</td>
<td>0100 000B 0010 110w : imm</td>
</tr>
<tr>
<td>immediate32 from RAX</td>
<td>0100 1000 0010 1101 : imm32</td>
</tr>
<tr>
<td>immediate from memory</td>
<td>0100 00XB 1000 00sw : mod 101 r/m : imm</td>
</tr>
<tr>
<td>immediate8 from memory8</td>
<td>0100 00XB 1000 0000 : mod 101 r/m : imm8</td>
</tr>
<tr>
<td>immediate32 from memory64</td>
<td>0100 10XB 1000 0001 : mod 101 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 from memory64</td>
<td>0100 10XB 1000 0011 : mod 101 r/m : imm8</td>
</tr>
</tbody>
</table>

**SWAPGS – Swap GS Base Register**

Exchanges the current GS base register value for value in MSR C0000102H

0000 1111 0000 0001 1111 1000

**SYSCALL – Fast System Call**

fast call to privilege level 0 system procedures

0000 1111 0000 0101

**SYSRET – Return From Fast System Call**

return from fast system call

0000 1111 0000 0111

**TEST – Logical Compare**

register1 and register2

0100 0R0B 1000 010w : 11 reg1 reg2

byteregister1 and byteregister2

0100 0R0B 1000 0100 : 11 bytereg1 bytereg2

qwordregister1 and qwordregister2

0100 1R0B 1000 0101 : 11 qwordreg1 qwordreg2

memory and register

0100 0R0B 1000 010w : mod reg r/m

memory8 and byteregister

0100 0RXB 1000 0100 : mod bytereg r/m

memory64 and qwordregister

0100 1RXB 1000 0101 : mod qwordreg r/m

immediate and register

0100 000B 1111 011w : 11 000 reg : imm

immediate8 and byteregister

0100 000B 1111 0110 : 11 000 bytereg : imm8

immediate32 and qwordregister

0100 100B 1111 0111 : 11 000 bytereg : imm8

immediate and AL, AX, or EAX

0100 000B 1010 100w : imm

immediate32 and RAX

0100 1000 1010 1001 : imm32

immediate and memory

0100 00XB 1111 011w : mod 000 r/m : imm

immediate8 and memory8

0100 1000 1111 0110 : mod 000 r/m : imm8

immediate32 and memory64

0100 1000 1111 0111 : mod 000 r/m : imm32

**UD2 – Undefined instruction**

0000 FFFF : 0000 1011

**VERR – Verify a Segment for Reading**

register

0100 000B 0000 1111 : 0000 0000 : 11 100 reg

memory

0100 00XB 0000 1111 : 0000 0000 : mod 100 r/m

**VERW – Verify a Segment for Writing**

register

0100 000B 0000 1111 : 0000 0000 : 11 101 reg

memory

0100 00XB 0000 1111 : 0000 0000 : mod 101 r/m

**WAIT – Wait**

1001 1011

**WBINVD – Writeback and Invalidate Data Cache**

0000 1111 : 0000 1001
### Instruction Formats and Encodings

#### WRMSR – Write to Model-Specific Register
- Write EDX:EAX to ECX specified MSR
  - Encoding: 0000 1111 : 0011 0000
- Write RDX[31:0]:RAX[31:0] to RCX specified MSR
  - Encoding: 0100 1000 0000 1111 : 0011 0000

#### XADD – Exchange and Add
- Input register1, register2
  - Encoding: 0100 0R0B 0000 1111 : 1100 0001 : 11 reg1 reg2
- Input byteregister1, byteregister2
  - Encoding: 0100 0R0B 0011 0000 : 11 bytereg1 bytereg2
- Input qwordregister1, qwordregister2
  - Encoding: 0100 1R0B 0011 0001 : 11 qwordreg1 qwordreg2
- Input memory, register
  - Encoding: 0100 0RXB 0000 1111 : 1100 0000w : 11 mod reg r/m
- Input memory8, bytereg
  - Encoding: 0100 1RXB 0000 1111 : 1100 0000 : 11 mod bytereg r/m
- Input memory64, qwordreg
  - Encoding: 0100 1RXB 0000 1111 : 1100 0001 : 11 mod qwordreg r/r

#### XCHG – Exchange Register/Memory with Register
- Input register1 with register2
  - Encoding: 1000 011w : 11 reg1 reg2
- Input AX or EAX with register
  - Encoding: 1001 0 reg
- Input memory with register
  - Encoding: 1000 011w : mod reg r/m

#### XLAT/XLATB – Table Look-up Translation
- AL to byte DS:[(E)BX + unsigned AL]
  - Encoding: 1101 0111
- AL to byte DS:[RBX + unsigned AL]
  - Encoding: 0100 1000 1101 0111

#### XOR – Logical Exclusive OR
- Input register1 to register2
  - Encoding: 0100 0RXB 0011 000w : 11 reg1 reg2
- Input byteregister1 to byteregister2
  - Encoding: 0100 0RXB 0011 0000 : 11 bytereg1 bytereg2
- Input qwordregister1 to qwordregister2
  - Encoding: 0100 1RXB 0011 0001 : 11 qwordreg1 qwordreg2
- Input register2 to register1
  - Encoding: 0100 0RXB 0011 001w : 11 reg1 reg2
- Input byteregister2 to byteregister1
  - Encoding: 0100 0RXB 0011 0010 : 11 bytereg1 bytereg2
- Input qwordregister2 to qwordregister1
  - Encoding: 0100 1RXB 0011 0011 : 11 qwordreg1 qwordreg2
- Input memory to register
  - Encoding: 0100 0RXB 0011 001w : mod reg r/m
- Input memory8 to byteregister
  - Encoding: 0100 0RXB 0011 0010 : mod bytereg r/m
- Input memory64 to qwordregister
  - Encoding: 0100 1RXB 0011 0011 : mod qwordreg r/m
- Input register to memory
  - Encoding: 0100 0RXB 0011 000w : mod reg r/m
- Input byteregister to memory8
  - Encoding: 0100 0RXB 0011 0000 : mod bytereg r/m
- Input qwordregister to memory8
  - Encoding: 0100 1RXB 0011 0001 : mod qwordreg r/m
- Input immediate to register
  - Encoding: 0100 000B 1000 00sw : 11 110 reg : imm
- Input immediate8 to byteregister
  - Encoding: 0100 000B 1000 0000 : 11 110 bytereg : imm8
- Input immediate32 to qwordregister
  - Encoding: 0100 100B 1000 0001 : 11 110 qwordreg : imm32
- Input immediate8 to qwordregister
  - Encoding: 0100 100B 1000 0011 : 11 110 qwordreg : imm8
- Input immediate to AL, AX, or EAX
  - Encoding: 0100 000B 0011 0101 : immediate data
- Input immediate to memory
  - Encoding: 0100 00XB 1000 00sw : mod 110 r/m : imm
- Input immediate8 to memory8
  - Encoding: 0100 00XB 1000 0000 : mod 110 r/m : imm8
INSTRUCTION FORMATS AND ENCODINGS

B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS

The following table shows formats and encodings introduced by the Pentium processor family.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate32 to memory64</td>
<td>0100 10XB 1000 0001 : mod 110 r/m : imm32</td>
</tr>
<tr>
<td>immediate8 to memory64</td>
<td>0100 10XB 1000 0011 : mod 110 r/m : imm8</td>
</tr>
</tbody>
</table>

Prefix Bytes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>address size</td>
<td>0110 0111</td>
</tr>
<tr>
<td>LOCK</td>
<td>1111 0000</td>
</tr>
<tr>
<td>operand size</td>
<td>0110 0110</td>
</tr>
<tr>
<td>CS segment override</td>
<td>0010 1110</td>
</tr>
<tr>
<td>DS segment override</td>
<td>0011 1110</td>
</tr>
<tr>
<td>ES segment override</td>
<td>0010 0110</td>
</tr>
<tr>
<td>FS segment override</td>
<td>0110 0100</td>
</tr>
<tr>
<td>GS segment override</td>
<td>0110 0101</td>
</tr>
<tr>
<td>SS segment override</td>
<td>0011 0110</td>
</tr>
</tbody>
</table>

Table B-16. Pentium® Processor Family Instruction Formats and Encodings, Non-64-Bit Modes

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPXCHG8B – Compare and Exchange 8 Bytes</td>
<td>EDX:EAX with memory64 0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
</tbody>
</table>

Table B-17. Pentium® Processor Family Instruction Formats and Encodings, 64-Bit Mode

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPXCHG8B/CMPXCHG16B – Compare and Exchange Bytes</td>
<td>EDX:EAX with memory64 0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
<tr>
<td></td>
<td>RDX:RAX with memory128 0100 10XB 0000 1111 : 1100 0111 : mod 001 r/m</td>
</tr>
</tbody>
</table>

B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS

Non-64-bit mode instruction encodings for MMX Technology, SSE, SSE2, and SSE3 are covered by applying these rules to Table B-19 through Table B-31. Table B-34 lists special encodings (instructions that do not follow the rules below).

1. The REX instruction has no effect:
   - On immediates.
   - If both operands are MMX registers.
   - On MMX registers and XMM registers.
   - If an MMX register is encoded in the reg field of the ModR/M byte.
2. If a memory operand is encoded in the r/m field of the ModR/M byte, REX.X and REX.B may be used for encoding the memory operand.

3. If a general-purpose register is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding and REX.W may be used to encode the 64-bit operand size.

4. If an XMM register operand is encoded in the reg field of the ModR/M byte, REX.R may be used for register encoding. If an XMM register operand is encoded in the r/m field of the ModR/M byte, REX.B may be used for register encoding.

B.5 MMX INSTRUCTION FORMATS AND ENCODINGS

MMX instructions, except the EMMS instruction, use a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

B.5.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-18 shows the encoding of the gg field.

<table>
<thead>
<tr>
<th>gg</th>
<th>Granularity of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Packed Bytes</td>
</tr>
<tr>
<td>01</td>
<td>Packed Words</td>
</tr>
<tr>
<td>10</td>
<td>Packed Doublewords</td>
</tr>
<tr>
<td>11</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg)

When MMX technology registers (mmxreg) are used as operands, they are encoded in the ModR/M byte in the reg field (bits 5, 4, and 3) and/or the R/M field (bits 2, 1, and 0).

If an MMX instruction operates on a general-purpose register (reg), the register is encoded in the R/M field of the ModR/M byte.

B.5.3 MMX Instruction Formats and Encodings Table

Table B-19 shows the formats and encodings of the integer instructions.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMMS – Empty MMX technology state</td>
<td>0000 1111:01110111</td>
</tr>
<tr>
<td>MOVQ – Move quadword</td>
<td></td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:0110 1111:11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:0111 1111:11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>mem to mmxreg</td>
<td>0000 1111:0110 1111: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem from mmxreg</td>
<td>0000 1111:0111 1111: mod mmxreg r/m</td>
</tr>
</tbody>
</table>

**PACKSSDW** – Pack dword to word data (signed with saturation)

| mmxreg2 to mmxreg1              | 0000 1111:0110 1011: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:0110 1011: mod mmxreg r/m         |

**PACKSSWB** – Pack word to byte data (signed with saturation)

| mmxreg2 to mmxreg1              | 0000 1111:0110 0111: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:0110 0111: mod mmxreg r/m         |

**PACKUSWB** – Pack word to byte data (unsigned with saturation)

| mmxreg2 to mmxreg1              | 0000 1111:0110 0111: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:0110 0111: mod mmxreg r/m         |

**PADD** – Add with wrap-around

| mmxreg2 to mmxreg1              | 0000 1111:1111 11gg: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1111 11gg: mod mmxreg r/m         |

**PADDS** – Add signed with saturation

| mmxreg2 to mmxreg1              | 0000 1111:1110 11gg: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1110 11gg: mod mmxreg r/m         |

**PADDUS** – Add unsigned with saturation

| mmxreg2 to mmxreg1              | 0000 1111:1101 11gg: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1101 11gg: mod mmxreg r/m         |

**PAND** – Bitwise And

| mmxreg2 to mmxreg1              | 0000 1111:1101 1011: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1101 1011: mod mmxreg r/m         |

**PANDN** – Bitwise AndNot

| mmxreg2 to mmxreg1              | 0000 1111:1101 1111: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1101 1111: mod mmxreg r/m         |

**PCMPEQ** – Packed compare for equality

| mmxreg1 with mmxreg2            | 0000 1111:0111 01gg: 11 mmxreg1 mmxreg2     |
| mmxreg with memory              | 0000 1111:0111 01gg: mod mmxreg r/m         |

**PCMPGT** – Packed compare greater (signed)

| mmxreg1 with mmxreg2            | 0000 1111:0110 01gg: 11 mmxreg1 mmxreg2     |
| mmxreg with memory              | 0000 1111:0110 01gg: mod mmxreg r/m         |

**PMADDWdW** – Packed multiply add

| mmxreg2 to mmxreg1              | 0000 1111:1111 0101: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1111 0101: mod mmxreg r/m         |

**PMULHUW** – Packed multiplication, store high word (unsigned)

<p>| mmxreg2 to mmxreg1              | 0000 1111:1110 0100: 11 mmxreg1 mmxreg2     |
| memory to mmxreg                | 0000 1111:1110 0100: mod mmxreg r/m         |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMULHW – Packed multiplication, store high word</td>
<td>0000 1111:1110 0101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 0101: mod mmxreg r/m</td>
</tr>
<tr>
<td>PMULLW – Packed multiplication, store low word</td>
<td>0000 1111:1110 0101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 0101: mod mmxreg r/m</td>
</tr>
<tr>
<td>POR – Bitwise Or</td>
<td>0000 1111:1111 0101: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1111 0101: mod mmxreg r/m</td>
</tr>
<tr>
<td>PSLL2 – Packed shift left logical</td>
<td>0000 1111:1111 00gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:1111 00gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:0111 00gg: 11 010 mmxreg: imm8 data</td>
</tr>
<tr>
<td>PSRA2 – Packed shift right arithmetic</td>
<td>0000 1111:1111 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:1111 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:0111 00gg: 11 100 mmxreg: imm8 data</td>
</tr>
<tr>
<td>PSRL2 – Packed shift right logical</td>
<td>0000 1111:1110 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg1 by mmxreg2</td>
<td>0000 1111:1110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg by immediate</td>
<td>0000 1111:0111 00gg: 11 110 mmxreg: imm8 data</td>
</tr>
<tr>
<td>PSUB – Subtract with wrap-around</td>
<td>0000 1111:1111 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:1111 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:1111 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PSUBS – Subtract signed with saturation</td>
<td>0000 1111:1110 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:1110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:1110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PSUBUS – Subtract unsigned with saturation</td>
<td>0000 1111:1110 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 from mmxreg1</td>
<td>0000 1111:1110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>memory from mmxreg</td>
<td>0000 1111:1110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>PUNPCKH – Unpack high data to next larger type</td>
<td>0000 1111:0110 10gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>mmxreg2 to mmxreg1</td>
<td>0000 1111:0110 10gg: mod mmxreg r/m</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 00gg: 11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:0110 00gg: mod mmxreg r/m</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

**B.6 PROCESSOR EXTENDED STATE INSTRUCTION FORMATS AND ENCODINGS**

Table B-20 shows the formats and encodings for several instructions that relate to processor extended state management.

### Table B-20. Formats and Encodings of XSAVE/XRSTOR/XGETBV/XSETBV Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGETBV – Get Value of Extended Control Register</td>
<td>0000 1111:0000 0001: 1101 0000</td>
</tr>
<tr>
<td>XRSTOR – Restore Processor Extended States&lt;sup&gt;1&lt;/sup&gt;</td>
<td>0000 1111:1010 1110: modA 101 r/m</td>
</tr>
<tr>
<td>XSAVE – Save Processor Extended States&lt;sup&gt;1&lt;/sup&gt;</td>
<td>0000 1111:1010 1110: modA 100 r/m</td>
</tr>
<tr>
<td>XSETBV – Set Extended Control Register</td>
<td>0000 1111:0000 0001: 1101 0001</td>
</tr>
</tbody>
</table>

**NOTES:**
1. For XSAVE and XRSTOR, “mod = 11” is reserved.

### Table B-19. MMX Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PXOR – Bitwise Xor</td>
<td>0000 1111:1110 1111:11 mmxreg1 mmxreg2</td>
</tr>
<tr>
<td>memory to mmxreg</td>
<td>0000 1111:1110 1111: mod mmxreg r/m</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The pack instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.
2. The format of the shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

**B.7 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS**

Table B-20 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

### Table B-21. Formats and Encodings of P6 Family Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOVcc – Conditional Move</td>
<td>0000 1111:0100 tttn : 11 reg1 reg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0000 1111 : 0100 tttn : mod reg r/m</td>
</tr>
<tr>
<td>FCMOVcc – Conditional Move on EFLAG Register Condition Codes</td>
<td>11011 010 : 11 000 ST(i)</td>
</tr>
<tr>
<td>move if below (B)</td>
<td>11011 010 : 11 001 ST(i)</td>
</tr>
<tr>
<td>move if equal (E)</td>
<td>11011 010 : 11 010 ST(i)</td>
</tr>
<tr>
<td>move if below or equal (BE)</td>
<td>11011 010 : 11 011 ST(i)</td>
</tr>
<tr>
<td>move if unordered (U)</td>
<td>11011 011 : 11 000 ST(i)</td>
</tr>
<tr>
<td>move if not below (NB)</td>
<td>11011 011 : 11 001 ST(i)</td>
</tr>
<tr>
<td>move if not equal (NE)</td>
<td>11011 011 : 11 010 ST(i)</td>
</tr>
<tr>
<td>move if not below or equal (NBE)</td>
<td>11011 011 : 11 010 ST(i)</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

### B.8 SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables B-22, B-23, and B-24) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. Mandatory prefixes are included in the tables.

### Table B-22. Formats and Encodings of SSE Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS—Add Packed Single Precision Floating-Point Values</td>
<td>0000 1111:0101 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td>ADDSS—Add Scalar Single Precision Floating-Point Values</td>
<td>1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:1100 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td>ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values</td>
<td>0000 1111:0101 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td>ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values</td>
<td>0000 1111:0101 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td>CMPPS—Compare Packed Single Precision Floating-Point Values</td>
<td>0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0000 1111:1100 0010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>CMPSS—Compare Scalar Single Precision Floating-Point Values</td>
<td>1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>1111 0011:0000 1111:1100 0010: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>COMISS—Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS</td>
<td>0000 1111:0010 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0010 1111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### NOTES:

1. For FXSAVE and FXRSTOR, “mod = 11” is reserved.
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTPI2PS—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>0000 1111:0010 1010:11 xmmreg1 mmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0010 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td>CVTPS2PI—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0000 1111:0010 1101:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0010 1101: mod mmreg r/m</td>
</tr>
<tr>
<td>CVTSS2SI—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg</td>
<td>1111 0011:0000 1111:001011010111 xmmreg1 r32</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:00101: mod r32 r/m</td>
</tr>
<tr>
<td>CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0011:0000 1111:0010 1101:11 r32 xmmreg1</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0011:0000 1111:0010 mod r32 r/m</td>
</tr>
<tr>
<td>CVTTPS2PI—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0000 1111:0010 1100:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0010 1100: mod mmreg r/m</td>
</tr>
<tr>
<td>CVTTSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0011:0000 1111:0010 1100:11 r32 xmmreg1</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0011:0000 1111:0010 mod r32 r/m</td>
</tr>
<tr>
<td>DIVPS—Divide Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg</td>
<td>0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>DIVSS—Divide Scalar Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg</td>
<td>1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 mod xmmreg r/m</td>
</tr>
<tr>
<td>LDMXCSR—Load MXCSR Register State</td>
<td></td>
</tr>
<tr>
<td>m32 to MXCSR</td>
<td>0000 1111:1010 1110:mod 010 mem</td>
</tr>
<tr>
<td>MAXPS—Return Maximum Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg</td>
<td>0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>MAXSS—Return Maximum Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg</td>
<td>1111 0011:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>MINPS—Return Minimum Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg</td>
<td>0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td>MINSS—Return Minimum Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg</td>
<td>1111 0011:0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1101: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

#### Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOVAPS</strong>—Move Aligned Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0010 1000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0010 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0000 1111:0010 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0000 1111:0010 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVHLPS</strong>—Move Packed Single Precision Floating-Point Values High to Low</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0010 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td><strong>MOVHPS</strong>—Move High Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0001 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVLHPS</strong>—Move Packed Single Precision Floating-Point Values Low to High</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0010 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td><strong>MOVLPS</strong>—Move Low Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0001 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVMSKPS</strong>—Extract Packed Single Precision Floating-Point Sign Mask</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td><strong>MOVSS</strong>—Move Scalar Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0001 0000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>1111 0011:0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>1111 0011:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVUPS</strong>—Move Unaligned Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0001 0000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0000 1111:0001 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULPS</strong>—Multiply Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0001 0000:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULSS</strong>—Multiply Scalar Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ORPS</strong>—Bitwise Logical OR of Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RCPAPS</strong>—Compute Reciprocals of Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0011:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>
### Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RCPSS</strong>—Compute Reciprocals of Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:1101 0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:1101 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>RSQRTPS</strong>—Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0010: mode xmmreg r/m</td>
</tr>
<tr>
<td><strong>RSQRTSS</strong>—Compute Reciprocals of Square Roots of Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:1101 0010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:1101 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>SHUFPS</strong>—Shuffle Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0000 1111:1101 0110:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0000 1111:1101 0110: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>SQRTPS</strong>—Compute Square Roots of Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>SQRRTSS</strong>—Compute Square Root of Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:1101 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:1101 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>STMXCSR</strong>—Store MXCSR Register State</td>
<td></td>
</tr>
<tr>
<td>MXCSR to mem</td>
<td>0000 1111:1010 1110:mod 011 mem</td>
</tr>
<tr>
<td><strong>SUBPS</strong>—Subtract Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1100:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>SUBSS</strong>—Subtract Scalar Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:1101 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:1101 1100:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>UCOMISS</strong>—Unordered Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>UNPCKHPS</strong>—Unpack and Interleave High Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0001 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0001 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>UNPKLPS</strong>—Unpack and Interleave Low Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0001 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0001 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>XORPS</strong>—Bitwise Logical XOR of Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 0111:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>
### Table B-22. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 0111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### Table B-23. Formats and Encodings of SSE Integer Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAVGB/PAVGW—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 0000:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td></td>
<td>0000 1111:1110 0011:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 0000: mod mmreg r/m</td>
</tr>
<tr>
<td></td>
<td>0000 1111:1110 0011: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PEXTRW—Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32, imm8</td>
<td>0000 1111:1100 0101:11 r32 mmreg: imm8</td>
</tr>
<tr>
<td><strong>PINSRW—Insert Word</strong></td>
<td></td>
</tr>
<tr>
<td>reg32 to mmreg, imm8</td>
<td>0000 1111:1100 0100:11 mmreg r32: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0000 1111:1100 0100: mod mmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMAXSW—Maximum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 1110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 1110: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXUB—Maximum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1101 1110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1101 1110: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMINSW—Minimum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 1010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 1010: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMINUB—Minimum of Packed Unsigned Byte Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1101 1010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1101 1010: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PMOVMSKB—Move Byte Mask To Integer</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to reg32</td>
<td>0000 1111:1101 0111:11 r32 mmreg</td>
</tr>
<tr>
<td><strong>PMULHUW—Multiply Packed Unsigned Integers and Store High Result</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 0100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 0100: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PSADBW—Compute Sum of Absolute Differences</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1110 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1110 0110: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>PSHUFW—Shuffle Packed Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1, imm8</td>
<td>0000 1111:0111 0000:11 mmreg1 mmreg2: imm8</td>
</tr>
<tr>
<td>mem to mmreg, imm8</td>
<td>0000 1111:0111 0000: mod mmreg r/m: imm8</td>
</tr>
</tbody>
</table>
B.9 SSE2 INSTRUCTION FORMATS AND ENCODINGS

The SSE2 instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables show the formats and encodings for the SSE2 SIMD floating-point, SIMD integer, and cacheability instructions, respectively. Some SSE2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

B.9.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-25 shows the encoding of this gg field.

<table>
<thead>
<tr>
<th>gg</th>
<th>Granularity of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Packed Bytes</td>
</tr>
<tr>
<td>01</td>
<td>Packed Words</td>
</tr>
<tr>
<td>10</td>
<td>Packed Doublewords</td>
</tr>
<tr>
<td>11</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

Table B-24. Format and Encoding of SSE Cacheability & Memory Ordering Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASKMOVQ—Store Selected Bytes of Quadword</td>
<td>0000 1111:1111 0111:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>MOVNTPS—Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint</td>
<td>0000 1111:0010 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVNTQ—Store Quadword Using Non-Temporal Hint</td>
<td>0000 1111:1110 0111: mod mmreg r/m</td>
</tr>
<tr>
<td>PREFETCHT0—Prefetch Temporal to All Cache Levels</td>
<td>0000 1111:0001 1000:modA 001 mem</td>
</tr>
<tr>
<td>PREFETCHT1—Prefetch Temporal to First Level Cache</td>
<td>0000 1111:0001 1000:modA 010 mem</td>
</tr>
<tr>
<td>PREFETCHT2—Prefetch Temporal to Second Level Cache</td>
<td>0000 1111:0001 1000:modA 011 mem</td>
</tr>
<tr>
<td>PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels</td>
<td>0000 1111:0001 1000:modA 000 mem</td>
</tr>
<tr>
<td>SFENCE—Store Fence</td>
<td>0000 1111:1010 1110:11 111 000</td>
</tr>
</tbody>
</table>

Table B-25. Encoding of Granularity of Data Field (gg)
### Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADDPD</strong>—Add Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ADDSD</strong>—Add Scalar Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ANDNPD</strong>—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>ANDPD</strong>—Bitwise Logical AND of Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CMPPD</strong>—Compare Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0101 0111: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>CMPSD</strong>—Compare Scalar Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>1111 0010:0000 1111:0101 0111:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>1111 0010:0000 1111:0101 0111: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>COMISD</strong>—Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPI2PD</strong>—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>0110 0110:0000 1111:0101 0110:11 xmmreg1 mmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTPD2PI</strong>—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0110 0110:0000 1111:0101 1101:11 mmreg1 xmmreg1</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0110 0110:0000 1111:0101 1101: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSD2SI</strong>—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1010:11 xmmreg r32</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSI2SD</strong>—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0010:0000 1111:0101 1101:11 xmmreg r32</td>
</tr>
<tr>
<td>mem to r32</td>
<td>1111 0010:0000 1111:0101 1101: mod r32 r/m</td>
</tr>
<tr>
<td><strong>CVTTPD2PI</strong>—Convert with Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>0110 0110:0000 1111:0101 1100:11 mmreg xmmreg</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0110 0110:0000 1111:0101 1100: mod mmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTSD2SI</strong>—Convert with Truncation Scalar Double Precision Floating-Point Value to Doubleword Integer</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>1111 0010:0000 1111:0101 1100:11 xmmreg r32</td>
</tr>
</tbody>
</table>
### Table B-26. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to r32</td>
<td>1111 0010:0000 1111:0010 1100: mod r32 r/m</td>
</tr>
<tr>
<td><strong>CVTPD2PS</strong>—Covert Packed Double Precision Floating-Point Values to Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPS2PD</strong>—Covert Packed Single Precision Floating-Point Values to Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSD2SS</strong>—Covert Scalar Double Precision Floating-Point Value to Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSS2SD</strong>—Covert Scalar Single Precision Floating-Point Value to Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPD2DQ</strong>—Convert Packed Double Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTPD2DQ</strong>—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1110 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTDQ2PD</strong>—Convert Packed Doubleword Integers to Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0111 0011:0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTPS2DQ</strong>—Convert Packed Single Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTTPS2DQ</strong>—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Doubleword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0101 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTDQ2PS</strong>—Convert Packed Doubleword Integers to Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0000 1111:0101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0000 1111:0101 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>DIVPD</strong>—Divide Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>DIVSDD</strong>—Divide Scalar Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MAXPD</strong>—Return Maximum Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MAXSD</strong>—Return Maximum Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MINPD</strong>—Return Minimum Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MINSD</strong>—Return Minimum Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVAPD</strong>—Move Aligned Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0110 0110:0000 1111:0101 1001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0110 0110:0000 1111:0101 1001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVHPD</strong>—Move High Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0110 0110:0000 1111:0001 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVLPD</strong>—Move Low Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0110 0110:0000 1111:0001 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVMSKPD</strong>—Extract Packed Double Precision Floating-Point Sign Mask</td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0110 0110:0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td><strong>MOVSD</strong>—Move Scalar Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>1111 0010:0000 1111:0001 0001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>1111 0010:0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0001 0000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVUPD</strong>—Move Unaligned Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 0001:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>0110 0110:0000 1111:0001 0000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>0110 0110:0000 1111:0001 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MULPD</strong>—Multiply Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1001: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>MULSD—Multiply Scalar Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0001111:01011100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0001111:01011001: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>ORPD—Bitwise Logical OR of Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:1100 1111:0101 1111:0101 1010: mod xmmreg/r/m: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 1111:0101 1111:0101: mod xmmreg/r/m: imm8</td>
</tr>
<tr>
<td><strong>SHUFPD—Shuffle Packed Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1111:0101 1010: mod xmmreg/r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1111:0101 1010: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>SQRTPD—Compute Square Roots of Packed Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 0001: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>SQRTSD—Compute Square Root of Scalar Double Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 0001: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>SUBPD—Subtract Packed Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1100: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>SUBSD—Subtract Scalar Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0010:0000 1111:0101 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0010:0000 1111:0101 1100: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>UCOMISD—Unordered Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0010 1111:0101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0010 1111:0101 1110: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>UNPCKHPD—Unpack and Interleave High Packed Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 1010: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>UNPCLLPD—Unpack and Interleave Low Packed Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0001 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0001 0100: mod xmmreg/r/m</td>
</tr>
<tr>
<td><strong>XORPD—Bitwise Logical OR of Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0101 1111:1101 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0101 1111:1101 1011: mod xmmreg/r/m</td>
</tr>
</tbody>
</table>
## Table B-27. Formats and Encodings of SSE2 Integer Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOVD—Move Doubleword</strong></td>
<td></td>
</tr>
<tr>
<td>reg to xmmreg</td>
<td>0110 0110:0000 1111:0110 1110:11 xmmreg reg</td>
</tr>
<tr>
<td>reg from xmmreg</td>
<td>0110 0110:0000 1111:0111 1110:11 xmmreg reg</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>0110 0110:0000 1111:0111 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVDQA—Move Aligned Double Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>0110 0110:0000 1111:0111 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVDQU—Move Unaligned Double Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0110 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>1111 0011:0000 1111:0111 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVQ2DQ—Move Quadword from MMX to XMM Register</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg to xmmreg</td>
<td>1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>MOVDQ2Q—Move Quadword from XMM to MMX Register</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to mmreg</td>
<td>1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td><strong>MOVQ—Move Quadword</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>1111 0011:0000 1111:0111 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 from xmmreg1</td>
<td>0110 0110:0000 1111:1101 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>1111 0011:0000 1111:0110 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem from xmmreg</td>
<td>0110 0110:0000 1111:1101 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKSSD1—Pack Dword To Word Data (signed with saturation)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:0110 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKSSWB—Pack Word To Byte Data (signed with saturation)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:0110 0011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PACKUSWB—Pack Word To Byte Data (unsigned with saturation)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0110 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:0110 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADDQ—Add Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:1101 0100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:1101 0100: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1101 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1101 0100: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
### Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PADD—Add With Wrap-around</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111: 1111 11gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111: 1111 11gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADD—Add Signed With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111: 1110 11gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111: 1110 11gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PADDUS—Add Unsigned With Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PAND—Bitwise And</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1101 1011: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:1101 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PANDN—Bitwise AndNot</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1101 1111: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:1101 1111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PAVGB—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11100000 mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PAVGW—Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110 0011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1110 0011 mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPEQ—Packed Compare For Equality</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 with xmmreg2</td>
<td>0110 0110:0000 1111:0111 01gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg with memory</td>
<td>0110 0110:0000 1111:0111 01gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PCMPGT—Packed Compare Greater (signed)</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 with xmmreg2</td>
<td>0110 0110:0000 1111:0110 01gg: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg with memory</td>
<td>0110 0110:0000 1111:0110 01gg: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PEXTRW—Extract Word</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to reg32, imm8</td>
<td>0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8</td>
</tr>
<tr>
<td><strong>PINSRW—Insert Word</strong></td>
<td></td>
</tr>
<tr>
<td>reg32 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0100:11 xmmreg r32: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0110 0110:0000 1111:1100 0100: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMADDWĐ—Packed Multiply Add</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1111 0101: 11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>memory to xmmreg</td>
<td>0110 0110:0000 1111:1111 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXWĐ—Maximum of Packed Signed Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>01100110:00001111:11101110: mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>PMAXUB</strong>—Maximum of Packed Unsigned Byte Integers</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>mem to xmmreg 0110 0110:0000 1111:1101 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMINSW</strong>—Minimum of Packed Signed Word Integers</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1110 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>mem to xmmreg 0110 0110:0000 1111:1110 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMINUB</strong>—Minimum of Packed Unsigned Byte Integers</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1101 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>mem to xmmreg 0110 0110:0000 1111:1101 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMOVMSKB</strong>—Move Byte Mask To Integer</td>
<td>xmmreg to reg32 0110 0110:0000 1111:1101 0111:11 r32 xmmreg</td>
</tr>
<tr>
<td><strong>PMULHUW</strong>—Packed multiplication, store high word (unsigned)</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1110 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>memory to xmmreg 0110 0110:0000 1111:1110 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMULHw</strong>—Packed Multiplication, store high word</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1110 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>memory to xmmreg 0110 0110:0000 1111:1110 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMULLw</strong>—Packed Multiplication, store low word</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1101 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>memory to xmmreg 0110 0110:0000 1111:1101 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMULUDQ</strong>—Multiply Packed Unsigned Doubleword Integers</td>
<td>mmreg2 to mmreg1 0000 1111:1111 0100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td></td>
<td>mem to mmreg 0000 1111:1111 0100: mod mmreg r/m</td>
</tr>
<tr>
<td></td>
<td>xmmreg2 to xmmreg1 0110 0110:00001111:1101 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>memory to xmmreg 0110 0110:00001111:1101 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>POR</strong>—Bitwise Or</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1110 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>memory to xmmreg 0110 0110:0000 1111:1110 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSADBW</strong>—Compute Sum of Absolute Differences</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td></td>
<td>mem to xmmreg 0110 0110:0000 1111:1111 0110: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSHUFLLW</strong>—Shuffle Packed Low Words</td>
<td>xmmreg2 to xmmreg1, imm8 1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td></td>
<td>mem to xmmreg, imm8 1111 0010:0000 1111:0111 0000: mod xmmreg r/m; imm8</td>
</tr>
<tr>
<td><strong>PSHUFHLW</strong>—Shuffle Packed High Words</td>
<td>xmmreg2 to xmmreg1, imm8 1111 0011:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td></td>
<td>mem to xmmreg, imm8 1111 0011:0000 1111:0111 0000: mod xmmreg r/m; imm8</td>
</tr>
<tr>
<td><strong>PSHUFDF</strong>—Shuffle Packed Doublewords</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0111 0000: mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>

**PSLLDQ—Shift Double Quadword Left Logical**

- xmmreg, imm8: 0110 0110:0000 1111:0111 0000:11 xmmreg: imm8

**PSLL—Packed Shift Left Logical**

- xmmreg1 by xmmreg2: 0110 0110:0000 1111:1111 00gg: 11 xmmreg1 xmmreg2
- xmmreg by memory: 0110 0110:0000 1111:1111 00gg: mod xmmreg r/m
- xmmreg by immediate: 0110 0110:0000 1111:0111 00gg: 11 110 xmmreg: imm8

**PSRA—Packed Shift Right Arithmetic**

- xmmreg1 by xmmreg2: 0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
- xmmreg by memory: 0110 0110:0000 1111:1101 00gg: mod xmmreg r/m
- xmmreg by immediate: 0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8

**PSRLDQ—Shift Double Quadword Right Logical**

- xmmreg, imm8: 0110 0110:0000 1111:0111 00gg: 11 xmmreg: imm8

**PSRL—Packed Shift Right Logical**

- xmmreg1 by xmmreg2: 0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
- xmmreg by memory: 0110 0110:0000 1111:1101 00gg: mod xmmreg r/m
- xmmreg by immediate: 0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8

**PSUBQ—Subtract Packed Quadword Integers**

- mmreg2 to mmreg1: 0000 1111:1111 011:11 mmreg1 mmreg2
- mem to mmreg: 0000 1111:1111 1011: mod mmreg r/m
- xmmreg2 to xmmreg1: 0110 0110:0000 1111:1111 1011:11 xmmreg1 xmmreg2
- mem to xmmreg: 0110 0110:0000 1111:1111 1011: mod xmmreg r/m

**PSUB—Subtract With Wrap-around**

- xmmreg2 from xmmreg1: 0110 0110:0000 1111:1111 10gg: 11 xmmreg1 xmmreg2
- memory from xmmreg: 0110 0110:0000 1111:1111 10gg: mod xmmreg r/m

**PSUBS—Subtract Signed With Saturation**

- xmmreg2 from xmmreg1: 0110 0110:0000 1111:1110 10gg: 11 xmmreg1 xmmreg2
- memory from xmmreg: 0110 0110:0000 1111:1110 10gg: mod xmmreg r/m

**PSUBUS—Subtract Unsigned With Saturation**

- xmmreg2 from xmmreg1: 0000 1111:1110 10gg: 11 xmmreg1 xmmreg2
- memory from xmmreg: 0000 1111:1110 10gg: mod xmmreg r/m

**PUNPCKH—Unpack High Data To Next Larger Type**

- xmmreg2 to xmmreg1: 0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2
- mem to xmmreg: 0110 0110:0000 1111:1111 0110: mod xmmreg r/m

**PUNPCKHQDQ—Unpack High Data**

- xmmreg2 to xmmreg1: 0110 0110:0000 1111:1110 1101:11 xmmreg1 xmmreg2
- mem to xmmreg: 0110 0110:0000 1111:1110 1101: mod xmmreg r/m
### Table B-27. Formats and Encodings of SSE2 Integer Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUNPCKL</strong>—Unpack Low Data To Next Larger Type</td>
<td>0110 0110:0000 1111:1110 0110:11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110:11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 00gg:11 xeXmmreg r/m</td>
</tr>
<tr>
<td><strong>PUNPCKLQDQ</strong>—Unpack Low Data</td>
<td>0110 0110:0000 1111:1110 1100:11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110 1100:11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0110 1100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PXOR</strong>—Bitwise Xor</td>
<td>0110 0110:0000 1111:1110 1111: 11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1110 1111: 11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:1110 1111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### Table B-28. Format and Encoding of SSE2 Cacheability Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MASKMOVDQU</strong>—Store Selected Bytes of Double Quadword</td>
<td>0110 0110:0000 1111:1111 0111:11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:1111 0111:11 xeXmmreg1 XxeXmmreg2</td>
</tr>
<tr>
<td><strong>CLFLUSH</strong>—Flush Cache Line</td>
<td>0000 1111:1010 1110:11 xeXmmreg r/m</td>
</tr>
<tr>
<td>mem</td>
<td>0000 1111:1010 1110:11 xeXmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVNTPD</strong>—Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint</td>
<td>0110 0110:0000 1111:0010 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0110 0110:0000 1111:0010 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVNTDQ</strong>—Store Double Quadword Using Non-Temporal Hint</td>
<td>0110 0110:0000 1111:1110 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td>xmmreg to mem</td>
<td>0110 0110:0000 1111:1110 0111: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>MOVNTI</strong>—Store Doubleword Using Non-Temporal Hint</td>
<td>0000 1111:1100 0011: mod reg r/m</td>
</tr>
<tr>
<td>reg to mem</td>
<td>0000 1111:1100 0011: mod reg r/m</td>
</tr>
<tr>
<td><strong>PAUSE</strong>—Spin Loop Hint</td>
<td>1111 0011:1001 0000</td>
</tr>
<tr>
<td><strong>LFENCE</strong>—Load Fence</td>
<td>0000 1111:1010 1110:11 101 000</td>
</tr>
<tr>
<td><strong>MFENCE</strong>—Memory Fence</td>
<td>0000 1111:1010 1110:11 110 000</td>
</tr>
</tbody>
</table>
### B.10 SSE3 FORMATS AND ENCODINGS TABLE

The tables in this section provide SSE3 formats and encodings. Some SSE3 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

When in IA-32e mode, use of the REX.R prefix permits instructions that use general purpose and XMM registers to access additional registers. Some instructions require the REX.W prefix to promote the instruction to 64-bit operation. Instructions that require the REX.W prefix are listed (with their opcodes) in Section B.13.

#### Table B-29. Formats and Encodings of SSE3 Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1</td>
<td>01100110:00001111:11010000:11 xmmreg1 xmmreg2 mem to xmmreg</td>
</tr>
<tr>
<td>HADDPD—Add horizontally packed DP FP numbers XMM2/Mem to XMM1</td>
<td>11110010:00001111:01111101:11 xmmreg1 xmmreg2 mem to xmmreg</td>
</tr>
<tr>
<td>HADDPS—Add horizontally packed SP FP numbers XMM2/Mem to XMM1</td>
<td>11110010:00001111:01111100:11 xmmreg1 xmmreg2 mem to xmmreg</td>
</tr>
<tr>
<td>HSUBPD—Sub horizontally packed DP FP numbers XMM2/Mem to XMM1</td>
<td>01100110:00001111:01111101:11 xmmreg1 xmmreg2 mem to xmmreg</td>
</tr>
<tr>
<td>HSUBPS—Sub horizontally packed SP FP numbers XMM2/Mem to XMM1</td>
<td>11110010:00001111:01111101:11 xmmreg1 xmmreg2 mem to xmmreg</td>
</tr>
</tbody>
</table>

#### Table B-30. Formats and Encodings for SSE3 Event Management Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR—Set up a linear address range to be monitored by hardware</td>
<td>0000 1111 : 0000 0001:11 001 000</td>
</tr>
<tr>
<td>MWAIT—Wait until write-back store performed within the range specified by the instruction MONITOR</td>
<td>0000 1111 : 0000 0001:11 001 001</td>
</tr>
</tbody>
</table>

#### Table B-31. Formats and Encodings for SSE3 Integer and Move Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>FISTTP—Store ST in int16 (chop) and pop</td>
<td>m16int 11011 111 : modA 001 r/m</td>
</tr>
<tr>
<td>FISTTP—Store ST in int32 (chop) and pop</td>
<td>m32int 11011 011 : modA 001 r/m</td>
</tr>
<tr>
<td>FISTTP—Store ST in int64 (chop) and pop</td>
<td>11011 011 : modA 001 r/m</td>
</tr>
</tbody>
</table>
B.11 SSSE3 FORMATS AND ENCODING TABLE

The tables in this section provide SSSE3 formats and encodings. Some SSSE3 instructions require a mandatory prefix (66H) as part of the three-byte opcode. These prefixes are included in the table below.

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>m64int</td>
<td>11011 101: modA 001 r/m</td>
</tr>
<tr>
<td>LDDQU—Load unaligned integer 128-bit</td>
<td></td>
</tr>
<tr>
<td>xmm, m128</td>
<td>11110010:00001111:11110000: modA xmmreg r/m</td>
</tr>
<tr>
<td>MOVDDUP—Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110010:00001111:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110010:00001111:00010110:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVSHDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110011:00001111:00010110:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:00010110: mod xmmreg r/m</td>
</tr>
<tr>
<td>MOVSLDUP—Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>11110011:00001111:00010010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>11110011:00001111:00010010: mod xmmreg r/m</td>
</tr>
<tr>
<td>PABSB—Packed Absolute Value Bytes</td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0001 1100: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1100: mod xmmreg r/m</td>
</tr>
<tr>
<td>PABSD—Packed Absolute Value Double Words</td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0001 1110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0001 1110: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1110: mod xmmreg r/m</td>
</tr>
<tr>
<td>PABSW—Packed Absolute Value Words</td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0001 1101:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0001 1101: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0001 1101: mod xmmreg r/m</td>
</tr>
<tr>
<td>PALIGNR—Packed Align Right</td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1, imm8</td>
<td>0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8</td>
</tr>
</tbody>
</table>
### Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to mmreg, imm8</td>
<td>0000 1111:0011 1010:0000 1111: mod mmreg r/m: imm8</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0111 1010:0000 1111:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0111 1010:0000 1111: mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>

**PHADDD—Packed Horizontal Add Double Words**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0101:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0110: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000:0000 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000:0000 0110: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PHADDSW—Packed Horizontal Add and Saturate**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0111:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0111: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000:0000 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000:0000 0111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PHADDW—Packed Horizontal Add Words**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0101:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0101: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000:0000 0101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000:0000 0101: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PHSUBD—Packed Horizontal Subtract Double Words**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0110:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0110: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000:0000 0110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000:0000 0110: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PHSUBSW—Packed Horizontal Subtract and Saturate**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0111:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0111: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000:0000 0111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000:0000 0111: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PHSUBW—Packed Horizontal Subtract Words**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000:0000 0101:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000:0000 0101: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000:0000 0101:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>
### Table B-32. Formats and Encodings for SSSE3 Instructions (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0000 0100:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0000 0100: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0000 0100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 0100: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMULHRSW—Packed Multiply Hi gn with Round and Scale</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0000 1011:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0000 1011: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1011:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1011: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSHUFB—Packed Shuffle Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0000 0000:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0000 0000: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0000 0000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 0000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSIGNB—Packed Sign Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0000 1000:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0000 1000: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1000: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSIGND—Packed Sign Double Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0000 1010:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0000 1010: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1010:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1010: mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PSIGNW—Packed Sign Words</strong></td>
<td></td>
</tr>
<tr>
<td>mmreg2 to mmreg1</td>
<td>0000 1111:0011 1000: 0000 1001:11 mmreg1 mmreg2</td>
</tr>
<tr>
<td>mem to mmreg</td>
<td>0000 1111:0011 1000: 0000 1001: mod mmreg r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0000 1001: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

### B.12 AESNI AND PCLMULQDQ INSTRUCTION FORMATS AND ENCODINGS

Table B-33 shows the formats and encodings for AESNI and PCLMULQDQ instructions.
B.13 SPECIAL ENCODINGS FOR 64-BIT MODE

The following Pentium, P6, MMX, SSE, SSE2, SSE3 instructions are promoted to 64-bit operation in IA-32e mode by using REX.W. However, these entries are special cases that do not follow the general rules (specified in Section B.4).

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESDEC—Perform One Round of an AES Decryption Flow</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0011 1000:1101 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>AESDECLAST—Perform Last Round of an AES Decryption Flow</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0011 1000:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>AESENCLAST—Perform Last Round of an AES Encryption Flow</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0011 1000:1101 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>AESIMC—Perform the AES InvMixColumn Transformation</td>
<td>xmmreg2 to xmmreg1 0110 0110:0000 1111:0011 1010:1101 1111:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOVcc—Conditional Move</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>register2 to register1</td>
<td>0100 0R0B 0000 1111:0100 ttn:11 reg1 reg2</td>
</tr>
<tr>
<td>qwordregister2 to qwordregister1</td>
<td>0100 1R0B 0000 1111:0100 ttn:11 qwordreg1 qwordreg2</td>
</tr>
<tr>
<td>memory to register</td>
<td>0100 0RXB 0000 1111:0100 ttn:mod reg r/m</td>
</tr>
<tr>
<td>memory64 to qwordregister</td>
<td>0100 1RXB 0000 1111:0100 ttn:mod qwordreg r/m</td>
</tr>
<tr>
<td><strong>CVTSD2SI—Convert Scalar Double Precision Floating-Point Value to Doubleword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0010:0000 1111:0010 1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1101:11 r64 xmmreg</td>
</tr>
<tr>
<td>mem64 to r32</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 1101:mod r32 r/m</td>
</tr>
<tr>
<td>mem64 to r64</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1101:mod r64 r/m</td>
</tr>
<tr>
<td><strong>CVTSI2SS—Convert Doubleword Integer to Scalar Single Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>0100 0R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r32</td>
</tr>
<tr>
<td>r64 to xmmreg1</td>
<td>0100 1R0B 1111 0011:0000 1111:0010 1010:11 xmmreg r64</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0100 0RXB 1111 0011:0000 1111:0010 1010:mod xmmreg r/m</td>
</tr>
<tr>
<td>mem64 to xmmreg</td>
<td>0100 1RXB 1111 0011:0000 1111:0010 1010:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSI2SD—Convert Doubleword Integer to Scalar Double Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>r32 to xmmreg1</td>
<td>0100 0R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r32</td>
</tr>
<tr>
<td>r64 to xmmreg1</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1010:11 xmmreg r64</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 1010:mod xmmreg r/m</td>
</tr>
<tr>
<td>mem64 to xmmreg</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1010:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>CVTSS2SI—Convert Scalar Single Precision Floating-Point Value to Doubleword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0011:0000 1111:0010 1101:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0011:0000 1111:0010 1101:11 r64 xmmreg</td>
</tr>
<tr>
<td>mem to r32</td>
<td>0100 0RXB 11110011:0001111:00101101:mod r32 r/m</td>
</tr>
<tr>
<td>mem32 to r64</td>
<td>0100 1RXB 1111 0011:0000 1111:0010 1101:mod r64 r/m</td>
</tr>
<tr>
<td><strong>CVTTSD2SI—Convert with Truncation Scalar Double Precision Floating-Point Value to Doubleword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 11110010:00001111:00101100:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0010:0000 1111:0010 1100:11 r64 xmmreg</td>
</tr>
<tr>
<td>mem64 to r32</td>
<td>0100 0RXB 1111 0010:0000 1111:0010 1100:mod r32 r/m</td>
</tr>
<tr>
<td>mem64 to r64</td>
<td>0100 1RXB 1111 0010:0000 1111:0010 1100:mod r64 r/m</td>
</tr>
</tbody>
</table>
### Table B-34. Special Case Instructions Promoted Using REX.W (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CVTSS2SI—Convert with Truncation Scalar Single Precision Floating-Point Value to Doubleword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 1111 0011:0000 1111:0010 1100:11 r32 xmmreg1</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 1111 0011:0000 1111:0010 1100:11 r64 xmmreg1</td>
</tr>
<tr>
<td>mem to r32</td>
<td>0100 0RXB 1111 0011:0000 1111:0010 1100:0 mod r32 r/m</td>
</tr>
<tr>
<td>mem32 to r64</td>
<td>0100 1RXB 1111 0011:0000 1111:0010 1100: mod r64 r/m</td>
</tr>
</tbody>
</table>

**MOV/MOVQ—Move doubleword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg to mmxreg</td>
<td>0100 0R0B 0000 1111:0110 1110:11 mmxreg reg</td>
</tr>
<tr>
<td>qwordreg to mmxreg</td>
<td>0100 1R0B 0000 1111:0110 1110:11 mmxreg qwordreg</td>
</tr>
<tr>
<td>reg from mmxreg</td>
<td>0100 0R0B 0000 1111:0111 1110:11 mmxreg reg</td>
</tr>
<tr>
<td>qwordreg from mmxreg</td>
<td>0100 1R0B 0000 1111:0111 1110:11 mmxreg qwordreg</td>
</tr>
<tr>
<td>mem to mmxreg</td>
<td>0100 0RXB 0000 1111:0110 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem64 to mmxreg</td>
<td>0100 1RXB 0000 1111:0110 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem from mmxreg</td>
<td>0100 0RXB 0000 1111:0111 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mem64 from mmxreg</td>
<td>0100 1RXB 0000 1111:0111 1110: mod mmxreg r/m</td>
</tr>
<tr>
<td>mmxreg with memory</td>
<td>0100 0RXB 0000 1111:0110 01gg mod mmxreg r/m</td>
</tr>
</tbody>
</table>

**MOVMSKPS—Extract Packed Single Precision Floating-Point Sign Mask**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg to r32</td>
<td>0100 0R0B 0000 1111:0101 0000:11 r32 xmmreg</td>
</tr>
<tr>
<td>xmmreg to r64</td>
<td>0100 1R0B 0000 1111:0101 0000:11 r64 xmmreg</td>
</tr>
</tbody>
</table>

**PEXTRw—Extract Word**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg to reg32, imm8</td>
<td>0100 0R0B 0000 1111:1100 0101:11 r32 mmreg: imm8</td>
</tr>
<tr>
<td>mmreg to reg64, imm8</td>
<td>0100 1R0B 0000 1111:1100 0101:11 r64 mmreg: imm8</td>
</tr>
<tr>
<td>xmmreg to reg32, imm8</td>
<td>0100 0R0B 0110 0110 0000 1111:1100 0101:11 r32 xmmreg: imm8</td>
</tr>
<tr>
<td>xmmreg to reg64, imm8</td>
<td>0100 1R0B 0110 0110 0000 1111:1100 0101:11 r64 xmmreg: imm8</td>
</tr>
</tbody>
</table>

**PINSRW—Insert Word**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg32 to mmreg, imm8</td>
<td>0100 0R0B 0000 1111:1100 0100:11 mmreg r32: imm8</td>
</tr>
<tr>
<td>reg64 to mmreg, imm8</td>
<td>0100 1R0B 0000 1111:1100 0100:11 mmreg r64: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0100 0R0B 0000 1111:1100 0100 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td>m16 to mmreg, imm8</td>
<td>0100 1RXB 0000 1111:11000100 mod mmreg r/m: imm8</td>
</tr>
<tr>
<td>reg32 to xmmreg, imm8</td>
<td>0100 0RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r32: imm8</td>
</tr>
<tr>
<td>reg64 to xmmreg, imm8</td>
<td>0100 1RXB 0110 0110 0000 1111:1100 0100:11 xmmreg r64: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0100 0RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>m16 to xmmreg, imm8</td>
<td>0100 1RXB 0110 0110 0000 1111:1100 0100 mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>

**PMOVMSKB—Move Byte Mask To Integer**
### B.14 SSE4.1 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.1 formats and encodings. Some SSE4.1 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables.

In 64-bit mode, some instructions require REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

#### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmreg to reg32</td>
<td>0100 0RX8 0000 1111:1101 0111:11 r32 mmreg</td>
</tr>
<tr>
<td>mmreg to reg64</td>
<td>0100 1R08 0000 1111:1101 0111:11 r64 mmreg</td>
</tr>
<tr>
<td>xmmreg to reg32</td>
<td>0100 0RX8 0110 0110 0000 1111:1101 0111:11 r32 mmreg</td>
</tr>
<tr>
<td>xmmreg to reg64</td>
<td>0110 0110 0000 1111:1101 0111:11 r64 xmmreg</td>
</tr>
</tbody>
</table>

#### BLENDPD — Blend Packed Double Precision Floats

| xmmreg2 to xmmreg1     | 0110 0110:0000 1111:0011 1010:0000 1101:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0110 0110:0000 1111:0011 1010:0000 1101:00 0011:11 xmmreg r/m |

#### BLENDPS — Blend Packed Single Precision Floats

| xmmreg2 to xmmreg1     | 0110 0110:0000 1111:0011 1010:0000 1100:11 xmmreg1 xmmreg2 |
| mem to xmmreg          | 0110 0110:0000 1111:0011 1010:0000 1100:00 0011:11 xmmreg r/m |

#### BLENDVPD — Variable Blend Packed Double Precision Floats

| xmmreg2 to xmmreg1 <xmm0> | 0110 0110:0000 1111:0011 1000:0001 0101:11 xmmreg1 xmmreg2 |
| mem to xmmreg <xmm0>      | 0110 0110:0000 1111:0011 1000:0001 0100:11 xmmreg r/m |

#### BLENDVPS — Variable Blend Packed Single Precision Floats

| xmmreg2 to xmmreg1 <xmm0> | 0110 0110:0000 1111:0011 1000:0001 0100:11 xmmreg1 xmmreg2 |
| mem to xmmreg <xmm0>      | 0110 0110:0000 1111:0011 1000:0001 0100:00 0011:11 xmmreg r/m |

#### DPPD — Packed Double Precision Dot Products

| xmmreg2 to xmmreg1, imm8 | 0110 0110:0000 1111:0011 1010:0100 0001:11 xmmreg1 xmmreg2: imm8 |
| mem to xmmreg, imm8       | 0110 0110:0000 1111:0011 1010:0100 0001:00 0001:11 xmmreg r/m: imm8 |

#### DPPS — Packed Single Precision Dot Products

| xmmreg2 to xmmreg1, imm8 | 0110 0110:0000 1111:0011 1010:0100 0000:11 xmmreg1 xmmreg2: imm8 |
| mem to xmmreg, imm8       | 0110 0110:0000 1111:0011 1010:0100 0000:00 0000:11 xmmreg r/m: imm8 |

#### EXTRACTPS — Extract From Packed Single Precision Floats

| reg from xmmreg, imm8     | 0110 0110:0000 1111:0011 1010:0001 0111:11 xmmreg reg: imm8 |
### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mem from xmmreg, imm8</strong></td>
<td>0110 0110:0000 1111:0011 1010:0001 0111:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>INSERTPS — Insert Into Packed Single Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0010 0001:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>MOVNTDQA — Load Double Quadword Non-temporal Aligned</strong></td>
<td></td>
</tr>
<tr>
<td>m128 to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0100 1010:11 r/m xmmreg2</td>
</tr>
<tr>
<td><strong>MPSADBW — Multiple Packed Sums of Absolute Difference</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0100 0010:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PACKUSDW — Pack with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0010 1001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0010 1001:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PBLENDVB — Variable Blend Packed Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 &lt;xmm0&gt;</td>
<td>0110 0110:0000 1111:0011 1000:0001 0000:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg &lt;xmm0&gt;</td>
<td>0110 0110:0000 1111:0011 1000:0000 0000:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PBLENDW — Blend Packed Words</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 1110:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1000:0100 1110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PCMPEQQ — Compare Packed Qword Data of Equal</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000:0100 1110:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0100 1110:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PEXTRB — Extract Byte</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0100:11 xmmreg reg: imm8</td>
</tr>
<tr>
<td>xmmreg to mem, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000 1110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PEXTRD — Extract DWord</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg to mem, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0110:mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>
### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>r64 from xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>m64 from xmmreg, imm8</td>
<td>0110 0110:REX.W:0000 1111:0011 1010:0001 0110:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>reg from xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>mem from xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0001 0101:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PHMINPOSUW — Packed Horizontal Word Minimum</strong></td>
<td>0110 0110:0000 1111:0011 1000:0100 0001:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0100 0001:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PINSRB — Extract Byte</strong></td>
<td>0110 0110:0000 1111:0011 1010:0010 0000:11 xmmreg reg: imm8</td>
</tr>
<tr>
<td>reg to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0010 0000:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>reg to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0010 0010:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>r64 to xmmreg, imm8</td>
<td></td>
</tr>
<tr>
<td>m64 to xmmreg, imm8</td>
<td>0110 0110:REX.W:0000 1111:0011 1010:0010 0010:mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td><strong>PMAXSB — Maximum of Packed Signed Byte Integers</strong></td>
<td>0110 0110:0000 1111:0011 1000:0011 1100:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1100:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXSD — Maximum of Packed Signed Dword Integers</strong></td>
<td>0110 0110:0000 1111:0011 1000:0011 1101:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1101:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXUD — Maximum of Packed Unsigned Dword Integers</strong></td>
<td>0110 0110:0000 1111:0011 1000:0011 1111:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000:0011 1111:mod xmmreg r/m</td>
</tr>
<tr>
<td><strong>PMAXUW — Maximum of Packed Unsigned Word Integers</strong></td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMINSB — Minimum of Packed Signed Byte Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMINSD — Minimum of Packed Signed Dword Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMINUd — Minimum of Packed Unsigned Dword Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMINuw — Minimum of Packed Unsigned Word Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMOVSBBD — Packed Move Sign Extend - Byte to Dword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMOVSBQ — Packed Move Sign Extend - Byte to Qword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMOVSBW — Packed Move Sign Extend - Byte to Word**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMOVSWD — Packed Move Sign Extend - Word to Dword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMOVSWQ — Packed Move Sign Extend - Word to Qword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PMOVSDQ — Packed Move Sign Extend - Dword to Qword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1000: 0011 1110:00 mod xmmreg r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0010 0101: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZXBBD — Packed Move Zero Extend - Byte to Dword</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0011:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0010:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0000:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZXBQ — Packed Move Zero Extend - Byte to Qword</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0011:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0010:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0000:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZXBBW — Packed Move Zero Extend - Byte to Word</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0011:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0010:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0000:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZXWD — Packed Move Zero Extend - Word to Dword</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZXWQ — Packed Move Zero Extend - Word to Qword</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMOVZXDQ — Packed Move Zero Extend - Dword to Qword</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0011 0100:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMULDQ — Multiply Packed Signed Dword Integers</td>
<td>0110 0110:0000 1111:0111 1000: 0100 0000:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0100 0000:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0100 0000:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PMULLD — Multiply Packed Signed Dword Integers, Store low Result</td>
<td>0110 0110:0000 1111:0111 1000: 0100 0000:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0100 0000:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0100 0000:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>PTEST — Logical Compare</td>
<td>0110 0110:0000 1111:0111 1000: 0001 0111:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>0110 0110:0000 1111:0111 1000: 0001 0111:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0111 1000: 0001 0111:11: mod xmmreg r/m</td>
</tr>
<tr>
<td>ROUNDPD — Round Packed Double Precision Values</td>
<td>0110 0110:0000 1111:0111 1010: 0000 1001:11 xmmreg1</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0111 1010: 0000 1001:11 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0111 1010: 0000 1001:11: mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>
### B.15 SSE4.2 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.2 formats and encodings. Some SSE4.2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables. In 64-bit mode, some instructions require REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

#### Table B-35. Encodings of SSE4.1 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0000:1000: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>ROUNDSD — Round Scalar Double Precision Value</td>
<td>0110 0110:0000 1111:0011 1010:0111:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0111: mod xmmreg r/m: imm8</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0101: mod xmmreg r/m: imm8</td>
</tr>
</tbody>
</table>

#### Table B-36. Encodings of SSE4.2 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC32 — Accumulate CRC32</td>
<td>1111 0010:0000 1111:0011 1000: 1111 000w :11 reg1 reg2</td>
</tr>
<tr>
<td>reg2 to reg1</td>
<td>1111 0010:0000 1111:0011 1000: 1111 000w : mod reg r/m</td>
</tr>
<tr>
<td>mem to reg</td>
<td>1111 0010:0000 1111:0011 1000: 1111 000w : mod reg r/m</td>
</tr>
<tr>
<td>bytereg2 to reg1</td>
<td>1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 :11 reg1 bytereg2</td>
</tr>
<tr>
<td>m8 to reg</td>
<td>1111 0010:0100 WR0B:0000 1111:0011 1000: 1111 0000 : mod reg r/m</td>
</tr>
<tr>
<td>qwreg2 to qwreg1</td>
<td>1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 :11 qwreg1 qwreg2</td>
</tr>
<tr>
<td>mem64 to qwreg</td>
<td>1111 0010:0100 1R0B:0000 1111:0011 1000: 1111 0001 : mod qwreg r/m</td>
</tr>
<tr>
<td>PCMPESTRI— Packed Compare Explicit-Length Strings To Index</td>
<td>0110 0110:0000 1111:0011 1010:0110 0001:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0110 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010:0110 0001: mod xmmreg r/m</td>
</tr>
<tr>
<td>PCMPESTRM— Packed Compare Explicit-Length Strings To Mask</td>
<td>0110 0110:0000 1111:0011 1010:0110 0000:11 xmmreg1 xmmreg2: imm8</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>0110 0110:0000 1111:0011 1010:0110 0000: mod xmmreg r/m</td>
</tr>
</tbody>
</table>
INSTRUCTION FORMATS AND ENCODINGS

Table B-36. Encodings of SSE4.2 instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg</td>
<td>0110 0110:0000 1111:0011 1010:0110 0000: mod xmmreg r/m</td>
</tr>
</tbody>
</table>

**PCMPISTRI— Packed Compare Implicit-Length String To Index**

- xmmreg2 to xmmreg1, imm8
  - Hex: 0110 0110:0000 1111:0011 1010:0110 0011:11 xmmreg1
  - Hex: 0110 0110:0000 1111:0011 1010:0010:11 xmmreg1

**PCMPISTRM— Packed Compare Implicit-Length Strings To Mask**

- mem to xmmreg
  - Hex: 0110 0110:0000 1111:0011 1010:0110 0010:11 xmmreg1

**PCMPGTQ— Packed Compare Greater Than**

- xmmreg to xmmreg
  - Hex: 0110 0110:0000 1111:0011 1000:0011 0111:11 xmmreg1

**POPCNT— Return Number of Bits Set to 1**

- reg2 to reg1
  - Hex: 1111 0011:0000 1111:1011 1000:11 reg1 reg2

B.16 AVX FORMATS AND ENCODING TABLE

The tables in this section provide AVX formats and encodings. A mixed form of bit/hex/symbolic forms are used to express the various bytes:

The C4/C5 and opcode bytes are expressed in hex notation; the first and second payload byte of VEX, the modR/M byte is expressed in combination of bit/symbolic form. The first payload byte of C4 is expressed as combination of bits and hex form, with the hex value preceded by an underscore. The VEX bit field to encode upper register 8-15 uses 1’s complement form, each of those bit field is expressed as lower case notation rxb, instead of RXB.

The hybrid bit-nibble-byte form is depicted below:

![Figure B-2. Hybrid Notation of VEX-Encoded Key Instruction Bytes](image-url)
Table B-37. Encodings of AVX instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VBLENDPD — Blend Packed Double Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0D:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0D:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0D:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0D:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VBLENDPS — Blend Packed Single Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0C:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:0C:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0C:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:0C:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VBLENDVPD — Variable Blend Packed Double Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4B:11 xmmreg1 xmmreg3: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4B:mod xmmreg1 r/m: xmmreg4</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4B:11 ymmreg1 ymmreg3: ymmreg4</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4B:mod ymmreg1 r/m: ymmreg4</td>
</tr>
<tr>
<td><strong>VBLENDVPS — Variable Blend Packed Single Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4A:11 xmmreg1 xmmreg3: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using xmmreg4 as mask</td>
<td>C4: rxb0_3: 0 xmmreg2 001:4A:mod xmmreg1 r/m: xmmreg4</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4A:11 ymmreg1 ymmreg3: ymmreg4</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1 using ymmreg4 as mask</td>
<td>C4: rxb0_3: 0 ymmreg2 101:4A:mod ymmreg1 r/m: ymmreg4</td>
</tr>
<tr>
<td><strong>VDPPD — Packed Double Precision Dot Products</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:41:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:41:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VDPPS — Packed Single Precision Dot Products</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:40:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:40:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 into ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:40:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_3: w ymmreg2 101:40:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VEXTRACTPS — Extract From Packed Single Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rxb0_3: w_F 001:17:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rxb0_3: w_F 001:17:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VINSERTPS — Insert Into Packed Single Precision Floats</strong></td>
<td></td>
</tr>
<tr>
<td>use imm to merge xmmreg3 with xmmreg2 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:21:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>use imm to merge mem with xmmreg2 into xmmreg1</td>
<td>C4: rxb0_3: w xmmreg2 001:21:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VMOVNTDQA — Load Double Quadword Non-temporal Aligned</strong></td>
<td></td>
</tr>
<tr>
<td>m128 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:2A:11 xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>VMPSADBW</strong> — Multiple Packed Sums of Absolute Difference</td>
<td>C4: rb0_3: w xmmreg2 001:42:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg3 with xmmreg2 into xmmreg1</td>
<td>C4: rb0_3: w xmmreg2 001:42:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>m128 with xmmreg2 into xmmreg1</td>
<td>C4: rb0_3: w xmmreg2 001:42:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPACKUSDW</strong> — Pack with Unsigned Saturation</td>
<td>C4: rb0_2: w xmmreg2 001:2B:11 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg3 and xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:2B:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>m128 and xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:2B:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPBLENDVB</strong> — Variable Blend Packed Bytes</td>
<td>C4: rb0_3: w xmmreg2 001:4C:11 xmmreg1 xmmreg3: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1 using xmmreg4 as mask</td>
<td>C4: rb0_3: w xmmreg2 001:4C:mod xmmreg1 r/m: xmmreg4</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using xmmreg4 as mask</td>
<td>C4: rb0_3: w xmmreg2 001:4C:mod xmmreg1 r/m: xmmreg4</td>
</tr>
<tr>
<td><strong>VPBLENDW</strong> — Blend Packed Words</td>
<td>C4: rb0_3: w xmmreg2 001:0E:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_3: w xmmreg2 001:0E:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_3: w xmmreg2 001:0E:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPCMPEQQ</strong> — Compare Packed Qword Data of Equal</td>
<td>C4: rb0_2: w xmmreg2 001:29:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:29:mod xmmreg1 r/m:</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:29:mod xmmreg1 r/m:</td>
</tr>
<tr>
<td><strong>VPEXTRB</strong> — Extract Byte</td>
<td>C4: rb0_3: 0_F 001:14:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:14:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:14:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPEXTRD</strong> — Extract DWord</td>
<td>C4: rb0_3: 0_F 001:16:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:16:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:16:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPEXTRQ</strong> — Extract QWord</td>
<td>C4: rb0_3: 1_F 001:16:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 1_F 001:16:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 1_F 001:16:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPEXTRW</strong> — Extract Word</td>
<td>C4: rb0_3: 0_F 001:15:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>reg from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:15:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>mem from xmmreg1 using imm</td>
<td>C4: rb0_3: 0_F 001:15:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPHMINPOSUW</strong> — Packed Horizontal Word Minimum</td>
<td>C4: rb0_2: w_F 001:41:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w_F 001:41:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: w_F 001:41:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPINSRB</strong> — Insert Byte</td>
<td>C4: rb0_3: 0_F 001:20:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>reg with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0_F 001:20:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>mem with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0_F 001:20:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPINSRD</strong> — Insert Dword</td>
<td>C4: rb0_3: 0_F 001:22:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>reg with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0_F 001:22:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>mem with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 0_F 001:22:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPINSRQ</strong> — Insert Qword</td>
<td>C4: rb0_3: 1_F 001:22:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>r64 with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: 1_F 001:22:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>m64 with xmmreg2 to xmmreg1, imm8</td>
<td>C4: rxb0_3: 1 xmmreg2 001:22:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPMAXSB</strong> — Maximum of Packed Signed Byte Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXSD</strong> — Maximum of Packed Signed Dword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXUD</strong> — Maximum of Packed Unsigned Dword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINUW</strong> — Minimum of Packed Unsigned Word Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINSB</strong> — Minimum of Packed Signed Byte Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:38:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:38:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINSD</strong> — Minimum of Packed Signed Dword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:39:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:39:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINUW</strong> — Minimum of Packed Unsigned Dword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 into xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3B:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:3B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBD</strong> — Packed Move Sign Extend - Byte to Dword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:21:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:21:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBQ</strong> — Packed Move Sign Extend - Byte to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:22:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:22:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXBw</strong> — Packed Move Sign Extend - Byte to Word</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:20:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:20:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXwD</strong> — Packed Move Sign Extend - Word to Dword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:23:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:23:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVSXwQ</strong> — Packed Move Sign Extend - Word to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_2: w_F 001:24:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>
## Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:24:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVSXQ</strong> — Packed Move Sign - Dword to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:25:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:25:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVXBD</strong> — Packed Move Zero - Byte to Dword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:31:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:31:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVXBQ</strong> — Packed Move Zero - Byte to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:32:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:32:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVXBW</strong> — Packed Move Zero - Byte to Word</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:30:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:30:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVXWD</strong> — Packed Move Zero - Word to Dword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:33:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:33:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVXWQ</strong> — Packed Move Zero - Word to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:34:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:34:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVVSXDQ</strong> — Packed Move Sign - Dword to Qword</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:35:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: r0x0_2: w 001:35:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMULDQ</strong> — Multiply Packed Signed Dword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: r0x0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: r0x0_2: w xmmreg2 001:28:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMULLD</strong> — Multiply Packed Signed Dword Integers, Store low Result</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: r0x0_2: w xmmreg2 001:40:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: r0x0_2: w xmmreg2 001:40:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPTEST</strong> — Logical Compare</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: r0x0_2: w 001:17:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg</td>
<td>C4: r0x0_2: w 001:17:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: r0x0_2: w 101:17:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg</td>
<td>C4: r0x0_2: w 101:17:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VROUNDPD</strong> — Round Packed Double Precision Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>C4: r0x0_3: w 001:09:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm8</td>
<td>C4: r0x0_3: w 001:09:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm8</td>
<td>C4: r0x0_3: w 101:09:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm8</td>
<td>C4: r0x0_3: w 101:09:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VROUNDPS</strong> — Round Packed Single Precision Values</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>C4: rxb0_3: w_F 001:08:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm8</td>
<td>C4: rxb0_3: w_F 001:08:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm8</td>
<td>C4: rxb0_3: w_F 101:08:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm8</td>
<td>C4: rxb0_3: w_F 101:08:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VROUNDSD — Round Scalar Double Precision Value**

| xmmreg2 and xmmreg3 to xmmreg1, imm8 | C4: rxb0_3: w xmmreg2 001:0B:11 xmmreg1 xmmreg3: imm |
| mem to xmmreg1, imm8  | C4: rxb0_3: w xmmreg2 001:0B:mod xmmreg1 r/m: imm |

**VROUNDSS — Round Scalar Single Precision Value**

| xmmreg2 and xmmreg3 to xmmreg1, imm8 | C4: rxb0_3: w xmmreg2 001:0A:11 xmmreg1 xmmreg3: imm |
| mem to xmmreg1, imm8  | C4: rxb0_3: w xmmreg2 001:0A:mod xmmreg1 r/m: imm |

**VPCMPGSTRI — Packed Compare Explicit Length Strings, Return Index**

| xmmreg2 with xmmreg1, imm8 | C4: rxb0_3: w_F 001:61:11 xmmreg1 xmmreg2: imm |
| mem with xmmreg1, imm8  | C4: rxb0_3: w_F 001:61:mod xmmreg1 r/m: imm |

**VPCMPGSTRM — Packed Compare Explicit Length Strings, Return Mask**

| xmmreg2 with xmmreg1, imm8 | C4: rxb0_3: w_F 001:60:11 xmmreg1 xmmreg2: imm |
| mem with xmmreg1, imm8  | C4: rxb0_3: w_F 001:60:mod xmmreg1 r/m: imm |

**VPCMPGTQ — Compare Packed Data for Greater Than**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_2: w xmmreg2 001:28:11 xmmreg1 xmmreg3 |
| mem with xmmreg2 to xmmreg1 | C4: rxb0_2: w xmmreg2 001:28:mod xmmreg1 r/m |

**VPCMPISTRI — Packed Compare Implicit Length Strings, Return Index**

| xmmreg2 with xmmreg1, imm8 | C4: rxb0_3: w_F 001:63:11 xmmreg1 xmmreg2: imm |
| mem with xmmreg1, imm8  | C4: rxb0_3: w_F 001:63:mod xmmreg1 r/m: imm |

**VPCMPISTRM — Packed Compare Implicit Length Strings, Return Mask**

| xmmreg2 with xmmreg1, imm8 | C4: rxb0_3: w_F 001:62:11 xmmreg1 xmmreg2: imm |
| mem with xmmreg1, imm8  | C4: rxb0_3: w_F 001:62:mod xmmreg1 r/m: imm |

**VAESDEC — Perform One Round of an AES Decryption Flow**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DE:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DE:mod xmmreg1 r/m |

**VAESDECLAST — Perform Last Round of an AES Decryption Flow**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DF:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DF:mod xmmreg1 r/m |

**VAESENCE — Perform One Round of an AES Encryption Flow**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DC:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DC:mod xmmreg1 r/m |

**VAESENCLAST — Perform Last Round of an AES Encryption Flow**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DD:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_2: w xmmreg2 001:DD:mod xmmreg1 r/m |

**VAESIMC — Perform the AES InvMixColumn Transformation**

<p>| xmmreg2 to xmmreg1 | C4: rxb0_2: w_F 001:DB:11 xmmreg1 xmmreg2 |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: w_F 001:DB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VAESKEYGENASSIST — AES Round Key Generation Assist</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm8</td>
<td>C4: rb0_3: w_F 001:DF:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg, imm8</td>
<td>C4: rb0_3: w_F 001:DF:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPABSB — Packed Absolute Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w_F 001:1C:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: w_F 001:1C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPABSD — Packed Absolute Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w_F 001:1E:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: w_F 001:1E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPABSW — Packed Absolute Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: w_F 001:1D:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: w_F 001:1D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPALIGNR — Packed Align Right</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1, imm8</td>
<td>C4: rb0_3: w xmmreg2 001:DD:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1, imm8</td>
<td>C4: rb0_3: w xmmreg2 001:DD:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPHADDD — Packed Horizontal Add</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:02:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:02:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPHADDW — Packed Horizontal Add</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:01:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:01:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPHADDSW — Packed Horizontal Add and Saturate</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:03:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:03:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPHSUBD — Packed Horizontal Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:06:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:06:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPHSUBW — Packed Horizontal Subtract</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:05:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:05:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPHSUBSW — Packed Horizontal Subtract and Saturate</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:07:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:07:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:04:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:04:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMULHRSW — Packed Multiply High with Round and Scale</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: w xmmreg2 001:0B:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:0B:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPShufB — Packed Shuffle Bytes</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:00:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:00:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPSignB — Packed SIGN</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:08:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:08:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPSignD — Packed SIGN</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:0A:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:0A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VPSignW — Packed SIGN</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:09:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_2: w xmmreg2 001:09:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VaddsSubPD — Packed Double-FP Add/Subtract</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D0:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D0:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:D0:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:D0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r ymmreglo2 101:D0:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r ymmreglo2 101:D0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VaddsSubPS — Packed Single-FP Add/Subtract</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:D0:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:D0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 011:D0:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 011:D0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:D0:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:D0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r ymmreglo2 111:D0:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r ymmreglo2 111:D0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VhaddsPD — Packed Double-FP Horizontal Add</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:7C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:7C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:7C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:7C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:7C:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:7C:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r ymmreglo2 101:7C:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:7Cmod ymmreg1 r/m</td>
</tr>
<tr>
<td>VADDPS — Packed Single-FP Horizontal Add</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:7C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:7Cmod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:7C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:7Cmod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with xmmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7C:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7Cmod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to ymmreg1</td>
<td>C5: r_xmmreglo2 111:7C:11 ymmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to ymmreg1</td>
<td>C5: r_xmmreglo2 111:7Cmod ymmreg1 r/m</td>
</tr>
<tr>
<td>VSUBPD — Packed Double-FP Horizontal Subtract</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:7D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:7Dmod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:7D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:7Dmod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with xmmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:7D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:7Dmod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to ymmreg1</td>
<td>C5: r_xmmreglo2 101:7D:11 ymmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to ymmreg1</td>
<td>C5: r_xmmreglo2 101:7Dmod ymmreg1 r/m</td>
</tr>
<tr>
<td>VSUBPS — Packed Single-FP Horizontal Subtract</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:7D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:7Dmod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:7D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:7Dmod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with xmmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 111:7Dmod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to ymmreg1</td>
<td>C5: r_xmmreglo2 111:7D:11 ymmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to ymmreg1</td>
<td>C5: r_xmmreglo2 111:7Dmod ymmreg1 r/m</td>
</tr>
<tr>
<td>VLDDQU — Load Unaligned Integer 128 Bits</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:F0:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 011:F0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:F0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 111:F0:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VMOVDDUP — Move One Double-FP and Duplicate</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:12:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 011:12:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 011:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:12:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 111:12:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 111:12:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 111:12:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VMOVHLPs — Move Packed Single Precision Floating-Point Values High to Low**

<table>
<thead>
<tr>
<th></th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 and xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:12:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreglo2 and xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 000:12:11 xmmreg1 xmmreglo3</td>
</tr>
</tbody>
</table>

**VMOVSHDUP — Move Packed Single-FP High and Duplicate**

<table>
<thead>
<tr>
<th></th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:16:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:16:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:16:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:16:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:16:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:16:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:16:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:16:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VMOVSLDUP — Move Packed Single-FP Low and Duplicate**

<table>
<thead>
<tr>
<th></th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:12:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:12:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:12:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:12:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:12:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:12:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:12:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VADDPD — Add Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th></th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:58:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:58:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:58:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:58:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r ymmreglo2 101:58:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r ymmreglo2 101:58:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VADDSD — Add Scalar Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th></th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:58:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 011:58:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 011:58:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VANDPD — Bitwise Logical AND of Packed Double Precision Floating-Point Values**
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:54:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:54:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:54:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:54:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:54:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:54:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:54:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:54:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VANDNPD — Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:55:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:55:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:55:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:55:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:55:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:55:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:55:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:55:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VCMPPD — Compare Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:C2:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:C2:11 xmmreg1 xmmreglo3: imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:C2:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:C2:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:C2:11 ymmreg1 ymmreglo3: imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:C2:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VCMPSD — Compare Scalar Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:C2:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:C2:11 xmmreg1 xmmreglo3: imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:C2:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VCOMISD — Compare Scalar Ordered Double Precision Floating-Point Values and Set EFLAGS**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:2F:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:2F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:2F:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:2F:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VCVTDQ2PD — Convert Packed Dword Integers to Packed Double Precision FP Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:E6:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:E6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:E6:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:E6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:E6:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:E6:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:E6:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:E6:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VCVTDQ2PS— Convert Packed Dword Integers to Packed Single Precision FP Values**

| xmmreg2 to xmmreg1                        | C4: rxb0_1: w_F 000:5B:11 xmmreg1 xmmreg2 |
| mem to xmmreg1                            | C4: rxb0_1: w_F 000:5B:mod xmmreg1 r/m |
| xmmreglo to xmmreg1                       | C5: r_F 000:5B:11 xmmreg1 xmmreglo |
| mem to xmmreg1                            | C5: r_F 000:5B:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1                        | C4: rxb0_1: w_F 100:5B:11 ymmreg1 ymmreg2 |
| mem to ymmreg1                            | C4: rxb0_1: w_F 100:5B:mod ymmreg1 r/m |
| ymmreglo to ymmreg1                       | C5: r_F 100:5B:11 ymmreg1 ymmreglo |
| mem to ymmreg1                            | C5: r_F 100:5B:mod ymmreg1 r/m |

**VCVTPD2DQ— Convert Packed Double Precision FP Values to Packed Dword Integers**

| xmmreg2 to xmmreg1                        | C4: rxb0_1: w_F 011:E6:11 xmmreg1 xmmreg2 |
| mem to xmmreg1                            | C4: rxb0_1: w_F 011:E6:mod xmmreg1 r/m |
| xmmreglo to xmmreg1                       | C5: r_F 011:E6:11 xmmreg1 xmmreglo |
| mem to xmmreg1                            | C5: r_F 011:E6:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1                        | C4: rxb0_1: w_F 111:E6:11 ymmreg1 ymmreg2 |
| mem to ymmreg1                            | C4: rxb0_1: w_F 111:E6:mod ymmreg1 r/m |
| ymmreglo to ymmreg1                       | C5: r_F 111:E6:11 ymmreg1 ymmreglo |
| mem to ymmreg1                            | C5: r_F 111:E6:mod ymmreg1 r/m |

**VCVTPD2PS— Convert Packed Double Precision FP Values to Packed Single Precision FP Values**

| xmmreg2 to xmmreg1                        | C4: rxb0_1: w_F 001:5A:11 xmmreg1 xmmreg2 |
| mem to xmmreg1                            | C4: rxb0_1: w_F 001:5A:mod xmmreg1 r/m |
| xmmreglo to xmmreg1                       | C5: r_F 001:5A:11 xmmreg1 xmmreglo |
| mem to xmmreg1                            | C5: r_F 001:5A:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1                        | C4: rxb0_1: w_F 101:5A:11 ymmreg1 ymmreg2 |
| mem to ymmreg1                            | C4: rxb0_1: w_F 101:5A:mod ymmreg1 r/m |
| ymmreglo to ymmreg1                       | C5: r_F 101:5A:11 ymmreg1 ymmreglo |
| mem to ymmreg1                            | C5: r_F 101:5A:mod ymmreg1 r/m |

**VCVTPS2DQ— Convert Packed Single Precision FP Values to Packed Dword Integers**

| xmmreg2 to xmmreg1                        | C4: rxb0_1: w_F 001:5B:11 xmmreg1 xmmreg2 |
| mem to xmmreg1                            | C4: rxb0_1: w_F 001:5B:mod xmmreg1 r/m |
| xmmreglo to xmmreg1                       | C5: r_F 001:5B:11 xmmreg1 xmmreglo |
| mem to xmmreg1                            | C5: r_F 001:5B:mod xmmreg1 r/m |
| ymmreg2 to ymmreg1                        | C4: rxb0_1: w_F 101:5B:11 ymmreg1 ymmreg2 |
### Instruction and Format Encoding

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:5B:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:5B:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:5B:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VCVTPS2PD— Convert Packed Single Precision FP Values to Packed Double Precision FP Values**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:5A:11 xmmreg1 xmmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:5A:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:5A:11 xmmreg1 xmmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:5A:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:5A:11 ymmreg1 ymmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:5A:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:5A:11 ymmreg1 ymmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:5A:mod ymmreg1 r/m</td>
<td></td>
</tr>
</tbody>
</table>

**VCVTS2SI— Convert Scalar Double Precision FP Value to Integer**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg1 to reg32</td>
<td>C4: rxb0_1: 0_F 011:2D:11 reg xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to reg32</td>
<td>C4: rxb0_1: 0_F 011:2D:mod reg r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to reg32</td>
<td>C5: r_F 011:2D:11 reg xmmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to reg32</td>
<td>C5: r_F 011:2D:mod reg r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg1 to reg64</td>
<td>C4: rxb0_1: 1_F 111:2D:11 reg ymmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to reg64</td>
<td>C4: rxb0_1: 1_F 111:2D:mod reg r/m</td>
<td></td>
</tr>
</tbody>
</table>

**VCVTSD2SS — Convert Scalar Double Precision FP Value to Scalar Single Precision FP Value**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5A:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5A:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5A:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5A:mod xmmreg1 r/m</td>
<td></td>
</tr>
</tbody>
</table>

**VCVTSI2SD— Convert Dword Integer to Scalar Double Precision FP Value**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with reg to xmmreg1</td>
<td>C4: rxb0_1: 0 xmmreg2 011:2A:11 xmmreg1 reg</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: 0 xmmreg2 011:2A:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with reglo to xmmreg1</td>
<td>C5: r_xmmreglo2 011:2A:11 xmmreg1 reglo</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:2A:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with reg to ymmreg1</td>
<td>C4: rxb0_1: 1 ymmreg2 111:2A:11 ymmreg1 reg</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: 1 ymmreg2 111:2A:mod ymmreg1 r/m</td>
<td></td>
</tr>
</tbody>
</table>

**VCVTSS2SD — Convert Scalar Single Precision FP Value to Scalar Double Precision FP Value**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5A:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5A:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5A:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5A:mod xmmreg1 r/m</td>
<td></td>
</tr>
</tbody>
</table>

**VCVTPPD2DQ— Convert with Truncation Packed Double Precision FP Values to Packed Dword Integers**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:E6:11 xmmreg1 xmmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:E6:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:E6:11 xmmreg1 xmmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:E6:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:E6:11 ymmreg1 ymmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:E6:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to ymmreg1</td>
<td>C5: r_F 101:E6:11 ymmreg1 ymmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:E6:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>VCVTTPS2DQ—Convert with Truncation Packed Single Precision FP Values to Packed Dword Integers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:5B:11 xmmreg1 xmmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:5B:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:5B:11 xmmreg1 xmmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:5B:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:5B:11 ymmreg1 ymmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:5B:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:5B:11 ymmreg1 ymmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:5B:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>VCVTTSI—Convert with Truncation Scalar Double Precision FP Value to Signed Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg32</td>
<td>C4: rxb0_1: 0_F 011:2C:11 reg xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to reg32</td>
<td>C4: rxb0_1: 0_F 011:2C:mod reg r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to reg32</td>
<td>C5: r_F 011:2C:11 reg xmmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to reg32</td>
<td>C5: r_F 011:2C:mod reg r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg64</td>
<td>C4: rxb0_1: 1_F 011:2C:11 reg xmmreg1</td>
<td></td>
</tr>
<tr>
<td>mem to reg64</td>
<td>C4: rxb0_1: 1_F 011:2C:mod reg r/m</td>
<td></td>
</tr>
<tr>
<td>VDIVPD—Divide Packed Double Precision Floating-Point Values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5E:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5E:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5E:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5E:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:5E:11 ymmreg1 ymmreg3</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:5E:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5E:11 ymmreg1 ymmreglo3</td>
<td></td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5E:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>VDIVSD—Divide Scalar Double Precision Floating-Point Values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5E:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5E:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5E:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5E:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>VMASKMOVQDU—Store Selected Bytes of Double Quadword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem; xmmreg2 as mask</td>
<td>C4: rxb0_1: w_F 001:F7:11 r/m xmmreg1: xmmreg2</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem; xmmreg2 as mask</td>
<td>C5: r_F 001:F7:11 r/m xmmreg1: xmmreg2</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td><strong>VMAXPD — Return Maximum Packed Double Precision Floating-Point Values</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 001:5F:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 001:5F:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5F:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5F:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rb0_1: w ymmreg2 101:5F:11 ymmreg1 ymmreg3</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rb0_1: w ymmreg2 101:5F:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5F:11 ymmreg1 ymmreglo3</td>
<td></td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:5F:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td><strong>VMAXSD — Return Maximum Scalar Double Precision Floating-Point Value</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 011:5F:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 011:5F:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5F:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5F:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td><strong>VMINPD — Return Minimum Packed Double Precision Floating-Point Values</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 001:5D:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 001:5D:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5D:11 xmmreg1xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5D:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td><strong>VMINSD — Return Minimum Scalar Double Precision Floating-Point Value</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 011:5D:11 xmmreg1 xmmreg3</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_1: w xmmreg2 011:5D:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5D:11 xmmreg1 xmmreglo3</td>
<td></td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5D:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td><strong>VMOVAPD — Move Aligned Packed Double Precision Floating-Point Values</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_1: w_F 001:28:11 xmmreg1 xmmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_1: w_F 001:28:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:28:11 xmmreg1 xmmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:28:mod xmmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rb0_1: w_F 001:29:11 xmmreg2 xmmreg1</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rb0_1: w_F 001:29:mod r/m xmmreg1</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 001:29:11 xmmreglo xmmreg1</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 001:29:mod r/m xmmreg1</td>
<td></td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rb0_1: w_F 101:28:11 ymmreg1 ymmreg2</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rb0_1: w_F 101:28:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:28:11 ymmreg1 ymmreglo</td>
<td></td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:28:mod ymmreg1 r/m</td>
<td></td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 101:29:11 ymmreg2 ymmreg1</td>
<td></td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:29:mod r/m ymmreg1</td>
<td></td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 101:29:11 ymmreglo ymmreg1</td>
<td></td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 101:29:mod r/m ymmreg1</td>
<td></td>
</tr>
</tbody>
</table>

**VMOVVD — Move Doubleword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg32 to xmmreg1</td>
<td>C4: rxb0_1: 0_F 001:6E:11 xmmreg1 reg32</td>
</tr>
<tr>
<td>mem32 to xmmreg1</td>
<td>C4: rxb0_1: 0_F 001:6E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>reg32 to xmmreg1</td>
<td>C5: r_F 001:6E:11 xmmreg1 reg32</td>
</tr>
<tr>
<td>mem32 to xmmreg1</td>
<td>C5: r_F 001:6E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to reg32</td>
<td>C4: rxb0_1: 0_F 001:7E:11 reg32 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem32</td>
<td>C4: rxb0_1: 0_F 001:7E:mod mem32 xmmreg1</td>
</tr>
<tr>
<td>xmmreglo to reg32</td>
<td>C5: r_F 001:7E:11 reg32 xmmreglo</td>
</tr>
<tr>
<td>xmmreglo to mem32</td>
<td>C5: r_F 001:7E:mod mem32 xmmreglo</td>
</tr>
</tbody>
</table>

**VMOVQ — Move Quadword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg64 to xmmreg1</td>
<td>C4: rxb0_1: 1_F 001:6E:11 xmmreg1 reg64</td>
</tr>
<tr>
<td>mem64 to xmmreg1</td>
<td>C4: rxb0_1: 1_F 001:6E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to reg64</td>
<td>C4: rxb0_1: 1_F 001:7E:11 reg64 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem64</td>
<td>C4: rxb0_1: 1_F 001:7E:mod r/m xmmreg1</td>
</tr>
</tbody>
</table>

**VMOVQQA — Move Aligned Double Quadword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:6F:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:6F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:6F:11 xmmreg1 xmmrello</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:6F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 001:7F:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 001:7F:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 001:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:6F:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:6F:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 101:7F:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:7F:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 101:7F:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 101:7F:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

**VMOVDQU — Move Unaligned Double Quadword**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:6F:11 xmmreg1 xmmreg2</td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:6F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 010:6F:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:6F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 010:7F:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 010:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 010:7F:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 010:7F:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:6F:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 110:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 110:6F:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 110:6F:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 110:7F:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 110:7F:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 110:7F:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 110:7F:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

**VMOVHPD — Move High Packed Double Precision Floating-Point Value**

| xmmreg1 and mem to xmmreg2 | C4: rxb0_1: w xmmreg1 001:16:11 xmmreg2 r/m |
| xmmreg1 and mem to xmmreglo2 | C5: r_xmmreg1 001:16:11 xmmreglo2 r/m |
| xmmreg1 to mem              | C4: rxb0_1: w_F 001:17:mod r/m xmmreg1 |
| xmmreglo to mem             | C5: r_F 001:17:mod r/m xmmreglo |

**VMOVLPD — Move Low Packed Double Precision Floating-Point Value**

| xmmreg1 and mem to xmmreg2 | C4: rxb0_1: w xmmreg1 001:12:11 xmmreg2 r/m |
| xmmreg1 and mem to xmmreglo2 | C5: r_xmmreg1 001:12:11 xmmreglo2 r/m |
| xmmreg1 to mem              | C4: rxb0_1: w_F 001:13:mod r/m xmmreg1 |
| xmmreglo to mem             | C5: r_F 001:13:mod r/m xmmreglo |

**VMOVMSKPD — Extract Packed Double Precision Floating-Point Sign Mask**

| xmmreg2 to reg              | C4: rxb0_1: w_F 001:50:11 reg xmmreg1 |
| xmmreglo to reg             | C5: r_F 001:50:11 reg xmmreglo |
| ymmreg2 to reg              | C4: rxb0_1: w_F 101:50:11 reg ymmreg1 |
| ymmreglo to reg             | C5: r_F 101:50:11 reg ymmreglo |

**VMOVNTDQ — Store Double Quadword Using Non-Temporal Hint**

| xmmreg1 to mem              | C4: rxb0_1: w_F 001:E7:11 r/m xmmreg1 |
| xmmreglo to mem             | C5: r_F 001:E7:11 r/m xmmreglo |
| ymmreg1 to mem              | C4: rxb0_1: w_F 101:E7:11 r/m ymmreg1 |
| ymmreglo to mem             | C5: r_F 101:E7:11 r/m ymmreglo |

**VMOVNTPD — Store Packed Double Precision Floating-Point Values Using Non-Temporal Hint**

| xmmreg1 to mem              | C4: rxb0_1: w_F 001:2B:11 r/m xmmreg1 |
| xmmreglo to mem             | C5: r_F 001:2B:11 r/m xmmreglo |
| ymmreg1 to mem              | C4: rxb0_1: w_F 101:2B:11 r/m ymmreg1 |
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreglo to mem</td>
<td>C5: r_F 101:2B:11r/m ymmreglo</td>
</tr>
<tr>
<td><strong>VMOVSD</strong> — Move Scalar Double Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:10:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 011:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:10:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 011:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:11:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 011:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:11:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 011:11:mod r/m xmmreglo</td>
</tr>
<tr>
<td><strong>VMOVUPD</strong> — Move Unaligned Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:10:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 001:10:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 001:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:10:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 101:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 101:10:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 101:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 001:11:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 001:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 001:11:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 101:11:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 101:11:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 101:11:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 101:11:mod r/m ymmreg1</td>
</tr>
<tr>
<td><strong>VMULPD</strong> — Multiply Packed Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:59:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:59:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:59:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:59:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:59:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:59:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:59:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:59:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VMULSD</strong> — Multiply Scalar Double Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:59:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:59:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 011:59:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 011:59:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VORPD — Bitwise Logical OR of Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:56:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:56:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:56:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:56:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreg2 with ymmreg3 to ymmreg1</code></td>
<td><code>C4: rxb0_1: w ymmreg2 101:56:11 ymmreg1 ymmreg3</code></td>
</tr>
<tr>
<td><code>ymmreg2 with mem to ymmreg1</code></td>
<td><code>C4: rxb0_1: w ymmreg2 101:56:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreglo2 with ymmreglo3 to ymmreg1</code></td>
<td><code>C5: r ymmreglo2 101:56:11 ymmreg1 ymmreglo3</code></td>
</tr>
<tr>
<td><code>ymmreglo2 with mem to ymmreg1</code></td>
<td><code>C5: r ymmrello2 101:56:mod ymmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VPACKSSWB— Pack with Signed Saturation**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:63:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:63:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:63:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:63:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VPACKSSDW— Pack with Signed Saturation**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:6B:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:6B:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:6B:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:6B:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VPACKUSWB— Pack with Unsigned Saturation**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:67:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:67:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:67:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:67:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VPADD — Add Packed Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:FC:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:Fcmod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:FC:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:Fcmod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VPADDD — Add Packed Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:FE:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:FEmod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

**VPADDB — Add Packed Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:FAd:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td><code>C4: rxb0_1: w xmmreg2 001:FAdmod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:FAd:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td><code>C5: r xmmreglo2 001:FAdmod xmmreg1 r/m</code></td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:FE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:FE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDQ — Add Packed Quadword Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D4:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDSB — Add Packed Signed Integers with Signed Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EC:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:EC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:EC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDSW — Add Packed Signed Integers with Signed Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:ED:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:ED:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:ED:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:ED:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDSB — Add Packed Unsigned Integers with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DC:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DC:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DC:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPADDUSW — Add Packed Unsigned Integers with Unsigned Saturation</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DD:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DD:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DD:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DD:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPAND — Logical AND</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DB:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DB:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPANDN — Logical AND NOT</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DF:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:DF:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:DF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPAVGB — Average Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E0:11 xmmreg1 xmmreg3</td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E0 mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:E0:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:E0 mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPAVGW — Average Packed Integers**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E3:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:E3:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:E3:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPCMPEQB — Compare Packed Data for Equal**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:74:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:74:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:74:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:74:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPCMPEQW — Compare Packed Data for Equal**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:75:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:75:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:75:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:75:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPCMPEQD — Compare Packed Data for Equal**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:64:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:64:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:64:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:64:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPEXTRW — Extract Word**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:65:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:65:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:65:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:65:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreg1 to reg using imm</td>
<td>C4: rxb0_1: 0_F 001:C5:11 reg xmmreg1: imm</td>
</tr>
<tr>
<td>xmmreg1 to reg using imm</td>
<td>C5: r_F 001:C5:11 reg xmmreg1: imm</td>
</tr>
<tr>
<td><strong>VPINSRW</strong> — Insert Word</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with reg to xmmreg1</td>
<td>C4: rxb0_1: 0 xmmreg2 001:C4:11 xmmreg1 reg: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: 0 xmmreg2 001:C4:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo2 with reglo to xmmreg1</td>
<td>C5: r_xmmreglo2 001:C4:11 xmmreg1 reglo: imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:C4:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPMADDWD</strong> — Multiply and Add Packed Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F5:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F5:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:F5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXSW</strong> — Maximum of Packed Signed Word Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EE:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMAXUB</strong> — Maximum of Packed Unsigned Byte Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DE:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:DE:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:DE:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINSW</strong> — Minimum of Packed Signed Word Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EA:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EA:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMINUB</strong> — Minimum of Packed Unsigned Byte Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DA:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:DA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:DA:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:DA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPMOVMSKB</strong> — Move Byte Mask</td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: w_F 001:D7:11 reg xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C5: r_F 001:D7:11 reg xmmreg1</td>
</tr>
<tr>
<td><strong>VPMULHUW</strong> — Multiply Packed Unsigned Integers and Store High Result</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E4:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:E4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E4:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPMULHW — Multiply Packed Signed Integers and Store High Result**

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:E5:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E5:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E5:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPMULLw — Multiply Packed Signed Integers and Store Low Result**

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:D5:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D5:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D5:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D5:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPMULUDQ — Multiply Packed Unsigned Doubleword Integers**

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:F4:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F4:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:F4:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:F4:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPOR — Bitwise Logical OR**

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:EB:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:EB:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:EB:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPSADBW — Compute Sum of Absolute Differences**

<table>
<thead>
<tr>
<th>xmmreg2 with xmmreg3 to xmmreg1</th>
<th>C4: rxb0_1: w xmmreg2 001:F6:11 xmmreg1 xmmreg3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F6:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:F6:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:F6:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VPSHUFd — Shuffle Packed Doublewords**

<table>
<thead>
<tr>
<th>xmmreg2 to xmmreg1 using imm</th>
<th>C4: rxb0_1: w_F 001:70:11 xmmreg1 xmmreg2: imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 001:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 001:70:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C5: r_F 001:70:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPSHUFhW — Shuffle Packed High Words**

<table>
<thead>
<tr>
<th>xmmreg2 to xmmreg1 using imm</th>
<th>C4: rxb0_1: w_F 010:70:11 xmmreg1 xmmreg2: imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 010:70:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 010:70:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C5: r_F 010:70:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPSHUFllW — Shuffle Packed Low Words**

<table>
<thead>
<tr>
<th>xmmreg2 to xmmreg1 using imm</th>
<th>C4: rxb0_1: w_F 011:70:11 xmmreg1 xmmreg2: imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C4: rxb0_1: w_F 011:70:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo to xmmreg1 using imm</td>
<td>C5: r_F 011:70:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td>mem to xmmreg1 using imm</td>
<td>C5: r_F 011:70:mod xmmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VPSLLDQ — Shift Double Quadword Left Logical**

| xmmreg2 to xmmreg1 using imm | C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm | C5: r_F 001:73:11 xmmreg1 xmmreglo: imm |

**VPSLLW — Shift Packed Data Left Logical**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F1:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F1:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:F1:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:F1:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8 | C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:71:11 xmmreg1 xmmreglo: imm |

**VPSLLD — Shift Packed Data Left Logical**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F2:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F2:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:F2:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:F2:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8 | C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:72:11 xmmreg1 xmmreglo: imm |

**VPSLLQ — Shift Packed Data Left Logical**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F3:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:F3:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:F3:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:F3:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8 | C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:73:11 xmmreg1 xmmreglo: imm |

**VPSRAW — Shift Packed Data Right Arithmetic**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:E1:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:E1:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:E1:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:E1:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8 | C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm |
| xmmreglo to xmmreg1 using imm8 | C5: r_F 001:71:11 xmmreg1 xmmreglo: imm |

**VPSRAD — Shift Packed Data Right Arithmetic**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:E2:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:E2:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:E2:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:E2:mod xmmreg1 r/m |
| xmmreg2 to xmmreg1 using imm8 | C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm |
**INSTRUCTION FORMATS AND ENCODINGS**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg0 to xmmreg1 using imm8</td>
<td>C5: r_F 001:72:11 xmmreg1 xmmreg0: imm</td>
</tr>
<tr>
<td><strong>VPSRLDQ — Shift Double Quadword Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>xmmreg0 to xmmreg1 using imm8</td>
<td>C5: r_F 001:73:11 xmmreg1 xmmreg0: imm</td>
</tr>
<tr>
<td><strong>VPSRLW — Shift Packed Data Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D1:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D1:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D1:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D1:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:71:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:71:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSRLD — Shift Packed Data Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D2:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D2:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D2:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D2:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:72:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:72:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSRLQ — Shift Packed Data Right Logical</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D3:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D3:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D3:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w_F 001:73:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1 using imm8</td>
<td>C5: r_F 001:73:11 xmmreg1 xmmreglo: imm</td>
</tr>
<tr>
<td><strong>VPSUBB — Subtract Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F8:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:F8:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:F8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBW — Subtract Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F9:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:F9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:F9:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:F9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBD — Subtract Packed Integers</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FA:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:FA:11 xmmreg1 xmmreglo3</td>
</tr>
</tbody>
</table>
### Instruction and Format

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:FA:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBQ</strong> — Subtract Packed Quadword Integers</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FB:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:FB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:FB:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:FB:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBSB</strong> — Subtract Packed Signed Integers with Signed Saturation</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E8:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E8:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBSW</strong> — Subtract Packed Signed Integers with Signed Saturation</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E9:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:E9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:E9:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:E9:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPSUBUSB</strong> — Subtract Packed Unsigned Integers with Unsigned Saturation</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D8:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:D8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:D8:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:D8:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHBW</strong> — Unpack High Data</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:68:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:68:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:68:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:68:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHWD</strong> — Unpack High Data</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:69:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:69:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 001:69:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 001:69:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VPUNPCKHDQ</strong> — Unpack High Data</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6A:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6A:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>
### Instruction and Format

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:6A:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:6A:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VPUNPCKHDQ — Unpack High Data

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:6D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:6D:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VPUNPCKLBW — Unpack Low Data

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:61:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:61:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:61:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:61:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VPUNPCKLD — Unpack Low Data

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:62:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:62:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:62:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:62:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VPUNPCKLQD — Unpack Low Data

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:6C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:6C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:6C:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VPXOR — Logical Exclusive OR

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EF:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:EF:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EF:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:EF:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

#### VSHUFPD — Shuffle Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w xmmreg2 001:C6:11 xmmreg1 xmmreg3:imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1 using imm8</td>
<td>C4: rxb0_1: w xmmreg2 001:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1 using imm8</td>
<td>C5: r_xmmreglo2 001:C6:11 xmmreg1 xmmreglo3:imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1 using imm8</td>
<td>C5: r_xmmreglo2 001:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1 using imm8</td>
<td>C4: rxb0_1: w ymmreg2 101:C6:11 ymmreg1 ymmreg3:imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1 using imm8</td>
<td>C4: rxb0_1: w ymmreg2 101:C6:mod ymmreg1 r/m:imm</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1 using imm8</td>
<td>C5: r_ymmreglo2 101:6:11 ymmreg1 ymmreglo3: imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1 using imm8</td>
<td>C5: r_ymmreglo2 101:6:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**VSQRTPD — Compute Square Roots of Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Command</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:51:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 001:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:51:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:51:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VSQRTSD — Compute Square Root of Scalar Double Precision Floating-Point Value**

<table>
<thead>
<tr>
<th>Command</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:51:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:51:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:51:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VSUBPD — Subtract Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Command</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:5C:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VSUBSD — Subtract Scalar Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Command</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 011:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 011:5C:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VUCOMISD — Unordered Compare Scalar Double Precision Floating-Point Values and Set EFLAGS**

<table>
<thead>
<tr>
<th>Command</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg1, set EFLAGS</td>
<td>C4: rxb0_1: w_F xmmreg1 001:2E:11 xmmreg2</td>
</tr>
<tr>
<td>mem with xmmreg1, set EFLAGS</td>
<td>C4: rxb0_1: w_F xmmreg1 001:2E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo with xmmreg1, set EFLAGS</td>
<td>C5: r_F xmmreg1 001:2E:11 xmmreglo</td>
</tr>
<tr>
<td>mem with xmmreg1, set EFLAGS</td>
<td>C5: r_F xmmreg1 001:2E:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**VUNPCKHPD — Unpack and Interleave High Packed Double Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Command</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:15:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 001:15:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 001:15:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 001:15:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:15:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 101:15:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 101:15:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 101:15:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VUNPKHPS — Unpack and Interleave High Packed Single Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:15:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 000:15:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 000:15:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:15:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:15:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r_ymmreglo2 100:15:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r_ymmreglo2 100:15:mod ymmreg1 r/m |

**VUNPKLPD — Unpack and Interleave Low Packed Double Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:14:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:14:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:14:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:14:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 101:14:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 101:14:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r_ymmreglo2 101:14:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r_ymmreglo2 101:14:mod ymmreg1 r/m |

**VUNPKLPS — Unpack and Interleave Low Packed Single Precision Floating-Point Values**

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 000:14:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 000:14:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 000:14:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 100:14:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 100:14:mod ymmreg1 r/m |
| ymmreglo2 with ymmreglo3 to ymmreg1 | C5: r_ymmreglo2 100:14:11 ymmreg1 ymmreglo3 |
| ymmreglo2 with mem to ymmreg1 | C5: r_ymmreglo2 100:14:mod ymmreg1 r/m |

**VXORPD — Bitwise Logical XOR for Double Precision Floating-Point Values**

<p>| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 001:57:11 xmmreg1 xmmreg3 |
| xmmreg2 with mem to xmmreg1 | C4: rxb0_1: w xmmreg2 001:57:mod xmmreg1 r/m |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 001:57:11 xmmreg1 xmmreglo3 |
| xmmreglo2 with mem to xmmreg1 | C5: r_xmmreglo2 001:57:mod xmmreg1 r/m |
| ymmreg2 with ymmreg3 to ymmreg1 | C4: rxb0_1: w ymmreg2 101:57:11 ymmreg1 ymmreg3 |
| ymmreg2 with mem to ymmreg1 | C4: rxb0_1: w ymmreg2 101:57:mod ymmreg1 r/m |</p>
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmmreglo2 101:57:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmmreglo2 101:57:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VADDPS — Add Packed Single Precision</strong></td>
<td></td>
</tr>
<tr>
<td>Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:58:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:58:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreglo3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:58:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:58:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:58:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:58:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VADDSS — Add Scalar Single Precision</strong></td>
<td></td>
</tr>
<tr>
<td>Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:58:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:58:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:58:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreglo3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:54:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:54:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:54:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:54:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VANDPS — Bitwise Logical AND of Packed</strong></td>
<td></td>
</tr>
<tr>
<td>Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:54:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:54:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:54:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:54:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreglo3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:54:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:54:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:54:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:54:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VANDNPS — Bitwise Logical AND NOT of</strong></td>
<td></td>
</tr>
<tr>
<td>Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:55:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:55:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:55:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:55:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreglo3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:55:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:55:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:55:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmmreglo2 100:55:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VCMP</strong> — Compare Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:C2:11 xmmreg1 xmmreg3:imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:C2:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:C2:11 xmmreg1 xmmreglo3:imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:C2:mod xmmreg1 r/m:imm</td>
</tr>
</tbody>
</table>
## INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:C2:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:C2:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmmeglo2 100:C2:11 ymmreg1 ymmreglo3: imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmmeglo2 100:C2:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VCMPSS — Compare Scalar Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:C2:11 xmmreg1 xmmreg3: imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:C2:11 xmmreg1 xmmreglo3: imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:C2:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VCOMISS — Compare Scalar Ordered Single Precision Floating-Point Values and Set EFLAGS</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg1</td>
<td>C4: rxb0_1: w_F 000:2F:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem with xmmreg1</td>
<td>C4: rxb0_1: w_F 000:2F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo with xmmreg1</td>
<td>C5: r_F 000:2F:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem with xmmreg1</td>
<td>C5: r_F 000:2F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VCVTI2SS — Convert Dword Integer to Scalar Single Precision FP Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with reg to xmmreg1</td>
<td>C4: rxb0_1: 0 xmmreg2 010:2A:11 xmmreg1 reg</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: 0 xmmreg2 010:2A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with reglo to xmmreg1</td>
<td>C5: r_xmmreglo2 010:2A:11 xmmreg1 reglo</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:2A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 with reg to xmmreg1</td>
<td>C4: rxb0_1: 1 xmmreg2 010:2A:11 xmmreg1 reg</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: 1 xmmreg2 010:2A:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VCVTSS2SI — Convert Scalar Single Precision FP Value to Dword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: 0_F 010:2D:11 reg xmmreg1</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C4: rxb0_1: 0_F 010:2D:mod reg r/m</td>
</tr>
<tr>
<td>xmmreglo to reg</td>
<td>C5: r_F 010:2D:11 reg xmmreglo</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C5: r_F 010:2D:mod reg r/m</td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: 1_F 010:2D:11 reg xmmreg1</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C4: rxb0_1: 1_F 010:2D:mod reg r/m</td>
</tr>
<tr>
<td><strong>VCVTSS2SI — Convert with Truncation Scalar Single Precision FP Value to Dword Integer</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: 0_F 010:2C:11 reg xmmreg1</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C4: rxb0_1: 0_F 010:2C:mod reg r/m</td>
</tr>
<tr>
<td>xmmreglo to reg</td>
<td>C5: r_F 010:2C:11 reg xmmreglo</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C5: r_F 010:2C:mod reg r/m</td>
</tr>
<tr>
<td>xmmreg1 to reg</td>
<td>C4: rxb0_1: 1_F 010:2C:11 reg xmmreg1</td>
</tr>
<tr>
<td>mem to reg</td>
<td>C4: rxb0_1: 1_F 010:2C:mod reg r/m</td>
</tr>
<tr>
<td><strong>VDIVPS — Divide Packed Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:5E:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>Instruction and Format</td>
<td>Encoding</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 000:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:5E:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:5E:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r ymmreglo2 100:5E:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r ymmreglo2 100:5E:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>VDIVSS — Divide Scalar Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5E:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 010:5E:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 010:5E:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VLDMXCSR — Load MXCSR Register</td>
<td></td>
</tr>
<tr>
<td>mem to MXCSR reg</td>
<td>C4: rxb0_1: w_F 000:AEmod 011 r/m</td>
</tr>
<tr>
<td>mem to MXCSR reg</td>
<td>C5: r_F 000:AEmod 011 r/m</td>
</tr>
<tr>
<td>VMAXPS — Return Maximum Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 000:5F:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 000:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VMAXSS — Return Maximum Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5F:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 010:5F:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 010:5F:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VMINPS — Return Minimum Packed Single Precision Floating-Point Values</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 000:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 000:5D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>VMINSS — Return Minimum Scalar Single Precision Floating-Point Value</td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:5D:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

**INSTRUCTION FORMATS AND ENCODINGS**
## Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5D:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 010:5D:mod xmmreg1 r/m</td>
</tr>
</tbody>
</table>

### VMOVAPS — Move Aligned Packed Single Precision Floating-Point Values

| xmmreg2 to xmmreg1 | C4: rxb0_1: w_F 000:28:11 xmmreg1 xmmreg2 |
| mem to xmmreg1 | C4: rxb0_1: w_F 000:28:mod xmmreg1 r/m |
| xmmreglo to xmmreg1 | C5: r_F 000:28:11 xmmreg1 xmmreglo |
| mem to xmmreg1 | C5: r_F 000:28:mod xmmreg1 r/m |
| xmmreg1 to xmmreg2 | C4: rxb0_1: w_F 000:29:11 xmmreg2 xmmreg1 |
| xmmreg1 to mem | C4: rxb0_1: w_F 000:29:mod r/m xmmreg1 |
| xmmreg1 to xmmreglo | C5: r_F 000:29:11 xmmreglo xmmreg1 |
| mem to xmmreg1 | C5: r_F 000:29:mod r/m xmmreg1 |
| ymmreg2 to ymmreg1 | C4: rxb0_1: w_F 100:28:11 ymmreg1 ymmreg2 |
| mem to ymmreg1 | C4: rxb0_1: w_F 100:28:mod ymmreg1 r/m |
| ymmreglo to ymmreg1 | C5: r_F 100:28:11 ymmreg1 ymmreglo |
| mem to ymmreg1 | C5: r_F 100:28:mod ymmreg1 r/m |
| ymmreg1 to ymmreg2 | C4: rxb0_1: w_F 100:29:11 ymmreg2 ymmreg1 |
| ymmreg1 to mem | C4: rxb0_1: w_F 100:29:mod r/m ymmreg1 |
| ymmreg1 to ymmreglo | C5: r_F 100:29:11 ymmreglo ymmreg1 |
| mem to ymmreg1 | C5: r_F 100:29:mod r/m ymmreg1 |

### VMOVHPS — Move High Packed Single Precision Floating-Point Values

| xmmreg1 with mem to xmmreg2 | C4: rxb0_1: w xmmreg1 000:16:mod xmmreg2 r/m |
| xmmreg1 with mem to xmmreglo2 | C5: r_xmmreg1 000:16:mod xmmreglo2 r/m |
| xmmreg1 to mem | C4: rxb0_1: w_F 000:17:mod r/m xmmreg1 |
| xmmreglo to mem | C5: r_F 000:17:mod r/m xmmreglo |

### VMOVLHPS — Move Packed Single Precision Floating-Point Values Low to High

| xmmreg2 with xmmreg3 to xmmreg1 | C4: rxb0_1: w xmmreg2 000:16:11 xmmreg1 xmmreg3 |
| xmmreglo2 with xmmreglo3 to xmmreg1 | C5: r_xmmreglo2 000:16:11 xmmreg1 xmmreglo3 |

### VMOVLPs — Move Low Packed Single Precision Floating-Point Values

| xmmreg1 with mem to xmmreg2 | C4: rxb0_1: w xmmreg1 000:12:mod xmmreg2 r/m |
| xmmreg1 with mem to xmmreglo2 | C5: r_xmmreg1 000:12:mod xmmreglo2 r/m |
| xmmreg1 to mem | C4: rxb0_1: w_F 000:13:mod r/m xmmreg1 |
| xmmreglo to mem | C5: r_F 000:13:mod r/m xmmreglo |

### VMOVMSKPS — Extract Packed Single Precision Floating-Point Sign Mask

| xmmreg2 to reg | C4: rxb0_1: w_F 000:50:11 reg xmmreg2 |
| xmmreglo to reg | C5: r_F 000:50:11 reg xmmreglo |
| ymmreg2 to reg | C4: rxb0_1: w_F 100:50:11 reg ymmreg2 |
| ymmreglo to reg | C5: r_F 100:50:11 reg ymmreglo |

### VMOVNTPS — Store Packed Single Precision Floating-Point Values Using Non-Temporal Hint

| xmmreg1 to mem | C4: rxb0_1: w_F 000:28:mod r/m xmmreg1 |
### Instruction and Format

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 000:2B:mod r/m xmmreglo</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 100:2B:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreglo to mem</td>
<td>C5: r_F 100:2B:mod r/m ymmreglo</td>
</tr>
</tbody>
</table>

**VMOVSS — Move Scalar Single Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:10:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 010:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C5: r_xmmreg2 010:10:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 010:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 010:11:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 010:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 010:11:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo to mem</td>
<td>C5: r_F 010:11:mod r/m xmmreglo</td>
</tr>
</tbody>
</table>

**VMOVUPS — Move Unaligned Packed Single Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:10:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1: w_F 000:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r_F 000:10:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r_F 000:10:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:10:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1: w_F 100:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r_F 100:10:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r_F 100:10:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg1 to xmmreg2</td>
<td>C4: rxb0_1: w_F 000:11:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C4: rxb0_1: w_F 000:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to xmmreglo</td>
<td>C5: r_F 000:11:11 xmmreglo xmmreg1</td>
</tr>
<tr>
<td>xmmreg1 to mem</td>
<td>C5: r_F 000:11:mod r/m xmmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreg2</td>
<td>C4: rxb0_1: w_F 100:11:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C4: rxb0_1: w_F 100:11:mod r/m ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to ymmreglo</td>
<td>C5: r_F 100:11:11 ymmreglo ymmreg1</td>
</tr>
<tr>
<td>ymmreg1 to mem</td>
<td>C5: r_F 100:11:mod r/m ymmreg1</td>
</tr>
</tbody>
</table>

**VMULPS — Multiply Packed Single Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:59:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1: w xmmreg2 000:59:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r_xmmreglo2 000:59:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r_xmmreglo2 000:59:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:59:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1: w ymmreg2 100:59:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r_ymmreglo2 100:59:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r_ymmreglo2 100:59:mod ymmreg1 r/m</td>
</tr>
</tbody>
</table>

**VMULSS — Multiply Scalar Single Precision Floating-Point Values**
### Instruction and Format

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 010:59:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 010:59:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r xmmreglo2 010:59:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r xmmreglo2 010:59:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

#### VORPS — Bitwise Logical OR of Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 000:56:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 000:56:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r xmmreglo2 000:56:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r xmmreglo2 000:56:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

#### VRCPPS — Compute Reciprocals of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 000:53:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 000:53:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo to xmmreg1</code></td>
<td>C5: <code>r_F 000:53:11 xmmreg1 xmmreglo</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C5: <code>r_F 000:53:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreg2 to ymmreg1</code></td>
<td>C4: <code>rb0_1: w_F 100:53:11 ymmreg1 ymmreg2</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C4: <code>rb0_1: w_F 100:53:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreglo to ymmreg1</code></td>
<td>C5: <code>r_F 100:53:11 ymmreg1 ymmreglo</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C5: <code>r_F 100:53:mod ymmreg1 r/m</code></td>
</tr>
</tbody>
</table>

#### VRCPSS — Compute Reciprocal of Scalar Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 010:53:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 010:53:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r xmmreglo2 010:53:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r xmmreglo2 010:53:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>

#### VRSQRTPS — Compute Reciprocals of Square Roots of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 000:52:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w_F 000:52:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo to xmmreg1</code></td>
<td>C5: <code>r_F 000:52:11 xmmreg1 xmmreglo</code></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C5: <code>r_F 000:52:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreg2 to ymmreg1</code></td>
<td>C4: <code>rb0_1: w_F 100:52:11 ymmreg1 ymmreg2</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C4: <code>rb0_1: w_F 100:52:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreglo to ymmreg1</code></td>
<td>C5: <code>r_F 100:52:11 ymmreg1 ymmreglo</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C5: <code>r_F 100:52:mod ymmreg1 r/m</code></td>
</tr>
</tbody>
</table>

#### VRSQRTSS — Compute Reciprocal of Square Root of Scalar Single Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 010:52:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rb0_1: w xmmreg2 010:52:mod xmmreg1 r/m</code></td>
</tr>
</tbody>
</table>
### INSTRUCTION FORMATS AND ENCODINGS

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 010:52:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 010:52:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VSHUFPS — Shuffle Packed Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1, imm8</td>
<td>C4: rxb0_1:w 000:C6:11 xmmreg1 xmmreg3:imm</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1, imm8</td>
<td>C4: rxb0_1:w 000:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1, imm8</td>
<td>C5: r xmmreglo2 000:C6:11 xmmreg1 xmmreglo3:imm</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1, imm8</td>
<td>C5: r xmmreglo2 000:C6:mod xmmreg1 r/m:imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1, imm8</td>
<td>C4: rxb0_1:w 100:C6:11 ymmreg1 ymmreg3:imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1, imm8</td>
<td>C4: rxb0_1:w 100:C6:mod ymmreg1 r/m:imm</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1, imm8</td>
<td>C5: r ymmreglo2 100:C6:11 ymmreg1 ymmreglo3:imm</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1, imm8</td>
<td>C5: r ymmreglo2 100:C6:mod ymmreg1 r/m:imm</td>
</tr>
<tr>
<td><strong>VSQRTPS — Compute Square Roots of Packed Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rxb0_1:w 000:51:11 xmmreg1 xmmreg2</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rxb0_1:w 000:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo to xmmreg1</td>
<td>C5: r xmmreglo 000:51:11 xmmreg1 xmmreglo</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C5: r xmmreglo 000:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rxb0_1:w 100:51:11 ymmreg1 ymmreg2</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rxb0_1:w 100:51:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo to ymmreg1</td>
<td>C5: r ymmreglo 100:51:11 ymmreg1 ymmreglo</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C5: r ymmreglo 100:51:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VSQRTSS — Compute Square Root of Scalar Single Precision Floating-Point Value</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1:w 010:51:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1:w 010:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 010:51:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 010:51:mod xmmreg1 r/m</td>
</tr>
<tr>
<td><strong>VSTMXCSR — Store MXCSR Register State</strong></td>
<td></td>
</tr>
<tr>
<td>MXCSR to mem</td>
<td>C4: rxb0_1:w 000:AE:mod 011 r/m</td>
</tr>
<tr>
<td>MXCSR to mem</td>
<td>C4: r_x 000:AE:mod 011 r/m</td>
</tr>
<tr>
<td><strong>VSUBPS — Subtract Packed Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1:w 000:5C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rxb0_1:w 000:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreglo2 with xmmreglo3 to xmmreg1</td>
<td>C5: r xmmreglo2 000:5C:11 xmmreg1 xmmreglo3</td>
</tr>
<tr>
<td>xmmreglo2 with mem to xmmreg1</td>
<td>C5: r xmmreglo2 000:5C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rxb0_1:w 100:5C:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rxb0_1:w 100:5C:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreglo2 with ymmreglo3 to ymmreg1</td>
<td>C5: r ymmreglo2 100:5C:11 ymmreg1 ymmreglo3</td>
</tr>
<tr>
<td>ymmreglo2 with mem to ymmreg1</td>
<td>C5: r ymmreglo2 100:5C:mod ymmreg1 r/m</td>
</tr>
<tr>
<td><strong>VSUBSS — Subtract Scalar Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rxb0_1:w 010:5C:11 xmmreg1 xmmreg3</td>
</tr>
</tbody>
</table>
### Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 010:5C:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r_xxmmreglo2 010:5C:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r_xxmmreglo2 010:5C:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VUCOMISS — Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg1</code></td>
<td>C4: <code>rxbo_1: w_F 000:2E:11 xmmreg1 xmmreg2</code></td>
</tr>
<tr>
<td><code>mem with xmmreg1</code></td>
<td>C4: <code>rxbo_1: w_F 000:2E:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo with xmmreg1</code></td>
<td>C5: <code>r_F 000:2E:11 xmmreg1 xmmreglo</code></td>
</tr>
<tr>
<td><code>mem with xmmreg1</code></td>
<td>C5: <code>r_F 000:2E:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>UNPCKHPS — Unpack and Interleave High Packed Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 000:15:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 000:15:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreg2 with ymmreg3 to ymmreg1</code></td>
<td>C4: <code>r_yymmreg2 100:15:11 ymmreg1 ymmreg3</code></td>
</tr>
<tr>
<td><code>ymmreg2 with mem to ymmreg1</code></td>
<td>C4: <code>r_yymmreg2 100:15:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>UNPCKLPS — Unpack and Interleave Low Packed Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 000:14:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 000:14:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>ymmreg2 with ymmreg3 to ymmreg1</code></td>
<td>C4: <code>rxbo_1: w ymmreg2 100:14:11 ymmreg1 ymmreg3</code></td>
</tr>
<tr>
<td><code>ymmreg2 with mem to ymmreg1</code></td>
<td>C4: <code>rxbo_1: w ymmreg2 100:14:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VXORPS — Bitwise Logical XOR for Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg2 with xmmreg3 to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 000:57:11 xmmreg1 xmmreg3</code></td>
</tr>
<tr>
<td><code>xmmreg2 with mem to xmmreg1</code></td>
<td>C4: <code>rxbo_1: w xmmreg2 000:57:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with xmmreglo3 to xmmreg1</code></td>
<td>C5: <code>r_xxmmreglo2 000:57:11 xmmreg1 xmmreglo3</code></td>
</tr>
<tr>
<td><code>xmmreglo2 with mem to xmmreg1</code></td>
<td>C5: <code>r_xxmmreglo2 000:57:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VBROADCAST — Load with Broadcast</strong></td>
<td></td>
</tr>
<tr>
<td><code>mem to xmmreg1</code></td>
<td>C4: <code>rxbo_2: 0_F 001:18:mod xmmreg1 r/m</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C4: <code>rxbo_2: 0_F 101:18:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C4: <code>rxbo_2: 0_F 101:19:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><code>mem to ymmreg1</code></td>
<td>C4: <code>rxbo_2: 0_F 101:1A:mod ymmreg1 r/m</code></td>
</tr>
<tr>
<td><strong>VEXTRACTF128 — Extract Packed Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>ymmreg2 to xmmreg1, imm8</code></td>
<td>C4: <code>rxbo_3: 0_F 001:19:11 xmmreg1 ymmreg2: imm</code></td>
</tr>
<tr>
<td><code>ymmreg2 to mem, imm8</code></td>
<td>C4: <code>rxbo_3: 0_F 001:19:mod r/m ymmreg2: imm</code></td>
</tr>
<tr>
<td><strong>VINSERTF128 — Insert Packed Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td><code>xmmreg3 and merge with ymmreg2 to ymmreg1, imm8</code></td>
<td>C4: <code>rxbo_3: 0 ymmreg2101:18:11 ymmreg1 xmmreg3: imm</code></td>
</tr>
<tr>
<td><code>mem and merge with ymmreg2 to ymmreg1, imm8</code></td>
<td>C4: <code>rxbo_3: 0 ymmreg2 101:18:mod ymmreg1 r/m: imm</code></td>
</tr>
<tr>
<td><strong>VPERMILPD — Permute Double Precision Floating-Point Values</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction and Formats

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: 0 xmmreg2 001:0D:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: 0 xmmreg2 001:0D:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rb0_2: 0 ymmreg2 101:0D:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rb0_2: 0 ymmreg2 101:0D:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm</td>
<td>C4: rb0_3: 0_F 001:05:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm</td>
<td>C4: rb0_3: 0_F 001:05:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm</td>
<td>C4: rb0_3: 0_F 101:05:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm</td>
<td>C4: rb0_3: 0_F 101:05:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPERMILPS — Permute Single Precision Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 with xmmreg3 to xmmreg1</td>
<td>C4: rb0_2: 0 xmmreg2 001:0C:11 xmmreg1 xmmreg3</td>
</tr>
<tr>
<td>xmmreg2 with mem to xmmreg1</td>
<td>C4: rb0_2: 0 xmmreg2 001:0C:mod xmmreg1 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1, imm</td>
<td>C4: rb0_3: 0_F 001:04:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1, imm</td>
<td>C4: rb0_3: 0_F 001:04:mod xmmreg1 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rb0_2: 0 ymmreg2 101:0C:11 ymmreg1 ymmreg3</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rb0_2: 0 ymmreg2 101:0C:mod ymmreg1 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1, imm</td>
<td>C4: rb0_3: 0_F 101:04:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1, imm</td>
<td>C4: rb0_3: 0_F 101:04:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VPERM2F128 — Permute Floating-Point Values</strong></td>
<td></td>
</tr>
<tr>
<td>ymmreg2 with ymmreg3 to ymmreg1</td>
<td>C4: rb0_3: 0_F 101:06:11 ymmreg1 ymmreg3: imm</td>
</tr>
<tr>
<td>ymmreg2 with mem to ymmreg1</td>
<td>C4: rb0_3: 0_F 101:06:mod ymmreg1 r/m: imm</td>
</tr>
<tr>
<td><strong>VTESTPD/VTESTPS — Packed Bit Test</strong></td>
<td></td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: 0_F 001:0E:11 xmmreg2 xmmreg1</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: 0_F 001:0E:mod xmmreg2 r/m</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rb0_2: 0_F 101:0E:11 ymmreg2 ymmreg1</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rb0_2: 0_F 101:0E:mod ymmreg2 r/m</td>
</tr>
<tr>
<td>xmmreg2 to xmmreg1</td>
<td>C4: rb0_2: 0_F 001:0F:11 xmmreg1 xmmreg2: imm</td>
</tr>
<tr>
<td>mem to xmmreg1</td>
<td>C4: rb0_2: 0_F 001:0F:mod xmmreg2 r/m: imm</td>
</tr>
<tr>
<td>ymmreg2 to ymmreg1</td>
<td>C4: rb0_2: 0_F 101:0F:11 ymmreg1 ymmreg2: imm</td>
</tr>
<tr>
<td>mem to ymmreg1</td>
<td>C4: rb0_2: 0_F 101:0F:mod ymmreg1 r/m: imm</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The term "lo" refers to the lower eight registers, 0-7
INSTRUCTION FORMATS AND ENCODINGS

B.17 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-38 shows the five different formats used for floating-point instructions. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011.

Table B-38. General Floating-Point Instruction Formats

<table>
<thead>
<tr>
<th>Instruction</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>Optional Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11011</td>
<td>OPA</td>
<td>1 mod 1 OPB r/m s-i-b disp</td>
</tr>
<tr>
<td>2</td>
<td>11011</td>
<td>MF</td>
<td>OPA mod OPB r/m s-i-b disp</td>
</tr>
<tr>
<td>3</td>
<td>11011</td>
<td>d P OPA</td>
<td>1 1 OPB R ST(i)</td>
</tr>
<tr>
<td>4</td>
<td>11011</td>
<td>0 0 1 1 1 1</td>
<td>OP</td>
</tr>
<tr>
<td>5</td>
<td>11011</td>
<td>0 1 1 1 1 1</td>
<td>OP</td>
</tr>
</tbody>
</table>

MF = Memory Format
00 — 32-bit real
01 — 32-bit integer
10 — 64-bit real
11 — 16-bit integer

P = Pop
0 — Do not pop stack
1 — Pop stack after operation

R X 0 = Destination OP Source
R XOR 0 = Source OP Destination

d = Destination
0 — Destination is ST(0)
1 — Destination is ST(i)

ST(i) = Register stack element i
000 = Stack Top
001 = Second stack element
111 = Eighth stack element

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-39 shows the formats and encodings of the floating-point instructions.

Table B-39. Floating-Point Instruction Formats and Encodings

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2XM1 - Compute 2^{ST(0)} - 1</td>
<td>11011 001:1111 0000</td>
</tr>
<tr>
<td>FABS - Absolute Value</td>
<td>11011 001:1110 0001</td>
</tr>
<tr>
<td>FADD - Add</td>
<td></td>
</tr>
<tr>
<td>ST(0) := ST(0) + 32-bit memory</td>
<td>11011 000:mod 000 r/m</td>
</tr>
<tr>
<td>ST(0) := ST(0) + 64-bit memory</td>
<td>11011 100:mod 000 r/m</td>
</tr>
<tr>
<td>ST(d) := ST(0) + ST(i)</td>
<td>11011 d00:11 000 ST(i)</td>
</tr>
<tr>
<td>FADDP - Add and Pop</td>
<td></td>
</tr>
<tr>
<td>ST(0) := ST(0) + ST(i)</td>
<td>11011 110:11 000 ST(i)</td>
</tr>
<tr>
<td>FBLD - Load Binary Coded Decimal</td>
<td>11011 111:mod 100 r/m</td>
</tr>
<tr>
<td>FBSTP - Store Binary Coded Decimal and Pop</td>
<td>11011 111:mod 110 r/m</td>
</tr>
<tr>
<td>FCHS - Change Sign</td>
<td>11011 001:1110 0000</td>
</tr>
<tr>
<td>FCLEX - Clear Exceptions</td>
<td>11011 011:1110 0010</td>
</tr>
<tr>
<td>FCOM - Compare Real</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit memory</td>
<td>11011 000 : mod 010 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 100 : mod 010 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 000 : 11 010 ST(i)</td>
</tr>
</tbody>
</table>

**FCOMP – Compare Real and Pop**

| 32-bit memory                          | 11011 000 : mod 011 r/m |
| 64-bit memory                          | 11011 100 : mod 011 r/m |
| ST(i)                                  | 11011 000 : 11 011 ST(i) |

**FCOMPP – Compare Real and Pop Twice**

11011 110 : 11 011 001

**FCOMP – Compare Real, Set EFLAGS, and Pop**

11011 111 : 11 110 ST(i)

**FCOS – Cosine of ST(0)**

11011 001 : 1111 1111

**FDECSTP – Decrement Stack-Top Pointer**

11011 001 : 1111 0110

**FDIV – Divide**

ST(0) := ST(0) ÷ 32-bit memory

11011 000 : mod 110 r/m

ST(0) := ST(0) ÷ 64-bit memory

11011 100 : mod 110 r/m

ST(d) := ST(0) ÷ ST(i)

11011 d00 : 1111 R ST(i)

**FDIVP – Divide and Pop**

ST(0) := ST(0) ÷ ST(i)

11011 110 : 1111 1 ST(i)

**FDIVR – Reverse Divide**

ST(0) := 32-bit memory ÷ ST(0)

11011 000 : mod 111 r/m

ST(0) := 64-bit memory ÷ ST(0)

11011 100 : mod 111 r/m

ST(d) := ST(i) ÷ ST(0)

11011 d00 : 1111 R ST(i)

**FDIVRP – Reverse Divide and Pop**

ST(0) := ST(i) ÷ ST(0)

11011 110 : 1111 0 ST(i)

**FFREE – Free ST(i) Register**

11011 101 : 1100 0 ST(i)

**FIADD – Add Integer**

ST(0) := ST(0) + 16-bit memory

11011 110 : mod 000 r/m

ST(0) := ST(0) + 32-bit memory

11011 010 : mod 000 r/m

**FICOM – Compare Integer**

16-bit memory

11011 110 : mod 010 r/m

32-bit memory

11011 010 : mod 010 r/m

**FICOMP – Compare Integer and Pop**

16-bit memory

11011 110 : mod 011 r/m

32-bit memory

11011 010 : mod 011 r/m

**FIDIV – Divide**

ST(0) := ST(0) ÷ 16-bit memory

11011 110 : mod 110 r/m

ST(0) := ST(0) ÷ 32-bit memory

11011 010 : mod 110 r/m

**FIDIVR – Reverse Divide**

ST(0) := 16-bit memory ÷ ST(0)

11011 110 : mod 111 r/m
<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) := 32-bit memory ÷ ST(0)</td>
<td>11011 010 : mod 111 r/m</td>
</tr>
<tr>
<td>FILD - Load Integer</td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 000 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 000 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 111 : mod 101 r/m</td>
</tr>
<tr>
<td>FIMUL - Multiply</td>
<td></td>
</tr>
<tr>
<td>ST(0) := ST(0) × 16-bit memory</td>
<td>11011 110 : mod 001 r/m</td>
</tr>
<tr>
<td>ST(0) := ST(0) × 32-bit memory</td>
<td>11011 010 : mod 001 r/m</td>
</tr>
<tr>
<td>FINCSTP - Increment Stack Pointer</td>
<td>11011 001 : 1111 0111</td>
</tr>
<tr>
<td>FINIT - Initialize Floating-Point Unit</td>
<td></td>
</tr>
<tr>
<td>FIST - Store Integer</td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 010 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 010 r/m</td>
</tr>
<tr>
<td>FISTP - Store Integer and Pop</td>
<td></td>
</tr>
<tr>
<td>16-bit memory</td>
<td>11011 111 : mod 011 r/m</td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 011 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 111 : mod 111 r/m</td>
</tr>
<tr>
<td>FISUB - Subtract</td>
<td></td>
</tr>
<tr>
<td>ST(0) := ST(0) - 16-bit memory</td>
<td>11011 110 : mod 100 r/m</td>
</tr>
<tr>
<td>ST(0) := ST(0) - 32-bit memory</td>
<td>11011 010 : mod 100 r/m</td>
</tr>
<tr>
<td>FISUBR - Reverse Subtract</td>
<td></td>
</tr>
<tr>
<td>ST(0) := 16-bit memory – ST(0)</td>
<td>11011 110 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(0) := 32-bit memory – ST(0)</td>
<td>11011 010 : mod 101 r/m</td>
</tr>
<tr>
<td>FLD - Load Real</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 000 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 000 r/m</td>
</tr>
<tr>
<td>80-bit memory</td>
<td>11011 011 : mod 101 r/m</td>
</tr>
<tr>
<td>ST(i)</td>
<td>11011 001 : 11 000 ST(i)</td>
</tr>
<tr>
<td>FLDE1 - Load +1.0 into ST(0)</td>
<td>11011 001 : 1111 1000</td>
</tr>
<tr>
<td>FLDCW - Load Control Word</td>
<td>11011 001 : mod 101 r/m</td>
</tr>
<tr>
<td>FLDENV - Load FPU Environment</td>
<td>11011 001 : mod 100 r/m</td>
</tr>
<tr>
<td>FLDLL2E - Load log₂(e) into ST(0)</td>
<td>11011 001 : 1110 1010</td>
</tr>
<tr>
<td>FLDLL2T - Load log₂(10) into ST(0)</td>
<td>11011 001 : 1110 1001</td>
</tr>
<tr>
<td>FLDLG2 - Load log₁₀(2) into ST(0)</td>
<td>11011 001 : 1110 1100</td>
</tr>
<tr>
<td>FLDLN2 - Load log₂(10) into ST(0)</td>
<td>11011 001 : 1111 1001</td>
</tr>
<tr>
<td>FLDPI - Load π into ST(0)</td>
<td>11011 001 : 1110 1011</td>
</tr>
<tr>
<td>FLDZ - Load +0.0 into ST(0)</td>
<td>11011 001 : 1110 1110</td>
</tr>
<tr>
<td>FMUL - Multiply</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-39. Floating-Point Instruction Formats and Encodings (Contd.)

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ST(0) := ST(0) \times 32\text{-}bit ) memory</td>
<td>11011 000 : mod 001 r/m</td>
</tr>
<tr>
<td>( ST(0) := ST(0) \times 64\text{-}bit ) memory</td>
<td>11011 100 : mod 001 r/m</td>
</tr>
<tr>
<td>( ST(d) := ST(0) \times ST(i) )</td>
<td>11011 d00 : 1100 1 ST(i)</td>
</tr>
<tr>
<td><strong>FMULP</strong> - Multiply</td>
<td></td>
</tr>
<tr>
<td>( ST(i) := ST(0) \times ST(i) )</td>
<td>11011 110 : 1101 ST(i)</td>
</tr>
<tr>
<td><strong>FNOP</strong> - No Operation</td>
<td>11011 001 : 1101 0000</td>
</tr>
<tr>
<td><strong>FPATAN</strong> - Partial Arctangent</td>
<td>11011 001 : 1111 0011</td>
</tr>
<tr>
<td><strong>FPREM</strong> - Partial Remainder</td>
<td>11011 001 : 1111 1000</td>
</tr>
<tr>
<td><strong>FPREM1</strong> - Partial Remainder (IEEE)</td>
<td>11011 001 : 1111 0101</td>
</tr>
<tr>
<td><strong>FPTAN</strong> - Partial Tangent</td>
<td>11011 001 : 1111 0010</td>
</tr>
<tr>
<td><strong>FRNDINT</strong> - Round to Integer</td>
<td>11011 001 : 1111 1100</td>
</tr>
<tr>
<td><strong>FRSTOR</strong> - Restore FPU State</td>
<td>11011 101 : mod 100 r/m</td>
</tr>
<tr>
<td><strong>FSAVE</strong> - Store FPU State</td>
<td>11011 101 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>FSCALE</strong> - Scale</td>
<td>11011 001 : 1111 1101</td>
</tr>
<tr>
<td><strong>FSIN</strong> - Sine</td>
<td>11011 001 : 1111 1110</td>
</tr>
<tr>
<td><strong>FSINCOS</strong> - Sine and Cosine</td>
<td>11011 001 : 1111 1011</td>
</tr>
<tr>
<td><strong>FSQRT</strong> - Square Root</td>
<td>11011 001 : 1111 1010</td>
</tr>
<tr>
<td><strong>FST</strong> - Store Real</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 010 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 010 r/m</td>
</tr>
<tr>
<td>( ST(i) )</td>
<td>11011 101 : 11 010 ST(i)</td>
</tr>
<tr>
<td><strong>FSTCW</strong> - Store Control Word</td>
<td>11011 001 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FSTENV</strong> - Store FPU Environment</td>
<td>11011 001 : mod 110 r/m</td>
</tr>
<tr>
<td><strong>FSTP</strong> - Store Real and Pop</td>
<td></td>
</tr>
<tr>
<td>32-bit memory</td>
<td>11011 001 : mod 011 r/m</td>
</tr>
<tr>
<td>64-bit memory</td>
<td>11011 101 : mod 011 r/m</td>
</tr>
<tr>
<td>80-bit memory</td>
<td>11011 011 : mod 111 r/m</td>
</tr>
<tr>
<td>( ST(i) )</td>
<td>11011 101 : 11 011 ST(i)</td>
</tr>
<tr>
<td><strong>FSTSW</strong> - Store Status Word into AX</td>
<td>11011 111 : 1110 0000</td>
</tr>
<tr>
<td><strong>FSTSW</strong> - Store Status Word into Memory</td>
<td>11011 101 : mod 111 r/m</td>
</tr>
<tr>
<td><strong>FSUB</strong> - Subtract</td>
<td></td>
</tr>
<tr>
<td>( ST(0) := ST(0) - 32\text{-}bit ) memory</td>
<td>11011 000 : mod 100 r/m</td>
</tr>
<tr>
<td>( ST(0) := ST(0) - 64\text{-}bit ) memory</td>
<td>11011 100 : mod 100 r/m</td>
</tr>
<tr>
<td>( ST(d) := ST(0) - ST(i) )</td>
<td>11011 d00 : 1110 R ST(i)</td>
</tr>
<tr>
<td><strong>FSUBP</strong> - Subtract and Pop</td>
<td></td>
</tr>
<tr>
<td>( ST(0) := ST(0) - ST(i) )</td>
<td>11011 110 : 1110 1 ST(i)</td>
</tr>
<tr>
<td><strong>FSUBR</strong> - Reverse Subtract</td>
<td></td>
</tr>
<tr>
<td>( ST(0) := 32\text{-}bit \text{ memory} - ST(0) )</td>
<td>11011 000 : mod 101 r/m</td>
</tr>
</tbody>
</table>
## INSTRUCTION FORMATS AND ENCODINGS

### B.18 VMX INSTRUCTIONS

Table B-40 describes virtual-machine extensions (VMX).

#### Table B-40. Encodings for VMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVEPT—Invalid Cache EPT Mappings</td>
<td>01100110 00001111 00111000 10000000: mod reg r/m</td>
</tr>
<tr>
<td>INVPID—Invalid Cache VPID Mappings</td>
<td>01100110 00001111 00111000 10000001: mod reg r/m</td>
</tr>
<tr>
<td>VMCALL—Call to VM Monitor</td>
<td>00001111 00000001 11000001</td>
</tr>
<tr>
<td>VMCLEAR—Clear Virtual-Machine Control Structure</td>
<td>00001111 00000001 11000001</td>
</tr>
<tr>
<td>VMFUNC—Invoke VM Function</td>
<td>00001111 00000001 11000001</td>
</tr>
<tr>
<td>VMLAUNCH—Launch Virtual Machine</td>
<td>00001111 00000001 11000001</td>
</tr>
<tr>
<td>VMRESUME—Resume Virtual Machine</td>
<td>00001111 00000001 11000001</td>
</tr>
<tr>
<td>VMPTRLD—Load Pointer to Virtual-Machine Control Structure</td>
<td>00001111 11000001 11000001: mod 110 r/m</td>
</tr>
</tbody>
</table>
### Table B-38. Encodings for SMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETSEC—GETSEC leaf functions are selected by the value in EAX on input</td>
<td></td>
</tr>
<tr>
<td>GETSEC[CAPABILITIES]</td>
<td>00001111 00110111 (EAX= 0)</td>
</tr>
<tr>
<td>GETSEC[ENTERACCS]</td>
<td>00001111 00110111 (EAX= 2)</td>
</tr>
<tr>
<td>GETSEC(EXITAC)</td>
<td>00001111 00110111 (EAX= 3)</td>
</tr>
<tr>
<td>GETSEC[SENTER]</td>
<td>00001111 00110111 (EAX= 4)</td>
</tr>
<tr>
<td>GETSEC(SEXIT)</td>
<td>00001111 00110111 (EAX= 5)</td>
</tr>
<tr>
<td>GETSEC[PARAMETERS]</td>
<td>00001111 00110111 (EAX= 6)</td>
</tr>
<tr>
<td>GETSEC[SMCTRL]</td>
<td>00001111 00110111 (EAX= 7)</td>
</tr>
<tr>
<td>GETSEC[wAKEUP]</td>
<td>00001111 00110111 (EAX= 8)</td>
</tr>
</tbody>
</table>

### Table B-39. Encodings for VMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETSEC leaf functions are selected by a valid value in EAX on input</td>
<td></td>
</tr>
<tr>
<td>GETSEC[CAPABILITIES]</td>
<td>00001111 01111001: 11 reg2 reg1</td>
</tr>
<tr>
<td>GETSEC[ENTERACCS]</td>
<td>00001111 01111000: mod r32 r/m</td>
</tr>
<tr>
<td>GETSEC(EXITAC)</td>
<td>00001111 01111000: 11 reg2 reg1</td>
</tr>
<tr>
<td>GETSEC[SENTER]</td>
<td>00001111 01111000: mod r64 r/m</td>
</tr>
<tr>
<td>GETSEC[SEXIT]</td>
<td>00001111 01111000: 11 reg2 reg1</td>
</tr>
<tr>
<td>GETSEC[PARAMETERS]</td>
<td>00001111 01111000: mod r64 r/m</td>
</tr>
<tr>
<td>GETSEC[SMCTRL]</td>
<td>00001111 01111000: 11 reg2 reg1</td>
</tr>
<tr>
<td>GETSEC[wAKEUP]</td>
<td>00001111 01111000: mod r64 r/m</td>
</tr>
</tbody>
</table>

### Table B-40. Encodings for SMX Instructions

<table>
<thead>
<tr>
<th>Instruction and Format</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETSEC—GETSEC leaf functions are selected by the value in EAX on input</td>
<td></td>
</tr>
<tr>
<td>GETSEC[CAPABILITIES]</td>
<td>00001111 00110111 (EAX= 0)</td>
</tr>
<tr>
<td>GETSEC[ENTERACCS]</td>
<td>00001111 00110111 (EAX= 2)</td>
</tr>
<tr>
<td>GETSEC(EXITAC)</td>
<td>00001111 00110111 (EAX= 3)</td>
</tr>
<tr>
<td>GETSEC[SENTER]</td>
<td>00001111 00110111 (EAX= 4)</td>
</tr>
<tr>
<td>GETSEC(SEXIT)</td>
<td>00001111 00110111 (EAX= 5)</td>
</tr>
<tr>
<td>GETSEC[PARAMETERS]</td>
<td>00001111 00110111 (EAX= 6)</td>
</tr>
<tr>
<td>GETSEC[SMCTRL]</td>
<td>00001111 00110111 (EAX= 7)</td>
</tr>
<tr>
<td>GETSEC[wAKEUP]</td>
<td>00001111 00110111 (EAX= 8)</td>
</tr>
</tbody>
</table>
The two tables in this appendix itemize the Intel C/C++ compiler intrinsics and functional equivalents for the Intel MMX technology, SSE, SSE2, SSE3, and SSSE3 instructions.

There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics. Please refer to http://www.intel.com/support/performancetools/.

Table C-1 presents simple intrinsics and Table C-2 presents composite intrinsics. Some intrinsics are “composites” because they require more than one instruction to implement them.

Intel C/C++ Compiler intrinsic names reflect the following naming conventions:

_mm_<intrin_op>_<suffix>

where:

<intrin_op> Indicates the intrinsics basic operation; for example, add for addition and sub for subtraction
<suffix> Denotes the type of data operated on by the instruction. The first one or two letters of each suffix denotes whether the data is packed (p), extended packed (ep), or scalar (s).

The remaining letters denote the type:

s single-precision floating-point
d double precision floating-point
i128 signed 128-bit integer
i64 signed 64-bit integer
u64 unsigned 64-bit integer
i32 signed 32-bit integer
u32 unsigned 32-bit integer
i16 signed 16-bit integer
u16 unsigned 16-bit integer
i8 signed 8-bit integer
u8 unsigned 8-bit integer

The variable r is generally used for the intrinsic’s return value. A number appended to a variable name indicates the element of a packed object. For example, r0 is the lowest word of r.

The packed values are represented in right-to-left order, with the lowest value being used for scalar operations. Consider the following example operation:

double a[2] = {1.0, 2.0};
_mm128d t = _mm_load_pd(a);

The result is the same as either of the following:

_mm128d t = _mm_set_pd(2.0, 1.0);
_mm128d t = _mm_setr_pd(1.0, 2.0);

In other words, the XMM register that holds the value t will look as follows:
The "scalar" element is 1.0. Due to the nature of the instruction, some intrinsics require their arguments to be immediates (constant integer literals).

To use an intrinsic in your code, insert a line with the following syntax:

    data_type intrinsic_name (parameters)

Where:

- **data_type** is the return data type, which can be either void, int, __m64, __m128, __m128d, or __m128i. Only the _mm_empty intrinsic returns void.
- **intrinsic_name** is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of in-lining the actual instruction.
- **parameters** represents the parameters required by each intrinsic.

### C.1 SIMPLE INTRINSICS

**NOTE**


<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD</td>
<td>__m128d _mm_add_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ADDPS</td>
<td>__m128d _mm_add_ps(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ADDSD</td>
<td>__m128d _mm_add_sd(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>ADDSS</td>
<td>__m128d _mm_add_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td>__m128d _mm_addsub_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td>__m128d _mm_addsub_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>AESDEC</td>
<td>__m128d _mm_aesdec(__m128, __m128)</td>
</tr>
<tr>
<td>AESDECLAST</td>
<td>__m128d _mm_aesdeclast(__m128, __m128)</td>
</tr>
<tr>
<td>AESENCE</td>
<td>__m128d _mm_aesenc(__m128, __m128)</td>
</tr>
<tr>
<td>AESENCLAST</td>
<td>__m128d _mm_aesenclast(__m128, __m128)</td>
</tr>
<tr>
<td>AESIMC</td>
<td>__m128d _mm_aesimc(__m128)</td>
</tr>
<tr>
<td>AESKEYGENASSIST</td>
<td>__m128d _mm_aesimc(__m128, const int)</td>
</tr>
<tr>
<td>ANDNPD</td>
<td>__m128d _mm_andnot_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ANDNPS</td>
<td>__m128d _mm_andnot_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>ANDPD</td>
<td>__m128d _mm_and_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>ANDPS</td>
<td>__m128d _mm_and_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>BLENDPD</td>
<td>__m128d _mm_blend_pd(__m128d v1, __m128d v2, const int mask)</td>
</tr>
<tr>
<td>BLENDPS</td>
<td>__m128d _mm_blend_ps(__m128d v1, __m128d v2, const int mask)</td>
</tr>
<tr>
<td>BLENDVPD</td>
<td>__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3)</td>
</tr>
<tr>
<td>BLENDVPS</td>
<td>__m128d _mm_blendv_ps(__m128d v1, __m128d v2, __m128d v3)</td>
</tr>
<tr>
<td>CLFLUSH</td>
<td>void _mm_clflush(void const *p)</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMPPD</strong></td>
<td>__m128d_mm_cmpeq_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmplt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmple_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpgt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmple_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpneq_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpnlt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpngt_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpnge_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpord_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpunord_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td><strong>CMPPS</strong></td>
<td>__m128_mm_cmpeq_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmplt_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmple_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpgt_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpge_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpneq_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpnlt_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpngt_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpnge_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpord_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpunord_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td><strong>CMPSD</strong></td>
<td>__m128d_mm_cmpeq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmplt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmple_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpgt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpge_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpneq_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpnlt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpngt_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpnge_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpord_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d_mm_cmpunord_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td><strong>CMPSS</strong></td>
<td>__m128_mm_cmpeq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmplt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmple_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpgt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td></td>
<td>__m128_mm_cmpge_ss(__m128 a, __m128 b)</td>
</tr>
</tbody>
</table>
Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>_m128_mm_cmpneq_ss(_m128 a, _m128 b)</td>
<td>__m128 _mm_cmpneq_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>_m128_mm_cmpnlt_ss(_m128 a, _m128 b)</td>
<td>__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>_m128_mm_cmpnle_ss(_m128 a, _m128 b)</td>
<td>__m128 _mm_cmpnle_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>_m128_mm_cmpngt_ss(_m128 a, _m128 b)</td>
<td>__m128 _mm_cmpngt_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>_m128_mm_cmpnge_ss(_m128 a, _m128 b)</td>
<td>__m128 _mm_cmpnge_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>_m128_mm_cmpord_ss(_m128 a, _m128 b)</td>
<td>__m128 _mm_cmpord_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>COMISD</td>
<td>int _mm_comieq_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comilt_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comile_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comigtd_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comige_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comineq_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>COMISS</td>
<td>int _mm_comieq_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comilt_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comile_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comigtd_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comige_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td></td>
<td>int _mm_comineq_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td>CRC32</td>
<td>unsigned int _mm_crc32_u8(unsigned int crc, unsigned char data)</td>
</tr>
<tr>
<td></td>
<td>unsigned int _mm_crc32_u16(unsigned int crc, unsigned short data)</td>
</tr>
<tr>
<td></td>
<td>unsigned int _mm_crc32_u32(unsigned int crc, unsigned int data)</td>
</tr>
<tr>
<td></td>
<td>unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)</td>
</tr>
<tr>
<td>CVTDQ2PD</td>
<td>__m128d _mm_cvtepi32_pd(_m128d a)</td>
</tr>
<tr>
<td>CVTDQ2PS</td>
<td>__m128d _mm_cvtepi32_ps(_m128d a)</td>
</tr>
<tr>
<td>CVTPD2DQ</td>
<td>__m128d _mm_cvtpd_epi32(_m128d a)</td>
</tr>
<tr>
<td>CVTPD2PI</td>
<td>__m64d _mm_cvtpd_pi32(_m128d a)</td>
</tr>
<tr>
<td>CVTPD2PS</td>
<td>__m128d _mm_cvtpd_ps(_m128d a)</td>
</tr>
<tr>
<td>CVTP2PD</td>
<td>__m128d _mm_cvtpi32_pd(_m64d a)</td>
</tr>
<tr>
<td>CVTP2PS</td>
<td>__m128d _mm_cvt_pi32ps(_m128 d, _m64d b)</td>
</tr>
<tr>
<td>CVTPS2DQ</td>
<td>__m128d _mm_cvtps_epi32(_m128d a)</td>
</tr>
<tr>
<td>CVTPS2PD</td>
<td>__m128d _mm_cvtps_pd(_m128d a)</td>
</tr>
<tr>
<td>CVTPS2PI</td>
<td>__m64d _mm_cvtps_pi32ps(_m128 d, _m64d b)</td>
</tr>
<tr>
<td>CVTSD2SI</td>
<td>int _mm_cvtsd_si32(_m128d a)</td>
</tr>
<tr>
<td>CVTSD2SS</td>
<td>__m128d _mm_cvtsd_ss(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>CVTS2SD</td>
<td>__m128d _mm_cvtsi32_sd(_m128d a, int b)</td>
</tr>
<tr>
<td>CVTS2SS</td>
<td>__m128d _mm_cvtsi32_ss(_m128d a, int b)</td>
</tr>
<tr>
<td>CVTSS2SD</td>
<td>__m128d _mm_cvtsi64_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cvtsi64_ss(_m128d a, _m128d b)</td>
</tr>
<tr>
<td></td>
<td>__m128d _mm_cvtsi64_ss(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
| CVTSS2SI | int _mm_cvt_ss2si(__m128 a)  
         | int _mm_cvtsi32((__m128 a)  
|         | __m128i _mm_cvttss_epi32((__m128d a) |
| CVTTPD2DQ| __m128i _mm_cvttpd_epi32((__m128d a) |
| CVTTPD2PI| __m64 _mm_cvttpd_pl32((__m128d a) |
| CVTTPS2DQ| __m128i _mm_cvttps_epi32((__m128 a) |
| CVTTPS2PI| __m64 _mm_cvttps_epi32((__m128 a) |
| CVTTSD2SI| __m64 _mm_cvttss_epi32((__m128 a) |
| CVTTSS2SI| int _mm_cvttss(__m128 a)  
         | int _mm_cvtss(__m128 a) |
|         | __m128i _mm_cvtsi32((__m128 a)  
|         | __m64 _mm_cvtsi32((__m128 a) |
|         | int _mm_cvtsi32((__m64 m) |
| DIVPD   | __m128d _mm_div_pd((__m128 d, __m128 b)  
| DIVPS   | __m128d _mm_div_ps((__m128 d, __m128 b)  
| DIVSD   | __m128d _mm_div_sd((__m128 d, __m128 b)  
| DIVSS   | __m128d _mm_div_sd((__m128 d, __m128 b)  
| DPPD    | __m128d _mm_dp_pd((__m128 d, __m128 b, const int mask)  
| DPPS    | __m128d _mm_dp_ps((__m128 d, __m128 b, const int mask)  
| EMMS    | void _mm_empty() |
| EXTRACTPS| int _mm_extract_ps((__m128 src, const int ndx) |
| HADDPD  | __m128d _mm_hadd_pd((__m128 d, __m128 b)  
| HADDP   | __m128d _mm_hadd_ps((__m128 d, __m128 b)  
| HSUBPD  | __m128d _mm_hsub_pd((__m128 d, __m128 b)  
| HSUBPS  | __m128d _mm_hsub_ps((__m128 d, __m128 b)  
| INSERTPS| __m128d _mm_insert_ps((__m128 dst, __m128 src, const int ndx) |
| LDDQU   | __m128d _mm_loadl_epi64((__m128 src, const int *p) |
| LDMXCSR | __mm_setcsr(unsigned int i) |
| LFENCE  | void _mm_lfence(void) |
| MASKMOVQ | void _mm_maskmove_u32((__m128 d, __m128 b) |
| MAXPD   | __m128d _mm_max_pd((__m128 d, __m128 b)  
| MAXPS   | __m128d _mm_max_ps((__m128 d, __m128 b)  
| MAXSD   | __m128d _mm_max_sd((__m128 d, __m128 b) |
| MAXSS   | __m128d _mm_max_ss((__m128 d, __m128 b) |
| MFENCE  | void _mm_mfence(void) |
| MINPD   | __m128d _mm_min_pd((__m128 d, __m128 b)  
| MINPS   | __m128d _mm_min_ps((__m128 d, __m128 b)  
| MINSD   | __m128d _mm_min_sd((__m128 d, __m128 b)  
| MINSS   | __m128d _mm_min_ss((__m128 d, __m128 b)  
| MONITOR | void _mm_monitor(void const *, unsigned extensions, unsigned hints) |
| MOVAPD  | void _mm_store_pd(double * p)  
<pre><code>     | void _mm_store_postpd(double *p, __m128d a) |
</code></pre>
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
</table>
| MOVAPS   | __m128_mm_load_ps(float * p)  
void_mm_store_ps(float *p, __m128 a) |
| MOVD     | __m128i_mm_cvtsi32_si128(int a)  
int_mm_cvtsi128_si32(__m128i a)  
__m64_mm_cvtsi32_si64(int a)  
int_mm_cvtsi64_si32(__m64 a) |
| MOVDDUP  | __m128d_mm_movedup_pd(__m128d a)  
__m128d_mm_loaddup_pd(double const * dp) |
| MOVDQA   | __m128i_mm_load_si128(__m128i * p)  
void_mm_store_si128(__m128i *p, __m128i a) |
| MOVDQU   | __m128i_mm_loadu_si128(__m128i * p)  
void_mm_storeu_si128(__m128i *p, __m128i a) |
| MOVDQ2Q  | __m64_mm_movepi64_pi64(__m128i a) |
| MOVHLPS  | __m128_mm_movelh_ps(__m128 a, __m128 b) |
| MOVHPD   | __m128d_mm_loadh_pd(__m128d a, double * p)  
void_mm_storeh_pd(double * p, __m128d a) |
| MOVHPS   | __m128l_mm_loadh_pi(__m128 a, __m64 * p)  
void_mm_storeh_pi(__m64 * p, __m128 a) |
| MOVLPD   | __m128d_mm_loadl_pd(__m128d a, double * p)  
void_mm_storel_pd(double * p, __m128d a) |
| MOVLPS   | __m128l_mm_loadl_pi(__m128 a, __m64 * p)  
void_mm_storel_pi(__m64 * p, __m128 a) |
| MOVLHPS  | __m128l_mm_movelh_ps(__m128 a, __m128 b) |
| MOVMSKPD | int_mm_movemask_pd(__m128d a) |
| MOVMSKPS | int_mm_movemask_ps(__m128 a) |
| MOVNTDQA | __m128i_mm_stream_load_si128(__m128i *p) |
| MOVNTDQ  | void_mm_stream_si128(__m128i * p, __m128i a) |
| MOVNTPD  | void_mm_stream_pd(double * p, __m128d a) |
| MOVNTPS  | void_mm_stream_ps(float * p, __m128 a) |
| MOVNTI   | void_mm_stream_si32(int * p, int a) |
| MOVNTQ   | void_mm_stream_pi(__m64 * p, __m128 a) |
| MOVQ     | __m128i_mm_loadl_epi64(__m128i * p)  
void_mm_storel_epi64(__m128i * p, __m128i a)  
__m128i_mm_move_epi64(__m128i a) |
| MOVQ2DQ  | __m128i_mm_movpi64_epi64(__m64 a) |
| MOVSD    | __m128d_mm_load_sd(double * p)  
void_mm_store_sd(double * p, __m128d a)  
__m128d_mm_move_sd(__m128d a, __m128d b) |
| MOVS DUP | __m128_mm_movedup_ps(__m128 a) |
| MOVS LDUP| __m128_mm_moveldup_ps(__m128 a) |
| MOVS SS  | __m128_mm_load_ss(float * p) |
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>void_mm_store_ss(float * p, __m128 a)</td>
<td>__m128 _mm_move_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>__m128d _mm_loadu_pd(double * p)</td>
<td>void_mm_storeu_pd(double *p, __m128d a)</td>
</tr>
<tr>
<td>__m128 _mm_loadu_ps(float * p)</td>
<td>void_mm_storeu_ps(float *p, __m128 a)</td>
</tr>
<tr>
<td>__m128d _mm_loadu_pd(__m128d a, __m128d b)</td>
<td>MOVUPD __m128d _mm_loadu_pd(double * p)</td>
</tr>
<tr>
<td>__m128 _mm_loadu_ps(__m128 a)</td>
<td>MOVUPS __m128 _mm_loadu_ps(float * p)</td>
</tr>
<tr>
<td>__m128d _mm_loadu_ps(__m128 a, __m128d b)</td>
<td>MPSADBW __m128i _mm_mpsadbw_epi8(__m128i s1, __m128i s2, const int mask)</td>
</tr>
<tr>
<td>__m128d _mm_mul_pd(__m128d a, __m128d b)</td>
<td>MULPD __m128d _mm_mul_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>__m128 _mm_mul_ss(__m128 a, __m128 b)</td>
<td>MULPS __m128 _mm_mul_ss(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>__m128d _mm_mul_sd(__m128d a, __m128d b)</td>
<td>MULSD __m128d _mm_mul_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>__m128 _mm_mul_sd(__m128d a, __m128d b)</td>
<td>MULSS __m128 _mm_mul_sd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>void _mm_mwait(unsigned extensions, unsigned hints)</td>
<td>Mwait</td>
</tr>
<tr>
<td>__m128d _mm_or_pd(__m128d a, __m128d b)</td>
<td>ORPD __m128d _mm_or_pd(__m128d a, __m128d b)</td>
</tr>
<tr>
<td>__m128 _mm_or_ps(__m128 a, __m128 b)</td>
<td>ORPS __m128 _mm_or_ps(__m128 a, __m128 b)</td>
</tr>
<tr>
<td>__m64d _mm_abs_epi8(__m64 a)</td>
<td>PABSB __m64 _mm_abs_epi8(__m64 a)</td>
</tr>
<tr>
<td>__m128i _mm_abs_epi8(__m128i a)</td>
<td>PABSD __m128i _mm_abs_epi8(__m128i a)</td>
</tr>
<tr>
<td>__m64d _mm_abs_epi32(__m64 a)</td>
<td>PABSW __m64d _mm_abs_epi32(__m64 a)</td>
</tr>
<tr>
<td>__m128i _mm_abs_epi32(__m128i a)</td>
<td>PACKSSWB __m128i _mm_abs_epi32(__m128i a)</td>
</tr>
<tr>
<td>__m64d _mm_abs_epi16(__m64 a)</td>
<td>PACKSSDW __m128i _mm_abs_epi16(__m64 a)</td>
</tr>
<tr>
<td>__m128i _mm_abs_epi16(__m128i a)</td>
<td>PACKUSDW __m128i _mm_abs_epi16(__m128i a)</td>
</tr>
<tr>
<td>__m64d _mm_abs_pu8(__m64 a)</td>
<td>PACKUSWB __m128i _mm_abs_pu8(__m64 a)</td>
</tr>
<tr>
<td>__m128i _mm_add_epi8(__m128i m1, __m128i m2)</td>
<td>PADDB __m128i _mm_add_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>__m64d _mm_add_epi8(__m64 d1, __m64 d2)</td>
<td>PADDW __m64 _mm_add_epi8(__m64 d1, __m64 d2)</td>
</tr>
<tr>
<td>__m128i _mm_add_epi16(__m128i m1, __m128i m2)</td>
<td>PADDW __m128i _mm_add_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>__m64d _mm_add_epi16(__m64 d1, __m64 d2)</td>
<td>PADDQ __m64 _mm_add_epi16(__m64 d1, __m64 d2)</td>
</tr>
<tr>
<td>__m128i _mm_add_epi64(__m128i m1, __m128i m2)</td>
<td>PADDQ __m128i _mm_add_epi64(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>__m64d _mm_add_epi64(__m64 d1, __m64 d2)</td>
<td>ADDQ __m64 _mm_add_epi64(__m64 d1, __m64 d2)</td>
</tr>
<tr>
<td>__m128i _mm_add_epi8(__m128i m1, __m128i m2)</td>
<td>ADDSB __m128i _mm_add_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>__m64d _mm_add_epi8(__m64 d1, __m64 d2)</td>
<td>ADDSB __m64 _mm_add_epi8(__m64 d1, __m64 d2)</td>
</tr>
<tr>
<td>__m128i _mm_add_epi16(__m128i m1, __m128i m2)</td>
<td>ADDSD __m128i _mm_add_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>__m64d _mm_add_epi16(__m64 d1, __m64 d2)</td>
<td>ADDSD __m64 _mm_add_epi16(__m64 d1, __m64 d2)</td>
</tr>
<tr>
<td>__m128i _mm_add_epi64(__m128i m1, __m128i m2)</td>
<td>ADDUSB __m128i _mm_add_epi64(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>__m64d _mm_add_epi64(__m64 d1, __m64 d2)</td>
<td>ADDUSB __m64 _mm_add_epi64(__m64 d1, __m64 d2)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PADDUSW</td>
<td>__m128i_mm_adds_epu16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PADDUSW</td>
<td>__m64_mm_adds_pu16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PALIGNR</td>
<td>__m64_mm_alignr_pi8(__m64 a, __m64 b, int n)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_alignr_epi8(__m128i a, __m128i b, int n)</td>
</tr>
<tr>
<td>PAND</td>
<td>__m128i_mm_and_si128(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PAND</td>
<td>__m64_mm_and_si64(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PANDN</td>
<td>__m128i_mm_andnot_si128(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PANDN</td>
<td>__m64_mm_andnot_si64(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PAUSE</td>
<td>void_mm_pause(void)</td>
</tr>
<tr>
<td>PAVGB</td>
<td>__m128i_mm_avg_epu8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PAVGB</td>
<td>__m64_mm_avg_pu8(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PAVGW</td>
<td>__m64_mm_avg_epu16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PAVGW</td>
<td>__m64_mm_avg_pu16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PBLENDVB</td>
<td>__m128i_mm_blendv_epi(__m128i v1, __m128i v2, __m128i mask)</td>
</tr>
<tr>
<td>PBLENDW</td>
<td>__m128i_mm_blend_epi16(__m128i v1, __m128i v2, const int mask)</td>
</tr>
<tr>
<td>PCLMULQDQ</td>
<td>__m128i_mm_clmulepi64_s_i128(__m128i, __m128i, const int)</td>
</tr>
<tr>
<td>PCMPEQB</td>
<td>__m128i_mm_cmpseq_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPEQB</td>
<td>__m64_mm_cmpseq_pi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPEQQ</td>
<td>__m128i_mm_cmpseq_epi64(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PCMPEQW</td>
<td>__m128i_mm_cmpseq_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPEQW</td>
<td>__m64_mm_cmpseq_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPEQD</td>
<td>__m128i_mm_cmpseq_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPEQD</td>
<td>__m64_mm_cmpseq_pi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPESTRI</td>
<td>int_mm_cmpести (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpестра (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpестрц (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpестроз (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td>PCMPESTRM</td>
<td>__m128i_mm_cmpestrm(__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestr (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestrc (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestrosso (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestroz (__m128i a, int ia, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td>PCMPGTB</td>
<td>__m128i_mm_cmpgt_epi8(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPGTB</td>
<td>__m64_mm_cmpgt_pi8(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPGTW</td>
<td>__m128i_mm_cmpgt_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPGTW</td>
<td>__m64_mm_cmpgt_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PCMPGTD</td>
<td>__m128i_mm_cmpgt_epi32(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PCMPGTD</td>
<td>__m64_mm_cmpgt_pi32(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCMPISTRI</td>
<td>__m128i_mm_cmpestrm(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestra(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestrc(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestro(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestrs(__m128i a, int la, __m128i b, int lb, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpestrz(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td>PCMPISTRM</td>
<td>__m128i_mm_cmpistrm(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpistra(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpistrc(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpistro(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpistrs(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td></td>
<td>int_mm_cmpistrz(__m128i a, __m128i b, const int mode)</td>
</tr>
<tr>
<td>PCMPGTQ</td>
<td>__m128i_mm_cmpgt_epi64(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PEXTRB</td>
<td>int_mm_extract_epi8(__m128i src, const int ndx)</td>
</tr>
<tr>
<td>PEXTRD</td>
<td>int_mm_extract_epi32(__m128i src, const int ndx)</td>
</tr>
<tr>
<td>PEXTRQ</td>
<td>__int64_mm_extract_epi64(__m128i src, const int ndx)</td>
</tr>
<tr>
<td>PEXTRW</td>
<td>int_mm_extract_epi16(__m128i a, int n)</td>
</tr>
<tr>
<td>PEXTRW</td>
<td>int_mm_extract_epi16(__m64 a, int n)</td>
</tr>
<tr>
<td>PHADD</td>
<td>__m64_mm_hadd_pi32(__m64 a, __m64 b)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_hadd_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PHADDW</td>
<td>__m64_mm_hadd_epi32(__m64 a, __m64 b)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_hadd_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PHMINPOSUW</td>
<td>__m128i_mm_minpos_epi64(__m128i packed_words)</td>
</tr>
<tr>
<td>PHSUBD</td>
<td>__m64_mm_hsub_pi32(__m64 a, __m64 b)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_hsub_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PHSUBSW</td>
<td>__m64_mm_hsubs_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_hsubs_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PHSUBW</td>
<td>__m64_mm_hsub_epi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td></td>
<td>__m128i_mm_hsub_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PINSRB</td>
<td>__m128i_mm_insert_epi8(__m128i s1, int s2, const int ndx)</td>
</tr>
<tr>
<td>PINSRD</td>
<td>__m128i_mm_insert_epi32(__m128i s2, int s, const int ndx)</td>
</tr>
<tr>
<td>PINSRQ</td>
<td>__m128i_mm_insert_epi64(__m128i s2, __int64 s, const int ndx)</td>
</tr>
<tr>
<td>PINSRW</td>
<td>__m128i_mm_insert_epi16(__m128i a, int d, int n)</td>
</tr>
<tr>
<td>PINSRW</td>
<td>__m64_mm_insert_pi16(__m64 a, int d, int n)</td>
</tr>
<tr>
<td>PMADDSUBSW</td>
<td>__m64_mm_maddubs_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMADDWD</td>
<td>__m64_mm_madd_epi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMADDSUBW</td>
<td>__m64_mm_maddubs_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMADDWD</td>
<td>__m64_mm_madd_epi16(__m64 a, __m64 b)</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMAXSB</td>
<td>__m128i_mm_max_epi8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXSD</td>
<td>__m128i_mm_max_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXSW</td>
<td>__m128i_mm_max_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXSW</td>
<td>__m64_mm_max_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMAXUB</td>
<td>__m128i_mm_max_epu8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXUDB</td>
<td>__m128i_mm_max_epu32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMAXUW</td>
<td>__m128i_mm_max_epu16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSB</td>
<td>__m128i_mm_min_epi8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSD</td>
<td>__m128i_mm_min_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSW</td>
<td>__m128i_mm_min_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINSW</td>
<td>__m64_mm_min_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMINUB</td>
<td>__m128i_mm_min_epu8(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINUB</td>
<td>__m64_mm_min_pu8(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMINUD</td>
<td>__m128i_mm_min_epu32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMINUW</td>
<td>__m128i_mm_min_epu16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMOVMSKB</td>
<td>int_mm_movemask_epi8(__m128i a)</td>
</tr>
<tr>
<td>PMOVMSKB</td>
<td>int_mm_movemask_pi8(__m64 a)</td>
</tr>
<tr>
<td>PMOVXSBW</td>
<td>__m128i_mm_cvtepi8_epi16(__m128i a)</td>
</tr>
<tr>
<td>PMOVXSBD</td>
<td>__m128i_mm_cvtepi8_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVXSBQ</td>
<td>__m128i_mm_cvtepi8_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVXWQ</td>
<td>__m128i_mm_cvtepi16_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVXQ</td>
<td>__m128i_mm_cvtepi32_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVZXBW</td>
<td>__m128i_mm_cvtepu8_epi16(__m128i a)</td>
</tr>
<tr>
<td>PMOVZXBQ</td>
<td>__m128i_mm_cvtepu8_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVZWB</td>
<td>__m128i_mm_cvtepu8_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVZWR</td>
<td>__m128i_mm_cvtepu16_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMOVZXWQ</td>
<td>__m128i_mm_cvtepu16_epi32(__m128i a)</td>
</tr>
<tr>
<td>PMOVZXDQ</td>
<td>__m128i_mm_cvtepu32_epi64(__m128i a)</td>
</tr>
<tr>
<td>PMULDQ</td>
<td>__m128i_mm_mul_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMULHRSW</td>
<td>__m64_mm_mulhrs_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMULHUW</td>
<td>__m128i_mm_mulhi_epi16(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMULHUW</td>
<td>__m64_mm_mulhi_pi16(__m64 a, __m64 b)</td>
</tr>
<tr>
<td>PMULHW</td>
<td>__m128i_mm_mulhi_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PMULLUD</td>
<td>__m128i_mm_mullo_epi32(__m128i a, __m128i b)</td>
</tr>
<tr>
<td>PMULLW</td>
<td>__m128i_mm_mullo_epi16(__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PMULLH</td>
<td>__m64_mm_mullo_pi16(__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Intrinsic</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
| PMULUDQ  | __m64 __mm_mul_su32(__m64 m1, __m64 m2)  
            | __m128i __mm_mul_epu32(__m128i m1, __m128i m2)  |
| POPCNT   | int __mm_popcnt_u32(unsigned int a)  
            | int64_t __mm_popcnt_u64(unsigned __int64 a)  |
| POR      | __m64 __mm_or_si64(__m64 m1, __m64 m2)  
            | __m128i __mm_or_si128(__m128i m1, __m128i m2)  |
| PREFETCHH| void __mm_prefetch(char *a, int sel)  |
| PSADBW   | __m128i __mm_sad_epu8(__m128i a, __m128i b)  
            | __m64 __mm_sad_pl8(__m64 a, __m64 b)  |
| PSHUFB   | __m128i __mm_shuffle_pi8 (__m64 a, __m64 b)  
            | __m128i __mm_shuffle_epi8 (__m128i a, __m128i b)  |
| PSHUFD   | __m128i __mm_shuffle_epi32(__m128i a, int n)  |
| PSHUFLW  | __m128i __mm_shufflelo_epi16(__m128i a, int n)  |
| PSHUFW   | __m64 __mm_shuffle_pi16(__m64 a, int n)  |
| PSIGNB   | __m64 __mm_sign_pi8 (__m64 a, __m64 b)  
            | __m128i __mm_sign_epi8 (__m128i a, __m128i b)  |
| PSIGND   | __m64 __mm_sign_pi32 (__m64 a, __m64 b)  
            | __m128i __mm_sign_epi32 (__m128i a, __m128i b)  |
| PSIGNW   | __m64 __mm_sign_pi16 (__m64 a, __m64 b)  
            | __m128i __mm_sign_epi16 (__m128i a, __m128i b)  |
| PSLLLW   | __m128i __mm_sll_epi16(__m128i m, __m128i count)  |
| PSLLEW   | __m128i __mm_slli_epi16(__m128i m, int count)  |
| PSLLEW   | __m64 __mm_slli_epi16(__m64 m, __m64 count)  
            | __m64 __mm_sll_epi16(__m64 m, __m64 count)  |
| PSLLD    | __m128i __mm_sll_epi32(__m128i m, int count)  
            | __m128i __mm_slli_epi32(__m128i m, __m128i count)  |
| PSLLD    | __m64 __mm_slli_epi32(__m64 m, int count)  
            | __m64 __mm_slli_epi32(__m64 m, __m64 count)  |
| PSLLQ    | __m64 __mm_slli_epi64(__m64 m, int count)  
            | __m64 __mm_sll_epi64(__m64 m, __m64 count)  |
| PSLLQ    | __m128i __mm_slli_epi64(__m128i m, __m128i count)  
            | __m128i __mm_sll_epi64(__m128i m, int count)  |
| PSLLDQ   | __m128i __mm_sll_epi128(__m128i m, int imm)  |
| PSRAW    | __m128i __mm_sra_epi16(__m128i m, __m128i count)  
            | __m128i __mm_srai_epi16(__m128i m, int count)  |
| PSRAW    | __m64 __mm_sra_epi16(__m64 m, __m64 count)  
            | __m64 __mm_srai_epi16(__m64 m, int count)  |
| PSRAD    | __m128i __mm_sra_epi32(__m128i m, __m128i count)  
<pre><code>        | __m128i __mm_srai_epi32(__m128i m, int count)  |
</code></pre>
<p>| PSRAD    | __m64 __mm_sra_epi32(__m64 m, __m64 count)  |</p>
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>m64</strong> mm_srai_pi32 (__m64 m, int count)</td>
<td></td>
</tr>
<tr>
<td>PSRLW</td>
<td><strong>m128i</strong> mm_srl_epi16 (__m128i m, __m128i count)</td>
</tr>
<tr>
<td></td>
<td><strong>m128i</strong> mm_srl_epi16 (__m128i m, int count)</td>
</tr>
<tr>
<td></td>
<td><strong>m64</strong> mm_srl_pi16 (__m64 m, __m64 count)</td>
</tr>
<tr>
<td></td>
<td><strong>m64</strong> mm_srl_pi16 (__m64 m, int count)</td>
</tr>
<tr>
<td>PSRLD</td>
<td><strong>m128i</strong> mm_srl_epi32 (__m128i m, __m128i count)</td>
</tr>
<tr>
<td></td>
<td><strong>m128i</strong> mm_srl_epi32 (__m128i m, int count)</td>
</tr>
<tr>
<td>PSRLD</td>
<td><strong>m64</strong> mm_srl_pi32 (__m64 m, __m64 count)</td>
</tr>
<tr>
<td>PSRLQ</td>
<td><strong>m128i</strong> mm_srl_epi64 (__m128i m, __m128i count)</td>
</tr>
<tr>
<td></td>
<td><strong>m128i</strong> mm_srl_epi64 (__m128i m, int count)</td>
</tr>
<tr>
<td>PSRLDQ</td>
<td><strong>m128i</strong> mm_srl_si128 (__m128i m, int imm)</td>
</tr>
<tr>
<td>PSUBB</td>
<td><strong>m128i</strong> mm_sub_epi8 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBB</td>
<td><strong>m64</strong> mm_sub_pi8 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBW</td>
<td><strong>m128i</strong> mm_sub_epi16 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBW</td>
<td><strong>m64</strong> mm_sub_pi16 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBD</td>
<td><strong>m128i</strong> mm_sub_epi32 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBD</td>
<td><strong>m64</strong> mm_sub_pi32 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBQ</td>
<td><strong>m128i</strong> mm_sub_epi64 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBQ</td>
<td><strong>m64</strong> mm_sub_pi64 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBSB</td>
<td><strong>m128i</strong> mm_subs_epi8 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBSB</td>
<td><strong>m64</strong> mm_subs_pi8 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBSW</td>
<td><strong>m128i</strong> mm_subs_epi16 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBSW</td>
<td><strong>m64</strong> mm_subs_pi16 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBUSB</td>
<td><strong>m128i</strong> mm_subs_epi8 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBUSB</td>
<td><strong>m64</strong> mm_subs_pi8 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PSUBUSW</td>
<td><strong>m128i</strong> mm_subs_epi16 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PSUBUSW</td>
<td><strong>m64</strong> mm_subs_pi16 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PTEST</td>
<td>int__ mm_testz_si128 (__m128i s1, __m128i s2)</td>
</tr>
<tr>
<td></td>
<td>int__ mm_testc_si128 (__m128i s1, __m128i s2)</td>
</tr>
<tr>
<td></td>
<td>int__ mm_testnzc_si128 (__m128i s1, __m128i s2)</td>
</tr>
<tr>
<td>PUNPCKHBW</td>
<td><strong>m64</strong> mm_unpackhi_pi8 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKHBW</td>
<td><strong>m128i</strong> mm_unpackhi_epi8 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKHDW</td>
<td><strong>m64</strong> mm_unpackhi_pi16 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKHDW</td>
<td><strong>m128i</strong> mm_unpackhi_epi16 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td><strong>m64</strong> mm_unpackhi_pi32 (__m64 m1, __m64 m2)</td>
</tr>
<tr>
<td>PUNPCKHDQ</td>
<td><strong>m128i</strong> mm_unpackhi_epi32 (__m128i m1, __m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLBD</td>
<td><strong>m64</strong> mm_unpacklo_epi8 (__m64 m1, __m64 m2)</td>
</tr>
</tbody>
</table>
### Table C-1. Simple Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUNPCKLBW</td>
<td>_m128i_mm_unpacklo_epi8 (_m128i m1, _m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLWD</td>
<td>_m64_mm_unpacklo_epi16(_m64 m1, _m64 m2)</td>
</tr>
<tr>
<td>PUNPCKLWD</td>
<td>_m128i_mm_unpacklo_epi16(_m128i m1, _m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>_m64_mm_unpacklo_epi32(_m64 m1, _m64 m2)</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>_m128i_mm_unpacklo_epi32(_m128i m1, _m128i m2)</td>
</tr>
<tr>
<td>PUNPCKLDQ</td>
<td>_m128i_mm_unpacklo_epi64(_m128i m1, _m128i m2)</td>
</tr>
<tr>
<td>PXOR</td>
<td>_m64_mm_xor_si64(_m64 m1, _m64 m2)</td>
</tr>
<tr>
<td>PXOR</td>
<td>_m128i_mm_xor_si128(_m128i m1, _m128i m2)</td>
</tr>
<tr>
<td>RCPPS</td>
<td>_m128_mm_rcp_ps(_m128 a)</td>
</tr>
<tr>
<td>RCPSS</td>
<td>_m128_mm_rcp_ss(_m128 a)</td>
</tr>
<tr>
<td>ROUNDPD</td>
<td>_m128_mm_round_pd(_m128d s1, int iRoundMode)</td>
</tr>
<tr>
<td>ROUNDPS</td>
<td>_m128_mm_round_ps(_m128 s1, int iRoundMode)</td>
</tr>
<tr>
<td>ROUNDSD</td>
<td>_m128d_mm_round_sd(_m128d dst, _m128d s1, int iRoundMode)</td>
</tr>
<tr>
<td>ROUNDSS</td>
<td>_m128d_mm_round_ss(_m128d dst, _m128d s1, int iRoundMode)</td>
</tr>
<tr>
<td>RSQRTPS</td>
<td>_m128_mm_rsqrt_ps(_m128 a)</td>
</tr>
<tr>
<td>RSQRTSS</td>
<td>_m128_mm_rsqrt_ss(_m128 a)</td>
</tr>
<tr>
<td>SFENCE</td>
<td>void_mm_sfence(void)</td>
</tr>
<tr>
<td>SHUFPD</td>
<td>_m128d_mm_shuffle_pd(_m128d a, _m128d b, unsigned int imm8)</td>
</tr>
<tr>
<td>SHUFPS</td>
<td>_m128d_mm_shuffle_ps(_m128 a, _m128 b, unsigned int imm8)</td>
</tr>
<tr>
<td>SQRTPD</td>
<td>_m128d_mm_sqrt_pd(_m128d a)</td>
</tr>
<tr>
<td>SQRTPS</td>
<td>_m128d_mm_sqrt_ps(_m128 a)</td>
</tr>
<tr>
<td>SQRTSD</td>
<td>_m128d_mm_sqrt_sd(_m128d a)</td>
</tr>
<tr>
<td>SQRTSS</td>
<td>_m128d_mm_sqrt_ss(_m128 a)</td>
</tr>
<tr>
<td>STMXCSR</td>
<td>_mm_getcsr(void)</td>
</tr>
<tr>
<td>SUBPD</td>
<td>_m128d_mm_sub_pd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>SUBPS</td>
<td>_m128d_mm_sub_ps(_m128 a, _m128 b)</td>
</tr>
<tr>
<td>SUBSD</td>
<td>_m128d_mm_sub_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>SUBSS</td>
<td>_m128d_mm_sub_ss(_m128 a, _m128 b)</td>
</tr>
<tr>
<td>UCOMISD</td>
<td>int_mm_ucomieq_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>UCOMILSD</td>
<td>int_mm_ucomilt_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>UCOMILD</td>
<td>int_mm_ucomile_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>UCOMIGT</td>
<td>int_mm_ucomigt_sd(_m128d a, _m128d b)</td>
</tr>
<tr>
<td>UCOMIGT</td>
<td>int_mm_ucomige_sd(_m128d a, _m128d b)</td>
</tr>
</tbody>
</table>
### C.2 COMPOSITE INTRINSICS

#### Table C-2. Composite Intrinsics

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128i _mm_set_epi64(__m64 q1, __m64 q0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w1,short w0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8, char w7,char w6, char w5, char w4, char w3, char w2,char w1, char w0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set1_epi64(__m64 q)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set1_epi32(int a)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set1_epi16(short a)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_set1_epi8(char a)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_setr_epi64(__m64 q1, __m64 q0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_setr_epi32(int i3, int i2, int i1, int i0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_setr_epi16(short w7,short w6, short w5, short w4, short w3, short w2, short w0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_setr_epi8(char w15,char w14, char w13, char w12, char w11, char w10, char w9, char w8,char w7,char w6, char w5, char w4, char w3, char w2,char w1, char w0)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128i _mm_setzero_si128()</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128 _mm_set_ps1(float w)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128 _mm_set1_ps(float w)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128cm _mm_set1_pd(double w)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128d _mm_set_sd(double w)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128d _mm_set_pd(double z, double y)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128 _mm_set_ps(float z, float y, float x, float w)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128d _mm_setr_pd(double z, double y)</td>
<td>(composite)</td>
</tr>
<tr>
<td>__m128 _mm_setr_ps(float z, float y, float x, float w)</td>
<td>(composite)</td>
</tr>
</tbody>
</table>
### Table C-2. Composite Intrinsics (Contd.)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(composite) __m128d_mm_setzero_pd(void)</td>
<td>(composite) __m128 __mm_setzero_ps(void)</td>
</tr>
<tr>
<td>MOVSD + shuffle __m128d_mm_load_pd(double * p)</td>
<td>MOVSS + shuffle __m128 __mm_load1_ps1(float * p)</td>
</tr>
<tr>
<td>MOVAPD + shuffle __m128d_mm_loadr_pd(double * p)</td>
<td>MOVEPS + shuffle __m128 __mm_loadr_ps(float * p)</td>
</tr>
<tr>
<td>MOVSD + shuffle void __mm_store1_pd(double *p, __m128d a)</td>
<td>MOVSS + shuffle void __mm_store1_ps(float *p, __m128 a)</td>
</tr>
<tr>
<td>MOVAPD + shuffle __mm_storer_pd(double *p, __m128d a)</td>
<td>MOVEPS + shuffle __mm_storer_ps(float *p, __m128 a)</td>
</tr>
</tbody>
</table>
NOTE: The Intel 64 and IA-32 Architectures Software Developer’s Manual consists of four volumes: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.
Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.
No product or component can be absolutely secure.
Your costs and results may vary.
You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.
All product plans and roadmaps are subject to change without notice.
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.
Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.
Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.
No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.
© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
## CONTENTS

### CHAPTER 1
**ABOUT THIS MANUAL**

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL .......................... 1-1  
1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE ........................................ 1-4  
1.3 NOTATIONAL CONVENTIONS ............................................................................ 1-6  
1.3.1 Bit and Byte Orders .................................................................................. 1-7  
1.3.2 Reserved Bits and Software Compatibility ................................................. 1-7  
1.3.3 Instruction Operands .................................................................................. 1-8  
1.3.4 Hexadecimal and Binary Numbers .............................................................. 1-8  
1.3.5 Segmented Addressing .............................................................................. 1-8  
1.3.6 Syntax for CPUID, CR, and MSR Values .................................................. 1-9  
1.3.7 Exceptions ................................................................................................. 1-10  
1.4 RELATED LITERATURE .................................................................................... 1-10

### CHAPTER 2
**SYSTEM ARCHITECTURE OVERVIEW**

2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE ..................................... 2-1  
2.1.1 Global and Local Descriptor Tables ............................................................ 2-3  
2.1.1.1 Global and Local Descriptor Tables in IA-32e Mode .......................... 2-4  
2.1.2 System Segments, Segment Descriptors, and Gates .................................. 2-4  
2.1.2.1 Gates in IA-32e Mode ....................................................................... 2-4  
2.1.3 Task-State Segments and Task Gates ........................................................ 2-5  
2.1.3.1 Task-State Segments in IA-32e Mode .............................................. 2-5  
2.1.4 Interrupt and Exception Handling ............................................................. 2-5  
2.1.4.1 Interrupt and Exception Handling IA-32e Mode ............................... 2-5  
2.1.5 Memory Management .............................................................................. 2-6  
2.1.5.1 Memory Management in IA-32e Mode ............................................ 2-6  
2.1.6 System Registers ....................................................................................... 2-6  
2.1.6.1 System Registers in IA-32e Mode ....................................................... 2-7  
2.1.7 Other System Resources ......................................................................... 2-7  
2.2 MODES OF OPERATION ............................................................................... 2-7  
2.2.1 Extended Feature Enable Register ............................................................ 2-9  
2.3 SYSTEM FLAGS AND FIELDS IN THE EFLAGS REGISTER ......................... 2-9  
2.3.1 System Flags and Fields in IA-32e Mode .................................................. 2-11  
2.4 MEMORY-MANAGEMENT REGISTERS ....................................................... 2-11  
2.4.1 Global Descriptor Table Register (GDTR) ............................................. 2-12  
2.4.2 Local Descriptor Table Register (LDTR) ................................................ 2-12  
2.4.3 IDTR Interrupt Descriptor Table Register ............................................. 2-12  
2.4.4 Task Register (TR) ................................................................................. 2-13  
2.5 CONTROL REGISTERS ............................................................................... 2-13  
2.5.1 CPUID Qualification of Control Register Flags ....................................... 2-20  
2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0) ....................... 2-20  
2.7 PROTECTION-KEY RIGHTS REGISTERS (PKRU AND IA32_PKRS) .......... 2-22  
2.8 SYSTEM INSTRUCTION SUMMARY ............................................................ 2-23  
2.8.1 Loading and Storing System Registers ....................................................... 2-24  
2.8.2 Verifying of Access Privileges .................................................................. 2-24  
2.8.3 Loading and Storing Debug Registers ..................................................... 2-25  
2.8.4 Invalidating Caches and TLBs ................................................................. 2-25  
2.8.5 Controlling the Processor ....................................................................... 2-26  
2.8.6 Reading Performance-Monitoring and Time-Stamp Counters ............... 2-26  
2.8.6.1 Reading Counters in 64-Bit Mode .................................................... 2-27  
2.8.7 Reading and Writing Model-Specific Registers ...................................... 2-27  
2.8.7.1 Reading and Writing Model-Specific Registers in 64-Bit Mode .......... 2-27  
2.8.8 Enabling Processor Extended States ....................................................... 2-27
## CONTENTS

### CHAPTER 3
**PROTECTED-MODE MEMORY MANAGEMENT**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 MEMORY MANAGEMENT OVERVIEW</td>
<td>3-1</td>
</tr>
<tr>
<td>3.2 USING SEGMENTS</td>
<td>3-2</td>
</tr>
<tr>
<td>3.2.1 Basic Flat Model</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2.2 Protected Flat Model</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2.3 Multi-Segment Model</td>
<td>3-4</td>
</tr>
<tr>
<td>3.2.4 Segmentation in IA-32e Mode</td>
<td>3-5</td>
</tr>
<tr>
<td>3.2.5 Paging and Segmentation</td>
<td>3-5</td>
</tr>
<tr>
<td>3.3 PHYSICAL ADDRESS SPACE</td>
<td>3-6</td>
</tr>
<tr>
<td>3.3.1 Intel 64 Processors and Physical Address Space</td>
<td>3-6</td>
</tr>
<tr>
<td>3.4 LOGICAL AND LINEAR ADDRESSES</td>
<td>3-6</td>
</tr>
<tr>
<td>3.4.1 Logical Address Translation in IA-32e Mode</td>
<td>3-7</td>
</tr>
<tr>
<td>3.4.2 Segment Selectors</td>
<td>3-7</td>
</tr>
<tr>
<td>3.4.3 Segment Registers</td>
<td>3-8</td>
</tr>
<tr>
<td>3.4.4 Segment Loading Instructions in IA-32e Mode</td>
<td>3-9</td>
</tr>
<tr>
<td>3.4.5 Segment Descriptors</td>
<td>3-9</td>
</tr>
<tr>
<td>3.4.5.1 Code- and Data-Segment Descriptor Types</td>
<td>3-12</td>
</tr>
<tr>
<td>3.5 SYSTEM DESCRIPTOR TYPES</td>
<td>3-13</td>
</tr>
<tr>
<td>3.5.1 Segment Descriptor Tables</td>
<td>3-14</td>
</tr>
<tr>
<td>3.5.2 Segment Descriptor Tables in IA-32e Mode</td>
<td>3-16</td>
</tr>
</tbody>
</table>

### CHAPTER 4
**PAGING**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 PAGING MODES AND CONTROL BITS</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1.1 Four Paging Modes</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1.2 Paging-Mode Enabling</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.3 Paging-Mode Modifiers</td>
<td>4-4</td>
</tr>
<tr>
<td>4.1.4 Enumeration of Paging Features by CPUID</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW</td>
<td>4-7</td>
</tr>
<tr>
<td>4.3 32-BIT PAGING</td>
<td>4-9</td>
</tr>
<tr>
<td>4.4 PAE PAGING</td>
<td>4-14</td>
</tr>
<tr>
<td>4.4.1 PDPT PE Registers</td>
<td>4-14</td>
</tr>
<tr>
<td>4.4.2 Linear-Address Translation with PAE Paging</td>
<td>4-15</td>
</tr>
<tr>
<td>4.5 4-LEVEL PAGING AND 5-LEVEL PAGING</td>
<td>4-20</td>
</tr>
<tr>
<td>4.5.1 Ordinary Paging and HLAT Paging</td>
<td>4-20</td>
</tr>
<tr>
<td>4.5.2 Use of CR3 with Ordinary 4-Level Paging and 5-Level Paging</td>
<td>4-20</td>
</tr>
<tr>
<td>4.5.3 Use of HLATP with HLAT 4-Level Paging and 5-Level Paging</td>
<td>4-21</td>
</tr>
<tr>
<td>4.5.4 Linear-Address Translation with 4-Level Paging and 5-Level Paging</td>
<td>4-22</td>
</tr>
<tr>
<td>4.5.5 Restart of HLAT Paging</td>
<td>4-33</td>
</tr>
<tr>
<td>4.6 ACCESS RIGHTS</td>
<td>4-33</td>
</tr>
<tr>
<td>4.6.1 Determination of Access Rights</td>
<td>4-33</td>
</tr>
<tr>
<td>4.6.2 Protection Keys</td>
<td>4-36</td>
</tr>
<tr>
<td>4.7 PAGE-FAULT EXCEPTIONS</td>
<td>4-37</td>
</tr>
<tr>
<td>4.8 ACCESS AND DIRTY FLAGS</td>
<td>4-39</td>
</tr>
<tr>
<td>4.9 PAGING AND MEMORY TYPING</td>
<td>4-39</td>
</tr>
<tr>
<td>4.9.1 Paging and Memory Typing when the PAT is Not Supported (Pentium Pro and Pentium II Processors)</td>
<td>4-40</td>
</tr>
<tr>
<td>4.9.2 Paging and Memory Typing when the PAT is Supported (Pentium III and More Recent Processor Families)</td>
<td>4-40</td>
</tr>
<tr>
<td>4.9.3 Caching Paging-Related Information about Memory Typing</td>
<td>4-41</td>
</tr>
<tr>
<td>4.10 CACHING TRANSLATION INFORMATION</td>
<td>4-41</td>
</tr>
<tr>
<td>4.10.1 Process-Context Identifiers (PCIDs)</td>
<td>4-41</td>
</tr>
<tr>
<td>4.10.2 Translation Lookaside Buffers (TLBs)</td>
<td>4-42</td>
</tr>
<tr>
<td>4.10.2.1 Page Numbers, Page Frames, and Page Offsets</td>
<td>4-42</td>
</tr>
<tr>
<td>4.10.2.2 Caching Translations in TLBs</td>
<td>4-42</td>
</tr>
<tr>
<td>4.10.2.3 Details of TLB Use</td>
<td>4-43</td>
</tr>
<tr>
<td>4.10.2.4 Global Pages</td>
<td>4-44</td>
</tr>
<tr>
<td>4.10.3 Paging-Structure Caches</td>
<td>4-44</td>
</tr>
<tr>
<td>4.10.3.1 Caches for Paging Structures</td>
<td>4-44</td>
</tr>
<tr>
<td>4.10.3.2 Using the Paging-Structure Caches to Translate Linear Addresses</td>
<td>4-47</td>
</tr>
<tr>
<td>4.10.3.3 Multiple Cached Entries for a Single Paging-Structure Entry</td>
<td>4-47</td>
</tr>
<tr>
<td>4.10.4 Invalidation of TLBs and Paging-Structure Caches</td>
<td>4-48</td>
</tr>
<tr>
<td>4.10.4.1 Operations that Invalidate TLBs and Paging-Structure Caches</td>
<td>4-48</td>
</tr>
<tr>
<td>4.10.4.2 Recommended Invalidation</td>
<td>4-50</td>
</tr>
</tbody>
</table>
### CHAPTER 5
#### PROTECTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Enabling and Disabling Segment and Page Protection</td>
<td>5-1</td>
</tr>
<tr>
<td>5.2</td>
<td>Fields and Flags Used for Segment-Level and Page-Level Protection</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Code-Segment Descriptor in 64-bit Mode</td>
<td>5-3</td>
</tr>
<tr>
<td>5.3</td>
<td>Limit Checking</td>
<td>5-4</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Limit Checking in 64-bit Mode</td>
<td>5-5</td>
</tr>
<tr>
<td>5.4</td>
<td>Type Checking</td>
<td>5-5</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Null Segment Selector Checking</td>
<td>5-6</td>
</tr>
<tr>
<td>5.4.1.1</td>
<td>NULL Segment Checking in 64-bit Mode</td>
<td>5-6</td>
</tr>
<tr>
<td>5.5</td>
<td>Privilege Levels</td>
<td>5-6</td>
</tr>
<tr>
<td>5.6</td>
<td>Privilege Level Checking When Accessing Data Segments</td>
<td>5-8</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Accessing Data in Code Segments</td>
<td>5-9</td>
</tr>
<tr>
<td>5.7</td>
<td>Privilege Level Checking When Loading the SS Register</td>
<td>5-10</td>
</tr>
<tr>
<td>5.8</td>
<td>Privilege Level Checking When Transferring Program Control Between Code Segments</td>
<td>5-10</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Direct Calls or Jumps to Code Segments</td>
<td>5-10</td>
</tr>
<tr>
<td>5.8.1.1</td>
<td>Accessing Nonconforming Code Segments</td>
<td>5-11</td>
</tr>
<tr>
<td>5.8.1.2</td>
<td>Accessing Conforming Code Segments</td>
<td>5-12</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Gate Descriptors</td>
<td>5-13</td>
</tr>
<tr>
<td>5.8.3</td>
<td>Call Gates</td>
<td>5-13</td>
</tr>
<tr>
<td>5.8.3.1</td>
<td>IA-32e Mode Call Gates</td>
<td>5-14</td>
</tr>
<tr>
<td>5.8.4</td>
<td>Accessing a Code Segment Through a Call Gate</td>
<td>5-15</td>
</tr>
<tr>
<td>5.8.5</td>
<td>Stack Switching</td>
<td>5-17</td>
</tr>
<tr>
<td>5.8.5.1</td>
<td>Stack Switching in 64-bit Mode</td>
<td>5-19</td>
</tr>
<tr>
<td>5.8.6</td>
<td>Returning from a Called Procedure</td>
<td>5-20</td>
</tr>
<tr>
<td>5.8.7</td>
<td>Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions</td>
<td>5-20</td>
</tr>
<tr>
<td>5.8.7.1</td>
<td>SYSENTER and SYSEXIT Instructions in IA-32e Mode</td>
<td>5-21</td>
</tr>
<tr>
<td>5.8.8</td>
<td>Fast System Calls in 64-Bit Mode</td>
<td>5-22</td>
</tr>
<tr>
<td>5.9</td>
<td>Privileged Instructions</td>
<td>5-23</td>
</tr>
<tr>
<td>5.10</td>
<td>Pointer Validation</td>
<td>5-24</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Checking Access Rights (LAR Instruction)</td>
<td>5-24</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Checking Read/Write Rights (VERR and VERW Instructions)</td>
<td>5-25</td>
</tr>
<tr>
<td>5.10.3</td>
<td>Checking That the Pointer Offset Is Within Limits (LSL Instruction)</td>
<td>5-25</td>
</tr>
<tr>
<td>5.10.4</td>
<td>Checking Caller Access Privileges (ARPL Instruction)</td>
<td>5-26</td>
</tr>
<tr>
<td>5.10.5</td>
<td>Checking Alignment</td>
<td>5-27</td>
</tr>
<tr>
<td>5.11</td>
<td>Page-Level Protection</td>
<td>5-27</td>
</tr>
<tr>
<td>5.11.1</td>
<td>Page-Protection Flags</td>
<td>5-28</td>
</tr>
<tr>
<td>5.11.2</td>
<td>Restricting Addressable Domain</td>
<td>5-28</td>
</tr>
<tr>
<td>5.11.3</td>
<td>Page Type</td>
<td>5-28</td>
</tr>
<tr>
<td>5.11.4</td>
<td>Combining Protection of Both Levels of Page Tables</td>
<td>5-28</td>
</tr>
<tr>
<td>5.11.5</td>
<td>Overrides to Page Protection</td>
<td>5-28</td>
</tr>
<tr>
<td>5.12</td>
<td>Combining Page and Segment Protection</td>
<td>5-29</td>
</tr>
<tr>
<td>5.13</td>
<td>Page-Level Protection and Execute-Disable Bit</td>
<td>5-30</td>
</tr>
<tr>
<td>5.13.1</td>
<td>Detecting and Enabling the Execute-Disable Capability</td>
<td>5-30</td>
</tr>
<tr>
<td>5.13.2</td>
<td>Execute-Disable Page Protection</td>
<td>5-30</td>
</tr>
<tr>
<td>5.13.3</td>
<td>Reserved Bit Checking</td>
<td>5-31</td>
</tr>
<tr>
<td>5.13.4</td>
<td>Exception Handling</td>
<td>5-32</td>
</tr>
</tbody>
</table>

### CHAPTER 6
#### INTERRUPT AND EXCEPTION HANDLING

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Interrupt and Exception Overview</td>
<td>6-1</td>
</tr>
<tr>
<td>6.2</td>
<td>Exception and Interrupt Vectors</td>
<td>6-1</td>
</tr>
<tr>
<td>6.3</td>
<td>Sources of Interrupts</td>
<td>6-2</td>
</tr>
</tbody>
</table>
6.3.1 External Interrupts .......................................................... 6-2
6.3.2 Maskable Hardware Interrupts ................................... 6-3
6.3.3 Software-Generated Interrupts .................................. 6-4
6.4 SOURCES OF EXCEPTIONS .............................................. 6-4
6.4.1 Program-Error Exceptions ....................................... 6-4
6.4.2 Software-Generated Exceptions ................................. 6-4
6.4.3 Machine-Check Exceptions ....................................... 6-4
6.5 EXCEPTION CLASSIFICATIONS ......................................... 6-5
6.6 PROGRAM OR TASK RESTART .......................................... 6-5
6.7 NONMASKABLE INTERRUPT (NMI) .................................. 6-6
6.7.1 Handling Multiple NMIs ........................................... 6-6
6.8 ENABLING AND DISABLING INTERRUPTS ....................... 6-7
6.8.1 Masking Maskable Hardware Interrupts ...................... 6-7
6.8.2 Masking Instruction Breakpoints .............................. 6-7
6.8.3 Masking Exceptions and Interrupts When Switching Stacks 6-8
6.9 PRIORITIZATION OF CONCURRENT EVENTS ....................... 6-8
6.10 INTERRUPT DESCRIPTOR TABLE (IDT) ............................ 6-9
6.11 IDT DESCRIPTORS .......................................................... 6-10
6.12 EXCEPTION AND INTERRUPT HANDLING ....................... 6-11
6.12.1 Exception- or Interrupt-Handler Procedures ............... 6-12
6.12.1.1 Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines 6-14
6.12.1.2 Protection of Exception- and Interrupt-Handler Procedures 6-16
6.12.1.3 Flag Usage By Exception- or Interrupt-Handler Procedure .................................................. 6-17
6.12.2 Interrupt Tasks .......................................................... 6-17
6.13 ERROR CODE ................................................................. 6-18
6.14 EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE .... 6-19
6.14.1 64-Bit Mode IDT ....................................................... 6-19
6.14.2 64-Bit Mode Stack Frame .......................................... 6-20
6.14.3 IRET in IA-32e Mode ................................................ 6-21
6.14.4 Stack Switching in IA-32e Mode ................................. 6-21
6.14.5 Interrupt Stack Table ............................................... 6-22
6.15 EXCEPTION AND INTERRUPT REFERENCE ....................... 6-23
   Interrupt 0—Divide Error Exception (#DE) ...................... 6-24
   Interrupt 1—Debug Exception (#DB) ............................... 6-25
   Interrupt 2—NMI Interrupt ........................................... 6-27
   Interrupt 3—Breakpoint Exception (#BP) ......................... 6-28
   Interrupt 4—Overflow Exception (#OF) ........................... 6-29
   Interrupt 5—BOUND Range Exceeded Exception (#BR) ........ 6-30
   Interrupt 6—Invalid Opcode Exception (#UD) .................. 6-31
   Interrupt 7—Device Not Available Exception (#NM) .......... 6-32
   Interrupt 8—Double Fault Exception (#DF) ..................... 6-33
   Interrupt 9—Coprocessor Segment Overrun ..................... 6-35
   Interrupt 10—Invalid TSS Exception (#TS) ...................... 6-36
   Interrupt 11—Segment Not Present (#NP) ........................ 6-38
   Interrupt 12—Stack Fault Exception (#SS) ...................... 6-40
   Interrupt 13—General Protection Exception (#GP) ............ 6-41
   Interrupt 14—Page-Fault Exception (#PF) ....................... 6-44
   Interrupt 16—x87 FPU Floating-Point Error (#MF) .......... 6-48
   Interrupt 17—Alignment Check Exception (#AC) ............... 6-50
   Interrupt 18—Machine-Check Exception (#MC) ................. 6-52
   Interrupt 19—SIMD Floating-Point Exception (#XM) ......... 6-53
   Interrupt 20—Virtualization Exception (#VE) .................. 6-55
   Interrupt 21—Control Protection Exception (#CP) ............. 6-56
   Interrupts 32 to 255—User Defined Interrupts ............... 6-58

CHAPTER 7
USER INTERRUPTS
7.1 INTRODUCTION ............................................................ 7-1
7.2 ENUMERATION AND ENABLING ....................................... 7-1
7.3 USER-INTERRUPT STATE AND USER-INTERRUPT MSRS .......... 7-1
7.3.1 User-Interrupt State ............................................... 7-2
CHAPTER 8
TASK MANAGEMENT
8.1 TASK MANAGEMENT OVERVIEW 8-1
8.1.1 Task Structure 8-1
8.1.2 Task State 8-2
8.1.3 Executing a Task 8-2
8.2 TASK MANAGEMENT DATA STRUCTURES 8-3
8.2.1 Task-State Segment (TSS) 8-3
8.2.2 TSS Descriptor 8-5
8.2.3 TSS Descriptor in 64-bit mode 8-6
8.2.4 Task Register 8-7
8.2.5 Task-Gate Descriptor 8-8
8.3 TASK SWITCHING 8-9
8.4 TASK LINKING 8-15
8.4.1 Use of Busy Flag To Prevent Recursive Task Switching 8-16
8.4.2 Modifying Task Linkages 8-16
8.5 TASK ADDRESS SPACE 8-16
8.5.1 Mapping Tasks to the Linear and Physical Address Spaces 8-17
8.5.2 Task Logical Address Space 8-18
8.6 16-BIT TASK-STATE SEGMENT (TSS) 8-18
8.7 TASK MANAGEMENT IN 64-BIT MODE 8-19

CHAPTER 9
MULTIPLE-PROCESSOR MANAGEMENT
9.1 LOCKED ATOMIC OPERATIONS 9-1
9.1.1 Guaranteed Atomic Operations 9-2
9.1.2 Bus Locking 9-3
9.1.2.1 Automatic Locking 9-3
9.1.2.2 Software Controlled Bus Locking 9-4
9.1.2.3 Features to Disable Bus Locks 9-4
9.1.3 Handling Self- and Cross-Modifying Code 9-5
9.1.4 Effects of a LOCK Operation on Internal Processor Caches 9-6
9.2 MEMORY ORDERING 9-6
9.2.1 Memory Ordering in the Intel Pentium® and Intel®64 * Processors 9-6
9.2.2 Memory Ordering in P6 and More Recent Processor Families 9-7
9.2.3 Examples Illustrating the Memory-Ordering Principles 9-8
9.2.3.1 Assumptions, Terminology, and Notation 9-8
9.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations 9-9
9.2.3.3 Stores Are Not Reordered With Earlier Loads 9-9
9.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations 9-10
9.2.3.5 Intra-Processor Forwarding Is Allowed 9-11
9.2.3.6 Stores Are Transitively Visible 9-11
9.2.3.7 Stores Are Seen in a Consistent Order by Other Processors 9-12
9.2.3.8 Locked Instructions Have a Total Order 9-12
9.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions 9-12
9.2.4 Fast-String Operation and Out-of-Order Stores 9-13
9.2.4.1 Memory-Ordering Model for String Operations on Write-Back (WB) Memory 9-13
9.2.4.2 Examples Illustrating Memory-Ordering Principles for String Operations 9-14
9.2.5 Strengthening or Weakening the Memory-Ordering Model 9-16
9.3 SERIALIZING INSTRUCTIONS 9-17
9.4 MULTIPLE-PROCESSOR (MP) INITIALIZATION 9-19
9.4.1 BSP and AP Processors 9-19
CHAPTER 10
PROCESSOR MANAGEMENT AND INITIALIZATION

10.1 INITIALIZE OVERVIEW .................................................................................. 10-1
10.1.1 Processor State After Reset ................................................................. 10-2
10.1.2 Processor Built-In Self-Test (BIST) .................................................... 10-5
10.1.3 Model and Stepping Information ......................................................... 10-5
10.1.4 First Instruction Executed ................................................................. 10-5
10.2 X87 FPU INITIALIZATION ......................................................................... 10-5
10.2.1 Configuring the x87 FPU Environment ............................................. 10-6
10.2.2 Setting the Processor for x87 FPU Software Emulation .................... 10-6
10.3 CACHE ENABLING .................................................................................. 10-7
10.4 MODEL-SPECIFIC REGISTERS (MSRS) .................................................. 10-7
10.5 MEMORY TYPE RANGE REGISTERS (MTRRS) ....................................... 10-8
10.6 INITIALIZING SSE/SSE2/SSSE3/SSSE3 EXTENSIONS .............................. 10-8
10.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION . 10-8
10.7.1 Real-Address Mode IDT ................................................................. 10-8
10.7.2 NMI Interrupt Handling ................................................................. 10-9
10.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION .. 10-9
10.8.1 Protected-Mode System Data Structures ........................................... 10-9
10.8.2 Initializing Protected-Mode Exceptions and Interrupts ..................... 10-10
10.8.3 Initializing Paging .......................................................................... 10-10
10.8.4 Initializing Multitasking ................................................................. 10-10
10.8.5 Initializing IA-32e Mode System Data Structures ............................ 10-11
10.8.5.1 IA-32e Mode System Data Structures ......................................... 10-11
10.8.5.2 IA-32e Mode Interrupts and Exceptions ...................................... 10-12
10.8.5.3 64-bit Mode and Compatibility Mode Operation ........................... 10-12
10.8.5.4 Switching Out of IA-32e Mode Operation .................................... 10-12
10.9 MODE SWITCHING .............................................................................. 10-13
10.9.1 Switching to Protected Mode ......................................................... 10-13
10.9.2 Switching Back to Real-Address Mode ............................................ 10-14
10.10 INITIALIZATION AND MODE SWITCHING EXAMPLE ....................... 10-14
10.10.1 Assembler Usage ......................................................................... 10-16
10.10.2 STARTUP.ASM Listing ............................................................... 10-16
10.10.3 MAIN.ASM Source Code ............................................................. 10-25
10.10.4 Supporting Files ......................................................................... 10-25
10.11 MICROCODE UPDATE FACILITIES ..................................................... 10-27
10.11.1 Microcode Update ....................................................................... 10-28
10.11.2 Optional Extended Signature Table .............................................. 10-31
10.11.3 Processor Identification ............................................................... 10-32
10.11.4 Platform Identification ............................................................... 10-32
10.11.5 Microcode Update Checksum ....................................................... 10-33
10.11.6 Microcode Update Loader ........................................................... 10-34
10.11.6.1 Hard Resets in Update Loading ............................................... 10-35
10.11.6.2 Update in a Multiprocessor System .......................................... 10-35
10.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology .10-35
10.11.6.4 Update in a System Supporting Dual-Core Technology ............. 10-35
10.11.6.5 Update Loader Enhancements .................................................. 10-35
10.11.7 Update Signature and Verification .................................................. 10-36
10.11.7.1 Determining the Signature ..................................................... 10-36
10.11.7.2 Authenticating the Update .................................................... 10-37
10.11.8 Optional Processor Microcode Update Specifications ...................... 10-37
10.11.8.1 Responsibilities of the BIOS ..................................................... 10-38
10.11.8.2 Responsibilities of the Calling Program ..................................... 10-39
10.11.8.3 Microcode Update Functions ................................................... 10-42
10.11.8.4 INT 15H-based Interface ........................................................ 10-42
10.11.8.5 Function 00H—Presence Test ................................................... 10-42
10.11.8.6 Function 01H—Write Microcode Update Data ............................. 10-43
10.11.8.7 Function 02H—Microcode Update Control .................................. 10-46
10.11.8.8 Function 03H—Read Microcode Update Data ............................ 10-47
10.11.8.9 Return Codes .................................................................. 10-48
CONTENTS

16.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE ................................................. 16-28
  16.10.1 Machine-Check Exception Handler ................................................................. 16-28
  16.10.2 Pentium Processor Machine-Check Exception Handling ................................. 16-29
  16.10.3 Logging Correctable Machine-Check Errors ............................................... 16-29
  16.10.4 Machine-Check Software Handler Guidelines for Error Recovery ...................... 16-31
  16.10.4.1 Machine-Check Exception Handler for Correctable Machine-Check Errors ...... 16-31
  16.10.4.2 Corrected Machine-Check Handler for Error Recovery ............................ 16-35

CHAPTER 17
INTERPRETING MACHINE CHECK ERROR CODES
  17.1 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H, MACHINE ERROR CODES FOR MACHINE CHECK ... 17-1
  17.2 INCREMENTAL DECODING INFORMATION: INTEL® CORE™ 2 PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK ................................................................. 17-3
    17.2.1 Model-Specific Machine Check Error Codes for Intel® Xeon® Processor 7400 Series ......................................................... 17-5
    17.2.1.1 Processor Machine Check Status Register, Incremental MCA Error Code Definition ......................................................... 17-6
    17.2.2 Intel® Xeon® Processor 7400 Model Specific Error Code Field .............................. 17-6
        17.2.2.1 Processor Model Specific Error Code Field, Type B: Bus and Interconnect Error Codes ......................................................... 17-6
        17.2.2.2 Processor Model Specific Error Code Field, Type C: Cache Bus Controller Error Codes ......................................................... 17-7
    17.3 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR 3400, 3500, 5500 SERIES, MACHINE ERROR CODES FOR MACHINE CHECK ......................................................... 17-7
        17.3.1 Intel® Xeon® Machine Check Errors ................................................................. 17-8
        17.3.2 Internal Machine Check Errors ..................................................................... 17-9
<table>
<thead>
<tr>
<th>Section Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.3.3</td>
<td>Memory Controller Errors</td>
<td>17-9</td>
</tr>
<tr>
<td>17.4</td>
<td>INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-10</td>
</tr>
<tr>
<td>17.4.1</td>
<td>Internal Machine Check Errors</td>
<td>17-10</td>
</tr>
<tr>
<td>17.4.2</td>
<td>Intel® QPI Machine Check Errors</td>
<td>17-11</td>
</tr>
<tr>
<td>17.4.3</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-11</td>
</tr>
<tr>
<td>17.5</td>
<td>INCENTRAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V2 AND INTEL® XEON® PROCESSOR E7 V2 FAMILIES, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-13</td>
</tr>
<tr>
<td>17.5.1</td>
<td>Internal Machine Check Errors</td>
<td>17-13</td>
</tr>
<tr>
<td>17.5.2</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-14</td>
</tr>
<tr>
<td>17.5.3</td>
<td>Home Agent Machine Check Errors</td>
<td>17-15</td>
</tr>
<tr>
<td>17.6</td>
<td>INCENTRAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V3 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-15</td>
</tr>
<tr>
<td>17.6.1</td>
<td>Internal Machine Check Errors</td>
<td>17-16</td>
</tr>
<tr>
<td>17.6.2</td>
<td>Intel® QPI Machine Check Errors</td>
<td>17-17</td>
</tr>
<tr>
<td>17.6.3</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-17</td>
</tr>
<tr>
<td>17.6.4</td>
<td>Home Agent Machine Check Errors</td>
<td>17-19</td>
</tr>
<tr>
<td>17.7</td>
<td>INCENTRAL DECODING INFORMATION: INTEL® XEON® PROCESSOR D FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-19</td>
</tr>
<tr>
<td>17.7.1</td>
<td>Internal Machine Check Errors</td>
<td>17-19</td>
</tr>
<tr>
<td>17.7.2</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-20</td>
</tr>
<tr>
<td>17.8</td>
<td>INCENTRAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V4 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-21</td>
</tr>
<tr>
<td>17.8.1</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-21</td>
</tr>
<tr>
<td>17.8.2</td>
<td>Home Agent Machine Check Errors</td>
<td>17-21</td>
</tr>
<tr>
<td>17.9</td>
<td>INCENTRAL DECODING INFORMATION: INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-22</td>
</tr>
<tr>
<td>17.9.1</td>
<td>Internal Machine Check Errors</td>
<td>17-22</td>
</tr>
<tr>
<td>17.9.2</td>
<td>Interconnect Machine Check Errors</td>
<td>17-24</td>
</tr>
<tr>
<td>17.9.3</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-25</td>
</tr>
<tr>
<td>17.9.4</td>
<td>M2M Machine Check Errors</td>
<td>17-26</td>
</tr>
<tr>
<td>17.9.5</td>
<td>Home Agent Machine Check Errors</td>
<td>17-27</td>
</tr>
<tr>
<td>17.10</td>
<td>INCENTRAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUD DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_5FH, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-28</td>
</tr>
<tr>
<td>17.10.1</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-28</td>
</tr>
<tr>
<td>17.11</td>
<td>INCENTRAL DECODING INFORMATION: 3RD GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-28</td>
</tr>
<tr>
<td>17.11.1</td>
<td>Internal Machine Check Errors</td>
<td>17-29</td>
</tr>
<tr>
<td>17.11.2</td>
<td>Interconnect Machine Check Errors</td>
<td>17-31</td>
</tr>
<tr>
<td>17.11.3</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-32</td>
</tr>
<tr>
<td>17.11.4</td>
<td>M2M Machine Check Errors</td>
<td>17-36</td>
</tr>
<tr>
<td>17.12</td>
<td>INCENTRAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUD DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_86H, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-36</td>
</tr>
<tr>
<td>17.12.1</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-36</td>
</tr>
<tr>
<td>17.12.2</td>
<td>M2M Machine Check Errors</td>
<td>17-36</td>
</tr>
<tr>
<td>17.13</td>
<td>INCENTRAL DECODING INFORMATION: 4TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-37</td>
</tr>
<tr>
<td>17.13.1</td>
<td>Internal Machine Check Errors</td>
<td>17-37</td>
</tr>
<tr>
<td>17.13.2</td>
<td>Interconnect Machine Check Errors</td>
<td>17-39</td>
</tr>
<tr>
<td>17.13.3</td>
<td>Integrated Memory Controller Machine Check Errors</td>
<td>17-41</td>
</tr>
<tr>
<td>17.13.4</td>
<td>M2M Machine Check Errors</td>
<td>17-43</td>
</tr>
<tr>
<td></td>
<td>High Bandwidth Memory Machine Check Errors</td>
<td>17-44</td>
</tr>
<tr>
<td>17.14</td>
<td>INCENTRAL DECODING INFORMATION: PROCESSOR FAMILY OFH, MACHINE ERROR CODES FOR MACHINE CHECK</td>
<td>17-44</td>
</tr>
<tr>
<td>17.14.1</td>
<td>Model-Specific Machine Check Error Codes for the Intel® Xeon® Processor MP 7100 Series</td>
<td>17-45</td>
</tr>
<tr>
<td>17.14.1.1</td>
<td>Processor Machine Check Status Register MCA Error Code Definition</td>
<td>17-45</td>
</tr>
<tr>
<td>17.14.2</td>
<td>Other_Info Field (All MCA Error Types)</td>
<td>17-47</td>
</tr>
<tr>
<td>17.14.3</td>
<td>Processor Model Specific Error Code Field</td>
<td>17-48</td>
</tr>
<tr>
<td>17.14.3.1</td>
<td>MCA Error Type A: L3 Error</td>
<td>17-48</td>
</tr>
<tr>
<td>17.14.3.2</td>
<td>Processor Model Specific Error Code Field Type B: Bus and Interconnect Error</td>
<td>17-48</td>
</tr>
<tr>
<td>17.14.3.3</td>
<td>Processor Model Specific Error Code Field Type C: Cache Bus Controller Error</td>
<td>17-49</td>
</tr>
</tbody>
</table>
## FEATURES

### 18.1 OVERVIEW OF DEBUG SUPPORT FACILITIES ......................................................... 18-1

### 18.2 DEBUG REGISTERS .............................................................................................. 18-2

#### 18.2.1 Debug Address Registers (DR0-DR3) .............................................................. 18-4

#### 18.2.2 Debug Registers DR4 and DR5 ........................................................................ 18-4

#### 18.2.3 Debug Status Register (DR6) .......................................................................... 18-4

#### 18.2.4 Debug Control Register (DR7) ........................................................................ 18-4

#### 18.2.5 Breakpoint Field Recognition ........................................................................... 18-6

#### 18.2.6 Debug Registers and Intel® 64 Processors ......................................................... 18-7

### 18.3 DEBUG EXCEPTIONS ............................................................................................ 18-7

#### 18.3.1 Debug Exception (#DB)—Interrupt Vector 1 ................................................... 18-7

#### 18.3.1.1 Instruction-Breakpoint Exception Condition .................................................. 18-9

#### 18.3.1.2 Data Memory and I/O Breakpoint Exception Conditions .............................. 18-10

#### 18.3.1.3 General-Detect Exception Condition ............................................................... 18-10

#### 18.3.1.4 Single-Step Exception Condition ................................................................. 18-11

#### 18.3.1.5 Task-Switch Exception Condition ................................................................. 18-11

#### 18.3.1.6 OS Bus-Lock Detection ................................................................................. 18-11

#### 18.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3 ........................................... 18-11

#### 18.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM) ........................................................................................................ 18-12

### 18.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW .......... 18-12

#### 18.4.1 IA32_DEBUGCTL MSR .................................................................................. 18-12

#### 18.4.2 Monitoring Branches, Exceptions, and Interrupts .......................................... 18-14

#### 18.4.3 Single-Stepping on Branches .......................................................................... 18-14

#### 18.4.4 Branch Trace Messages ................................................................................... 18-15

#### 18.4.5 Branch Trace Store (BTS) ................................................................................. 18-15

#### 18.4.6 CPL-Qualified Branch Trace Mechanism ............................................................. 18-15

#### 18.4.7 Freezing LBR and Performance Counters on PMI ............................................ 18-15

#### 18.4.8 Branch Trace Store (BTS) ................................................................................. 18-15

#### 18.4.8.1 LBR Stack and Intel® 64 Processors ................................................................. 18-18

#### 18.4.8.2 LBR Stack and IA-32 Processors ................................................................. 18-18

#### 18.4.8.3 Last Exception Records and Intel 64 Architecture ......................................... 18-18

#### 18.4.8.4 Last Exception Records and Intel® 64 Architecture ........................................ 18-18

#### 18.4.8.5 Branch Trace Message Visibility ..................................................................... 18-18

#### 18.4.8.6 Branch Trace Store (BTS) ................................................................................. 18-18

#### 18.4.8.7 Freezing LBR and Performance Counters on PMI ............................................ 18-18

#### 18.4.8.8 Freezing LBR and Performance Counters on PMI ........................................... 18-18

#### 18.4.8.9 Freezing LBR and Performance Counters on PMI ........................................... 18-18

#### 18.4.8.9.1 64 Bit Format of the DS Save Area ............................................................... 18-22

#### 18.4.8.9.2 Setting Up the DS Save Area ......................................................................... 18-24

#### 18.4.8.9.3 Setting Up the BTS Buffer ........................................................................... 18-25

#### 18.4.8.9.4 Setting Up CPL-Qualified BTS ..................................................................... 18-26

#### 18.4.8.9.5 Writing the DS Interrupt Service Routine ..................................................... 18-26

### 18.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL ATOM® PROCESSORS) ...................... 18-27

#### 18.5.1 LBR Stack ...................................................................................................... 18-27

#### 18.5.2 LBR in Intel Atom® Processors ........................................................................ 18-28

#### 18.5.3 Last Exception Records and Intel 64 Architecture ......................................... 18-28

### 18.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Goldmont Microarchitecture .......... 18-28

### 18.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Goldmont Plus Microarchitecture .................................................. 18-29

### 18.8 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR Intel® Xeon Phi™ Processor 7200/5200/3200 .......................... 18-29

### 18.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Nehalem Microarchitecture ........................................... 18-29

#### 18.9.1 LBR Stack ...................................................................................................... 18-30

#### 18.9.2 Filtering of Last Branch Records ...................................................................... 18-31

### 18.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Sandy Bridge Microarchitecture ...................................................... 18-31

### 18.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Haswell Microarchitecture ...................................................... 18-32

#### 18.11.1 LBR Stack Enhancement .................................................................................. 18-33

#### 18.11.2 Streamlined Freeze_LBRs_On_PMI Operation .................................................. 18-34

#### 18.11.3 LBR Behavior and Deep C-State .................................................................... 18-35

### 18.12 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Skylake Microarchitecture ........................................ 18-35

### 18.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE) ........................................ 18-35
## Chapter 20
### Performance Monitoring

#### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 PERFORMANCE MONITORING OVERVIEW</td>
<td>20-1</td>
</tr>
<tr>
<td>20.2 ARCHITECTURAL PERFORMANCE MONITORING</td>
<td>20-2</td>
</tr>
<tr>
<td>20.2.1 Architectural Performance Monitoring Version 1 Facilities</td>
<td>20-3</td>
</tr>
<tr>
<td>20.2.1.1 Pre-defined Architectural Performance Events</td>
<td>20-3</td>
</tr>
<tr>
<td>20.2.2 Architectural Performance Monitoring Version 2</td>
<td>20-5</td>
</tr>
<tr>
<td>20.2.3 Architectural Performance Monitoring Version 3</td>
<td>20-7</td>
</tr>
<tr>
<td>20.2.3.1 AnyThread Counting and Software Evolution</td>
<td>20-9</td>
</tr>
<tr>
<td>20.2.4 Architectural Performance Monitoring Version 4</td>
<td>20-10</td>
</tr>
<tr>
<td>20.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS</td>
<td>20-11</td>
</tr>
<tr>
<td>20.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRs</td>
<td>20-12</td>
</tr>
<tr>
<td>20.2.4.3 IA32_PERF_GLOBAL_INUSE MSR</td>
<td>20-12</td>
</tr>
<tr>
<td>20.2.5 Architecural Performance Monitoring Version 5</td>
<td>20-13</td>
</tr>
<tr>
<td>20.2.5.1 AnyThread Mode Deprecation</td>
<td>20-13</td>
</tr>
<tr>
<td>20.2.5.2 Fixed Counter Enumeration</td>
<td>20-13</td>
</tr>
<tr>
<td>20.2.5.3 Domain Separation</td>
<td>20-13</td>
</tr>
<tr>
<td>20.2.6 Full-Width Writes to Performance Counter Registers</td>
<td>20-14</td>
</tr>
<tr>
<td>20.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS)</td>
<td>20-15</td>
</tr>
<tr>
<td>20.3.1 Performance Monitoring for Processors Based on Nehalem Microarchitecture</td>
<td>20-16</td>
</tr>
<tr>
<td>20.3.1.1 Enhancements of Performance Monitoring in the Processor Core</td>
<td>20-18</td>
</tr>
<tr>
<td>20.3.1.2 Performance Monitoring Facility in the Uncore</td>
<td>20-20</td>
</tr>
<tr>
<td>20.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility</td>
<td>20-21</td>
</tr>
<tr>
<td>20.3.2 Performance Monitoring for Processors Based on Westmere Microarchitecture</td>
<td>20-22</td>
</tr>
<tr>
<td>20.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility</td>
<td>20-23</td>
</tr>
<tr>
<td>20.3.4 Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture</td>
<td>20-24</td>
</tr>
<tr>
<td>20.3.4.1 Global Counter Control Facilities in Sandy Bridge Microarchitecture</td>
<td>20-25</td>
</tr>
<tr>
<td>20.3.4.2 Counter Coalescence</td>
<td>20-26</td>
</tr>
<tr>
<td>20.3.4.3 Full Width Writes to Performance Counters</td>
<td>20-27</td>
</tr>
<tr>
<td>20.3.4.4 PEBS Support in Sandy Bridge Microarchitecture</td>
<td>20-28</td>
</tr>
<tr>
<td>20.3.4.5 Off-core Response Performance Monitoring</td>
<td>20-29</td>
</tr>
<tr>
<td>20.3.4.6 Uncore Performance Monitoring Facilities in the Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, and Intel® Core™ i3-2xxx Processor Series</td>
<td>20-30</td>
</tr>
<tr>
<td>20.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility</td>
<td>20-31</td>
</tr>
<tr>
<td>20.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility</td>
<td>20-32</td>
</tr>
</tbody>
</table>
CHAPTER 21
8086 EMULATION

21.1 REAL-ADDRESS MODE .............................................. 21-1
21.1.1 Address Translation in Real-Address Mode ............... 21-2
21.1.2 Registers Supported in Real-Address Mode .......... 21-3
21.1.3 Instructions Supported in Real-Address Mode .... 21-3
21.1.4 Interrupt and Exception Handling .................... 21-4
21.2 VIRTUAL-8086 MODE ............................................ 21-5
21.2.1 Enabling Virtual-8086 Mode .......................... 21-6
21.2.2 Structure of a Virtual-8086 Task ...................... 21-6
21.2.3 Paging of Virtual-8086 Tasks ......................... 21-7
21.2.4 Protection within a Virtual-8086 Task ............ 21-8
21.2.5 Entering Virtual-8086 Mode ......................... 21-8
21.2.6 Leaving Virtual-8086 Mode .......................... 21-9
21.2.7 Sensitive Instructions ................................. 21-10
21.2.8 Virtual-8086 Mode I/O .............................. 21-10

20.6.4.2 CCCR MSR .................................................................. 20-121
20.6.4.3 IA32_PEBs_ENABLE MSR ........................................ 20-123
20.6.4.4 Performance Monitoring Events ....................... 20-123
20.6.4.5 Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture .................................................. 20-124
20.6.5 Performance Monitoring and Dual-Core Technology ......... 20-125
20.6.6 Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache ............................. 20-125
20.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems ........................................ 20-127
20.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller ........................................ 20-128
20.6.7.2 GBSQ Event Interface ............................................. 20-129
20.6.7.3 GSNPQ Event Interface ......................................... 20-130
20.6.7.4 FSB Event Interface ............................................. 20-131
20.6.7.5 Common Event Control Interface ......................... 20-133
20.6.7.6 Starting and Stopping the Performance-Monitoring Counters .................................. 20-135
20.6.7.7 PerfCtr0 and PerfCtr1 MSRs ..................................... 20-135
20.6.7.8 Event and Time-Stamp Monitoring Software .................. 20-135
20.6.7.9 Monitoring Counter Overflow .............................. 20-136
20.6.8 Performance Monitoring (P6 Family Processor) ............. 20-136
20.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs ...................... 20-138
20.6.8.2 PerfCtro and PerfCrt 1 MSRs ...................................... 20-138
20.6.8.3 Starting and Stopping the Performance-Monitoring Counters .................................. 20-139
20.6.8.4 Event and Time-Stamp Monitoring Software .................. 20-139
20.6.8.5 Monitoring Counter Overflow .............................. 20-139
20.7 COUNTING CLOCKS .................................................. 20-139
20.7.1 Non-Halted Reference Clocks ................................. 20-139
20.7.2 Cycle Counting and Opportunistic Processor Operation .................. 20-140
20.7.3 Determining the Processor Base Frequency .................. 20-140
20.7.3.1 For Intel® Processors Based on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell Microarchitectures 20-140
20.7.3.2 For Intel® Processors Based on Nehalem Microarchitecture .................................. 20-140
20.7.3.3 For Intel Atom® Processors Based on Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture) .................. 20-140
20.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture 20-140
20.8 IA32_PERF_CAPABILITIES MSR ENUMERATION ............. 20-141
20.8.1 Filtering of SMM Handler Overhead ......................... 20-142
20.9 PEBS FACILITY ..................................................... 20-142
20.9.1 Extended PEBS .................................................. 20-142
20.9.2 Adaptive PEBS .................................................. 20-144
20.9.2.1 Adaptive_Record Counter Control ....................... 20-145
20.9.2.2 PEBS Record Format .......................................... 20-146
20.9.2.3 MSR_PEBS_DATA_CFG ...................................... 20-150
20.9.2.4 PEBS Record Examples ........................................ 20-151
20.9.3 Precise Distribution of Instructions Retired (PDIR) Facility .................................................. 20-153
20.9.4 Reduced Skid PEBS ............................................... 20-153
20.9.5 EPT-Friendly PEBS ............................................... 20-154
20.9.6 PDist: Precise Distribution ..................................... 20-154
20.9.7 Load Latency Facility .......................................... 20-154
20.9.8 Store Latency Facility .......................................... 20-155
## CHAPTER 22
### MIXING 16-BIT AND 32-BIT CODE

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1</td>
<td>DEFINING 16-BIT AND 32-BIT PROGRAM MODULES</td>
<td>22-1</td>
</tr>
<tr>
<td>22.2</td>
<td>MIXING 16-BIT AND 32-BIT OPERATIONS WITHIN A CODE SEGMENT</td>
<td>22-2</td>
</tr>
<tr>
<td>22.3</td>
<td>SHARING DATA AMONG MIXED-SIZE CODE SEGMENTS</td>
<td>22-3</td>
</tr>
<tr>
<td>22.4</td>
<td>TRANSFERRING CONTROL AMONG MIXED-SIZE CODE SEGMENTS</td>
<td>22-3</td>
</tr>
<tr>
<td>22.4.1</td>
<td>Code-Segment Pointer Size</td>
<td>22-4</td>
</tr>
<tr>
<td>22.4.2</td>
<td>Stack Management for Control Transfer</td>
<td>22-4</td>
</tr>
<tr>
<td>22.4.2.1</td>
<td>Controlling theOperand-Size Attribute For a Call</td>
<td>22-5</td>
</tr>
<tr>
<td>22.4.2.2</td>
<td>Passing Parameters With a Gate</td>
<td>22-6</td>
</tr>
<tr>
<td>22.4.3</td>
<td>Interrupt Control Transfers</td>
<td>22-6</td>
</tr>
<tr>
<td>22.4.4</td>
<td>Parameter Translation</td>
<td>22-6</td>
</tr>
<tr>
<td>22.4.5</td>
<td>Writing Interface Procedures</td>
<td>22-6</td>
</tr>
</tbody>
</table>

## CHAPTER 23
### ARCHITECTURE COMPATIBILITY

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.1</td>
<td>PROCESSOR FAMILIES AND CATEGORIES</td>
<td>23-1</td>
</tr>
<tr>
<td>23.2</td>
<td>RESERVED BITS</td>
<td>23-2</td>
</tr>
<tr>
<td>23.3</td>
<td>ENABLING NEW FUNCTIONS AND MODES</td>
<td>23-2</td>
</tr>
<tr>
<td>23.4</td>
<td>DETECTING THE PRESENCE OF NEW FEATURES THROUGH SOFTWARE</td>
<td>23-2</td>
</tr>
<tr>
<td>23.5</td>
<td>INTEL MMX TECHNOLOGY</td>
<td>23-2</td>
</tr>
<tr>
<td>23.6</td>
<td>STREAMING SIMD EXTENSIONS (SSE)</td>
<td>23-3</td>
</tr>
<tr>
<td>23.7</td>
<td>STREAMING SIMD EXTENSIONS 2 (SSE2)</td>
<td>23-3</td>
</tr>
<tr>
<td>23.8</td>
<td>STREAMING SIMD EXTENSIONS 3 (SSE3)</td>
<td>23-3</td>
</tr>
<tr>
<td>23.9</td>
<td>ADDITIONAL STREAMING SIMD EXTENSIONS</td>
<td>23-3</td>
</tr>
<tr>
<td>23.10</td>
<td>INTEL HYPER-THREADING TECHNOLOGY</td>
<td>23-3</td>
</tr>
<tr>
<td>23.11</td>
<td>MULTI-CORE TECHNOLOGY</td>
<td>23-4</td>
</tr>
<tr>
<td>23.12</td>
<td>SPECIFIC FEATURES OF DUAL-CORE PROCESSORS</td>
<td>23-4</td>
</tr>
<tr>
<td>23.13</td>
<td>NEW INSTRUCTIONS IN THE PENTIUM AND LATER IA-32 PROCESSORS</td>
<td>23-4</td>
</tr>
<tr>
<td>23.13.1</td>
<td>Instructions Added Prior to the Pentium Processor</td>
<td>23-4</td>
</tr>
<tr>
<td>23.14</td>
<td>OBSOLETE INSTRUCTIONS</td>
<td>23-5</td>
</tr>
<tr>
<td>23.15</td>
<td>UNDEFINED OPCODES</td>
<td>23-5</td>
</tr>
<tr>
<td>23.16</td>
<td>NEW FLAGS IN THE EFLAGS REGISTER</td>
<td>23-6</td>
</tr>
<tr>
<td>23.16.1</td>
<td>Using EFLAGS Flags to Distinguish Between 32-Bit IA-32 Processors</td>
<td>23-6</td>
</tr>
<tr>
<td>23.17</td>
<td>STACK OPERATIONS AND USER SOFTWARE</td>
<td>23-7</td>
</tr>
<tr>
<td>23.17.1</td>
<td>PUSH SP</td>
<td>23-7</td>
</tr>
<tr>
<td>23.17.2</td>
<td>EFLAGS Pushed on the Stack</td>
<td>23-7</td>
</tr>
<tr>
<td>23.18</td>
<td>X87 FPU</td>
<td>23-7</td>
</tr>
<tr>
<td>23.18.1</td>
<td>Control Register CR0 Flags</td>
<td>23-8</td>
</tr>
<tr>
<td>23.18.2</td>
<td>x87 FPU Status Word</td>
<td>23-8</td>
</tr>
<tr>
<td>23.18.2.1</td>
<td>Condition Code Flags (C0 through C3)</td>
<td>23-8</td>
</tr>
<tr>
<td>23.18.2.2</td>
<td>Stack Fault Flag</td>
<td>23-8</td>
</tr>
<tr>
<td>23.18.3</td>
<td>x87 FPU Control Word</td>
<td>23-9</td>
</tr>
<tr>
<td>23.18.4</td>
<td>x87 FPU Tag Word</td>
<td>23-9</td>
</tr>
<tr>
<td>23.18.5</td>
<td>Data Types</td>
<td>23-9</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>23.18.5.1 NaNs</td>
<td>23-9</td>
<td></td>
</tr>
<tr>
<td>23.18.5.2 Pseudo-zero, Pseudo-NaN, Pseudo-infinity, and Unnormal Formats</td>
<td>23-9</td>
<td></td>
</tr>
<tr>
<td>23.18.6 Floating-Point Exceptions</td>
<td>23-10</td>
<td></td>
</tr>
<tr>
<td>23.18.6.1 Denormal Operand Exception (#D)</td>
<td>23-10</td>
<td></td>
</tr>
<tr>
<td>23.18.6.2 Numeric Overflow Exception (#O)</td>
<td>23-10</td>
<td></td>
</tr>
<tr>
<td>23.18.6.3 Numeric Underflow Exception (#U)</td>
<td>23-10</td>
<td></td>
</tr>
<tr>
<td>23.18.6.4 Exception Precedence</td>
<td>23-10</td>
<td></td>
</tr>
<tr>
<td>23.18.6.5 CS and EIP For FPU Exceptions</td>
<td>23-11</td>
<td></td>
</tr>
<tr>
<td>23.18.6.6 FPU Error Signals</td>
<td>23-11</td>
<td></td>
</tr>
<tr>
<td>23.18.6.7 Assertion of the FERR# Pin</td>
<td>23-11</td>
<td></td>
</tr>
<tr>
<td>23.18.6.8 Invalid Operation Exception On Denormal</td>
<td>23-11</td>
<td></td>
</tr>
<tr>
<td>23.18.6.9 Alignment Check Exceptions (#AC)</td>
<td>23-11</td>
<td></td>
</tr>
<tr>
<td>23.18.6.10 Segment Not Present Exception During FLDENV</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.6.11 Device Not Available Exception (#NM)</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.6.12 Coprocessor Segment Overrun Exception</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.6.13 General Protection Exception (#GP)</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.6.14 Floating-Point Error Exception (#MF)</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.7 Changes to Floating-Point Instructions</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.7.1 FDIV, FPREM, and FSQRT Instructions</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.7.2 FSQRT Instruction</td>
<td>23-12</td>
<td></td>
</tr>
<tr>
<td>23.18.7.3 FPREM1 Instruction</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.4 FPREM Instruction</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.5 FUCOM, FUCOMP, and FUCOMPP Instructions</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.6 FPTAN Instruction</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.7 Stack Overflow</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.8 FSIN, FCOS, and FSINCOS Instructions</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.9 FPATAN Instruction</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.10 FZXM1 Instruction</td>
<td>23-13</td>
<td></td>
</tr>
<tr>
<td>23.18.7.11 FLD Instruction</td>
<td>23-14</td>
<td></td>
</tr>
<tr>
<td>23.18.7.12 FXTRACT Instruction</td>
<td>23-14</td>
<td></td>
</tr>
<tr>
<td>23.18.7.13 Load Constant Instructions</td>
<td>23-14</td>
<td></td>
</tr>
<tr>
<td>23.18.7.14 FXAM Instruction</td>
<td>23-14</td>
<td></td>
</tr>
<tr>
<td>23.18.7.15 FSAVE and FSTENV Instructions</td>
<td>23-14</td>
<td></td>
</tr>
<tr>
<td>23.18.8 Transcendental Instructions</td>
<td>23-14</td>
<td></td>
</tr>
<tr>
<td>23.18.9 Obsolete Instructions and Undefined Opcodes</td>
<td>23-15</td>
<td></td>
</tr>
<tr>
<td>23.18.10 WAIT/FWAIT Prefix Differences</td>
<td>23-15</td>
<td></td>
</tr>
<tr>
<td>23.18.11 Operands Split Across Segments and/or Pages</td>
<td>23-15</td>
<td></td>
</tr>
<tr>
<td>23.18.12 FPU Instruction Synchronization</td>
<td>23-15</td>
<td></td>
</tr>
<tr>
<td>23.19 SERIALIZING INSTRUCTIONS</td>
<td>23-16</td>
<td></td>
</tr>
<tr>
<td>23.20 FPU AND MATH COPROCESSOR INITIALIZATION</td>
<td>23-16</td>
<td></td>
</tr>
<tr>
<td>23.20.1 Intel® 387 and Intel® 287 Math Coprocessor Initialization</td>
<td>23-16</td>
<td></td>
</tr>
<tr>
<td>23.20.2 Intel®86 SX Processor and Intel® 487 SX Math Coprocessor Initialization</td>
<td>23-16</td>
<td></td>
</tr>
<tr>
<td>23.21 CONTROL REGISTERS</td>
<td>23-17</td>
<td></td>
</tr>
<tr>
<td>23.22 MEMORY MANAGEMENT FACILITIES</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.1 New Memory Management Control Flags</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.1.1 Physical Memory Addressing Extension</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.1.2 Global Pages</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.1.3 Larger Page Sizes</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.2 CD and NW Cache Control Flags</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.3 Descriptor Types and Contents</td>
<td>23-19</td>
<td></td>
</tr>
<tr>
<td>23.22.4 Changes in Segment Descriptor Loads</td>
<td>23-20</td>
<td></td>
</tr>
<tr>
<td>23.23 DEBUG FACILITIES</td>
<td>23-20</td>
<td></td>
</tr>
<tr>
<td>23.23.1 Differences in Debug Register DR6</td>
<td>23-20</td>
<td></td>
</tr>
<tr>
<td>23.23.2 Differences in Debug Register DR7</td>
<td>23-20</td>
<td></td>
</tr>
<tr>
<td>23.23.3 Debug Registers DR4 and DR5</td>
<td>23-20</td>
<td></td>
</tr>
<tr>
<td>23.24 RECOGNITION OF BREAKPOINTS</td>
<td>23-20</td>
<td></td>
</tr>
<tr>
<td>23.25 EXCEPTIONS AND/OR EXCEPTION CONDITIONS</td>
<td>23-21</td>
<td></td>
</tr>
<tr>
<td>23.25.1 Machine-Check Architecture</td>
<td>23-22</td>
<td></td>
</tr>
<tr>
<td>23.25.2 Priority of Exceptions</td>
<td>23-22</td>
<td></td>
</tr>
<tr>
<td>23.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers</td>
<td>23-22</td>
<td></td>
</tr>
<tr>
<td>23.26 INTERRUPTS</td>
<td>23-27</td>
<td></td>
</tr>
<tr>
<td>23.26.1 Interrupt Propagation Delay</td>
<td>23-27</td>
<td></td>
</tr>
<tr>
<td>23.26.2 NMI Interrupts</td>
<td>23-27</td>
<td></td>
</tr>
<tr>
<td>23.26.3 IDT Limit</td>
<td>23-27</td>
<td></td>
</tr>
<tr>
<td>23.27 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)</td>
<td>23-27</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>23.27.1 Software Visible Differences Between the Local APIC and the 82489DX</td>
<td>23-28</td>
<td></td>
</tr>
<tr>
<td>23.27.2 New Features Incorporated in the Local APIC for the P6 Family and Pentium Processors</td>
<td>23-28</td>
<td></td>
</tr>
<tr>
<td>23.27.3 New Features Incorporated in the Local APIC of the Pentium 4 and Intel Xeon Processors</td>
<td>23-28</td>
<td></td>
</tr>
<tr>
<td>23.28 TASK SWITCING AND TSS.</td>
<td>23-28</td>
<td></td>
</tr>
<tr>
<td>23.28.1 P6 Family and Pentium Processor TSS.</td>
<td>23-29</td>
<td></td>
</tr>
<tr>
<td>23.28.2 TSS Selector Writes</td>
<td>23-29</td>
<td></td>
</tr>
<tr>
<td>23.28.3 Order of Reads/Writes to the TSS</td>
<td>23-29</td>
<td></td>
</tr>
<tr>
<td>23.28.4 Using A 16-Bit TSS with 32-Bit Constructs</td>
<td>23-29</td>
<td></td>
</tr>
<tr>
<td>23.28.5 Differences in I/O Map Base Addresses</td>
<td>23-29</td>
<td></td>
</tr>
<tr>
<td>23.29 CACHE MANAGEMENT</td>
<td>23-30</td>
<td></td>
</tr>
<tr>
<td>23.29.1 Self-Modifying Code with Cache Enabled</td>
<td>23-30</td>
<td></td>
</tr>
<tr>
<td>23.29.2 Disabling the L3 Cache</td>
<td>23-31</td>
<td></td>
</tr>
<tr>
<td>23.30 PAGING</td>
<td>23-31</td>
<td></td>
</tr>
<tr>
<td>23.30.1 Large Pages</td>
<td>23-31</td>
<td></td>
</tr>
<tr>
<td>23.30.2 PCO and PW T Flags</td>
<td>23-31</td>
<td></td>
</tr>
<tr>
<td>23.30.3 Enabling and Disabling Paging</td>
<td>23-32</td>
<td></td>
</tr>
<tr>
<td>23.31 STACK OPERATIONS AND SUPERVISOR SOFTWARE</td>
<td>23-32</td>
<td></td>
</tr>
<tr>
<td>23.31.1 Selector Pushes and Pops</td>
<td>23-32</td>
<td></td>
</tr>
<tr>
<td>23.31.2 Error Code Pushes</td>
<td>23-32</td>
<td></td>
</tr>
<tr>
<td>23.31.3 Fault Handling Effects on the Stack</td>
<td>23-33</td>
<td></td>
</tr>
<tr>
<td>23.31.4 Interlevel RET/IRET from a 16-Bit Interrupt or Call Gate</td>
<td>23-33</td>
<td></td>
</tr>
<tr>
<td>23.32 MIXING 16- AND 32-BIT SEGMENTS</td>
<td>23-33</td>
<td></td>
</tr>
<tr>
<td>23.33 SEGMENT AND ADDRESS WRAPAROUND</td>
<td>23-33</td>
<td></td>
</tr>
<tr>
<td>23.33.1 Segment Wraparound</td>
<td>23-34</td>
<td></td>
</tr>
<tr>
<td>23.34 STORE BUFFERS AND ADDRESS ORDERING</td>
<td>23-35</td>
<td></td>
</tr>
<tr>
<td>23.35 BUS LOCKING</td>
<td>23-35</td>
<td></td>
</tr>
<tr>
<td>23.36 BUS HOLD</td>
<td>23-35</td>
<td></td>
</tr>
<tr>
<td>23.37 MODEL-SPECIFIC EXTENSIONS TO THE IA-32</td>
<td>23-35</td>
<td></td>
</tr>
<tr>
<td>23.37.1 Model-Specific Registers</td>
<td>23-36</td>
<td></td>
</tr>
<tr>
<td>23.37.2 RDSR and WR MSR Instructions</td>
<td>23-36</td>
<td></td>
</tr>
<tr>
<td>23.37.3 Memory Type Range Registers</td>
<td>23-36</td>
<td></td>
</tr>
<tr>
<td>23.37.4 Machine-Check Exception and Architecture</td>
<td>23-36</td>
<td></td>
</tr>
<tr>
<td>23.37.5 Performance-Monitoring Counters</td>
<td>23-37</td>
<td></td>
</tr>
<tr>
<td>23.38 Two WAYS TO RUN INTEL 286 PROCESSOR TASKS</td>
<td>23-37</td>
<td></td>
</tr>
<tr>
<td>23.39 INITIAL STATE OF PENTIUM, PENTIUM PRO AND PENTIUM 4 PROCESSORS</td>
<td>23-37</td>
<td></td>
</tr>
</tbody>
</table>

CHAPTER 24
INTRODUCTION TO VIRTUAL MACHINE EXTENSIONS

24.1 OVERVIEW                                                      24-1
24.2 VIRTUAL MACHINE ARCHITECTURE                                  24-1
24.3 INTRODUCTION TO VMX OPERATION                                  24-1
24.4 LIFE CYCLE OF VMM SOFTWARE                                     24-2
24.5 VIRTUAL-MACHINE CONTROL STRUCTURE                             24-2
24.6 DISCOVERING SUPPORT FOR VMX                                   24-3
24.7 ENABLING AND ENTERING VMX OPERATION                            24-3
24.8 RESTRICTIONS ON VMX OPERATION                                  24-3

CHAPTER 25
VIRTUAL MACHINE CONTROL STRUCTURES

25.1 OVERVIEW                                                     25-1
25.2 FORMAT OF THE VMCS REGION                                      25-2
25.3 ORGANIZATION OF VMCS DATA                                     25-3
25.4 GUEST-STATE AREA                                              25-4
25.4.1 Guest Register State                                        25-4
25.4.2 Guest Non-Register State                                    25-4
25.5 HOST-STATE AREA                                              25-6
25.6 VM-EXECUTION CONTROL FIELDS                                   25-8
25.6.1 Pin-Based VM-Execution Controls                             25-9
25.6.2 Processor-Based VM-Execution Controls                      25-10
25.6.3 Exception Bitmap                                           25-14
25.6.4 I/O-Bitmap Addresses                                       25-14
25.6.5 Time-Stamp Counter Offset and Multiplier                   25-14
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4</td>
<td>25-14</td>
</tr>
<tr>
<td>25.6.7 CR3-Target Controls</td>
<td>25-14</td>
</tr>
<tr>
<td>25.6.8 Controls for APIC Virtualization</td>
<td>25-15</td>
</tr>
<tr>
<td>25.6.9 MSR-Bitmap Address</td>
<td>25-16</td>
</tr>
<tr>
<td>25.6.10 Executive-VMCS Pointer</td>
<td>25-16</td>
</tr>
<tr>
<td>25.6.11 Extended-Page-Table Pointer (EPTP)</td>
<td>25-17</td>
</tr>
<tr>
<td>25.6.12 Virtual-Processor Identifier (VPID)</td>
<td>25-17</td>
</tr>
<tr>
<td>25.6.13 Controls for PAUSE-Loop Exiting</td>
<td>25-17</td>
</tr>
<tr>
<td>25.6.14 VM-Function Controls</td>
<td>25-18</td>
</tr>
<tr>
<td>25.6.15 VMCS Shadowing Bitmap Addresses</td>
<td>25-18</td>
</tr>
<tr>
<td>25.6.16 ENCLS-Exiting Bitmap</td>
<td>25-18</td>
</tr>
<tr>
<td>25.6.17 ENCLV-Exiting Bitmap</td>
<td>25-18</td>
</tr>
<tr>
<td>25.6.18 PCONFIG-Exiting Bitmap</td>
<td>25-18</td>
</tr>
<tr>
<td>25.6.19 Control Field for Page-Modification Logging</td>
<td>25-19</td>
</tr>
<tr>
<td>25.6.20 Controls for Virtualization Exceptions</td>
<td>25-19</td>
</tr>
<tr>
<td>25.6.21 XSS-Exiting Bitmap</td>
<td>25-19</td>
</tr>
<tr>
<td>25.6.22 Sub-Page-Permission-Table Pointer (SPPTP)</td>
<td>25-19</td>
</tr>
<tr>
<td>25.6.23 Fields Related to Hypervisor-Managed Linear-Address Translation</td>
<td>25-20</td>
</tr>
<tr>
<td>25.6.24 Fields Related to PASID Translation</td>
<td>25-20</td>
</tr>
<tr>
<td>25.6.25 Instruction-Timeout Control</td>
<td>25-20</td>
</tr>
<tr>
<td>25.6.26 Fields Controlling Virtualization of the IA32_SPEC_CTRL MSR</td>
<td>25-21</td>
</tr>
<tr>
<td>25.7 VM-EXIT CONTROL FIELDS</td>
<td>25-21</td>
</tr>
<tr>
<td>25.7.1 VM-Exit Controls</td>
<td>25-21</td>
</tr>
<tr>
<td>25.7.2 VM-Exit Controls for MSRs</td>
<td>25-22</td>
</tr>
<tr>
<td>25.8 VM-ENTRY CONTROL FIELDS</td>
<td>25-23</td>
</tr>
<tr>
<td>25.8.1 VM-Entry Controls</td>
<td>25-23</td>
</tr>
<tr>
<td>25.8.2 VM-Entry Controls for MSRs</td>
<td>25-24</td>
</tr>
<tr>
<td>25.8.3 VM-Entry Controls for Event Injection</td>
<td>25-24</td>
</tr>
<tr>
<td>25.9 VM-EXIT INFORMATION FIELDS</td>
<td>25-25</td>
</tr>
<tr>
<td>25.9.1 Basic VM-Exit Information</td>
<td>25-25</td>
</tr>
<tr>
<td>25.9.2 Information for VM Exits Due to Vectored Events</td>
<td>25-27</td>
</tr>
<tr>
<td>25.9.3 Information for VM Exits That Occur During Event Delivery</td>
<td>25-27</td>
</tr>
<tr>
<td>25.9.4 Information for VM Exits Due to Instruction Execution</td>
<td>25-28</td>
</tr>
<tr>
<td>25.9.5 VM-Instruction Error Field</td>
<td>25-29</td>
</tr>
<tr>
<td>25.10 VMCS TYPES: ORDINARY AND SHADOW</td>
<td>25-29</td>
</tr>
<tr>
<td>25.11 SOFTWARE USE OF THE VMCS AND RELATED STRUCTURES</td>
<td>25-29</td>
</tr>
<tr>
<td>25.11.1 Software Use of Virtual-Machine Control Structures</td>
<td>25-29</td>
</tr>
<tr>
<td>25.11.2 VMREAD, VMWRITE, and Encodings of VMCS Fields</td>
<td>25-30</td>
</tr>
<tr>
<td>25.11.3 Initializing a VMCS</td>
<td>25-32</td>
</tr>
<tr>
<td>25.11.4 Software Access to Related Structures</td>
<td>25-32</td>
</tr>
<tr>
<td>25.11.5 VMXON Region</td>
<td>25-32</td>
</tr>
</tbody>
</table>

## CHAPTER 26

**VMX NON-ROOT OPERATION**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1 INSTRUCTIONS THAT CAUSE VM EXITS</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1.1 Relative Priority of Faults and VM Exits</td>
<td>26-1</td>
</tr>
<tr>
<td>26.1.2 Instructions That Cause VM Exits Unconditionally</td>
<td>26-2</td>
</tr>
<tr>
<td>26.1.3 Instructions That Cause VM Exits Conditionally</td>
<td>26-2</td>
</tr>
<tr>
<td>26.2 OTHER CAUSES OF VM EXITS</td>
<td>26-5</td>
</tr>
<tr>
<td>26.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION</td>
<td>26-7</td>
</tr>
<tr>
<td>26.4 OTHER CHANGES IN VMX NON-ROOT OPERATION</td>
<td>26-13</td>
</tr>
<tr>
<td>26.4.1 Event Blocking</td>
<td>26-13</td>
</tr>
<tr>
<td>26.4.2 Treatment of Task Switches</td>
<td>26-13</td>
</tr>
<tr>
<td>26.4.3 Shadow-Stack Updates</td>
<td>26-14</td>
</tr>
<tr>
<td>26.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION</td>
<td>26-14</td>
</tr>
<tr>
<td>26.5.1 VMX-Preemption Timer</td>
<td>26-14</td>
</tr>
<tr>
<td>26.5.2 Monitor Trap Flag</td>
<td>26-15</td>
</tr>
<tr>
<td>26.5.3 Translation of Guest-Physical Addresses Using EPT</td>
<td>26-16</td>
</tr>
<tr>
<td>26.5.4 Translation of Guest-Physical Addresses Used by Intel Processor Trace</td>
<td>26-16</td>
</tr>
<tr>
<td>26.5.4.1 Guest-Physical Address Translation for Intel PT: Details</td>
<td>26-16</td>
</tr>
<tr>
<td>26.5.4.2 Trace-Address Pre-Translation (TAPT)</td>
<td>26-17</td>
</tr>
<tr>
<td>26.5.5 APIC Virtualization</td>
<td>26-17</td>
</tr>
<tr>
<td>26.5.6 VM Functions</td>
<td>26-17</td>
</tr>
<tr>
<td>26.5.6.1 Enabling VM Functions</td>
<td>26-17</td>
</tr>
</tbody>
</table>
### CHAPTER 27
**VM ENTRIES**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.1 BASIC VM-ENTRY CHECKS</td>
<td>27-2</td>
</tr>
<tr>
<td>27.2 CHECKS ON VMX CONTROLS AND HOST-STATE AREA</td>
<td>27-2</td>
</tr>
<tr>
<td>27.2.1 Checks on VMX Controls</td>
<td>27-2</td>
</tr>
<tr>
<td>27.2.1.1 VM-Execution Control Fields</td>
<td>27-2</td>
</tr>
<tr>
<td>27.2.1.2 VM-Exit Control Fields</td>
<td>27-5</td>
</tr>
<tr>
<td>27.2.1.3 VM-Entry Control Fields</td>
<td>27-6</td>
</tr>
<tr>
<td>27.2.2 Checks on Host Control Registers, MSRs, and SSP</td>
<td>27-7</td>
</tr>
<tr>
<td>27.2.3 Checks on Host Segment and Descriptor-Table Registers</td>
<td>27-7</td>
</tr>
<tr>
<td>27.2.4 Checks Related to Address-Space Size</td>
<td>27-8</td>
</tr>
<tr>
<td>27.3 CHECKING AND LOADING GUEST STATE</td>
<td>27-8</td>
</tr>
<tr>
<td>27.3.1 Checks on the Guest State Area</td>
<td>27-8</td>
</tr>
<tr>
<td>27.3.1.1 Checks on Guest Control Registers, Debug Registers, and MSRs</td>
<td>27-8</td>
</tr>
<tr>
<td>27.3.1.2 Checks on Guest Segment Registers</td>
<td>27-10</td>
</tr>
<tr>
<td>27.3.1.3 Checks on Guest Descriptor-Table Registers</td>
<td>27-12</td>
</tr>
<tr>
<td>27.3.1.4 Checks on Guest RIP, RFLAGS, and SSP</td>
<td>27-12</td>
</tr>
<tr>
<td>27.3.1.5 Checks on Guest Non-Register State</td>
<td>27-13</td>
</tr>
<tr>
<td>27.3.1.6 Checks on Guest Page-Directory-Pointer-Table Entries</td>
<td>27-15</td>
</tr>
<tr>
<td>27.3.2 Loading Guest State</td>
<td>27-15</td>
</tr>
<tr>
<td>27.3.2.1 Loading Guest Control Registers, Debug Registers, and MSRs</td>
<td>27-15</td>
</tr>
<tr>
<td>27.3.2.2 Loading Guest Segment Registers and Descriptor-Table Registers</td>
<td>27-17</td>
</tr>
<tr>
<td>27.3.2.3 Loading Guest RIP, RFLAGS, and SSP</td>
<td>27-17</td>
</tr>
<tr>
<td>27.3.2.4 Loading Page-Directory-Pointer-Table Entries</td>
<td>27-18</td>
</tr>
<tr>
<td>27.3.2.5 Updating Non-Register State</td>
<td>27-18</td>
</tr>
<tr>
<td>27.3.3 Clearing Address-Range Monitoring</td>
<td>27-18</td>
</tr>
<tr>
<td>27.4 LOADING MSRS</td>
<td>27-18</td>
</tr>
<tr>
<td>27.5 TRACE-ADDRESS PRE-TRANSLATION (TAPT)</td>
<td>27-19</td>
</tr>
<tr>
<td>27.6 EVENT INJECTION</td>
<td>27-19</td>
</tr>
<tr>
<td>27.6.1 Vectored-Event Injection</td>
<td>27-19</td>
</tr>
<tr>
<td>27.6.1.1 Details of Vectored-Event Injection</td>
<td>27-20</td>
</tr>
<tr>
<td>27.6.1.2 VM Exits During Event Injection</td>
<td>27-21</td>
</tr>
<tr>
<td>27.6.1.3 Event Injection for VM Exits to Real-Address Mode</td>
<td>27-22</td>
</tr>
<tr>
<td>27.6.2 Injection of Pending MTF VM Exits</td>
<td>27-22</td>
</tr>
<tr>
<td>27.7 SPECIAL FEATURES OF VM ENTRY</td>
<td>27-22</td>
</tr>
<tr>
<td>27.7.1 Interruption State</td>
<td>27-22</td>
</tr>
<tr>
<td>27.7.2 Activity State</td>
<td>27-23</td>
</tr>
<tr>
<td>27.7.3 Delivery of Pending Debug Exceptions after VM Entry</td>
<td>27-24</td>
</tr>
<tr>
<td>27.7.4 VMX-Preemption Timer</td>
<td>27-25</td>
</tr>
<tr>
<td>27.7.5 Interrupt-Window Exiting and Virtual-Interrupt Delivery</td>
<td>27-25</td>
</tr>
<tr>
<td>27.7.6 NMI-Window Exiting</td>
<td>27-25</td>
</tr>
<tr>
<td>27.7.7 VM Exits Induced by the TPR Threshold</td>
<td>27-25</td>
</tr>
<tr>
<td>27.7.8 Pending MTF VM Exits</td>
<td>27-26</td>
</tr>
<tr>
<td>27.7.9 VM Entries and Advanced Debugging Features</td>
<td>27-26</td>
</tr>
<tr>
<td>27.7.10 User-Interrupt Recognition After VM Entry</td>
<td>27-26</td>
</tr>
<tr>
<td>27.8 VM-ENTRY FAILURES DURING OR AFTER LOADING GUEST STATE</td>
<td>27-26</td>
</tr>
<tr>
<td>27.9 MACHINE-CHECK EVENTS DURING VM ENTRY</td>
<td>27-27</td>
</tr>
</tbody>
</table>

### CHAPTER 28
**VM EXITS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.1 ARCHITECTURAL STATE BEFORE A VM EXIT</td>
<td>28-1</td>
</tr>
<tr>
<td>28.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS</td>
<td>28-4</td>
</tr>
<tr>
<td>28.2.1 Basic VM-Exit Information</td>
<td>28-4</td>
</tr>
</tbody>
</table>
30.2.3 Virtualizing User-Interrupt Notifications ................................................................. 30-6
30.3 VIRTUALIZING CR8-BASED TPR ACCESSES .............................................................. 30-6
30.4 VIRTUALIZING MEMORY-MAPPED APIC ACCESSES .................................................... 30-6
30.4.1 Priority of APIC-Access VM Exits .............................................................................. 30-7
30.4.2 Virtualizing Reads from the APIC-Access Page ...................................................... 30-8
30.4.3 Virtualizing Writes to the APIC-Access Page ......................................................... 30-9
30.4.3.1 Determining Whether a Write Access is Virtualized ............................................. 30-9
30.4.3.2 APIC-Write Emulation ......................................................................................... 30-10
30.4.3.3 APIC-Write VM Exits .......................................................................................... 30-11
30.4.4 Instruction-Specific Considerations .......................................................................... 30-11
30.4.5 Issues Pertaining to Page Size and TLB Management .............................................. 30-12
30.4.6 APIC Accesses Not Directly Resulting From Linear Addresses ............................... 30-12
30.4.6.1 Guest-Physical Accesses to the APIC-Access Page ............................................. 30-13
30.4.6.2 Physical Accesses to the APIC-Access Page ...................................................... 30-13
30.5 VIRTUALIZING MSR-BASED APIC ACCESSES .......................................................... 30-14
30.6 POSTED-INTERRUPT PROCESSING ........................................................................... 30-15
30.7 VIRTUALIZING SENDUIPI ....................................................................................... 30-16

CHAPTER 31
VMX INSTRUCTION REFERENCE

31.1 OVERVIEW .................................................................................................................. 31-1
31.2 CONVENTIONS ......................................................................................................... 31-2
31.3 VMX INSTRUCTIONS ............................................................................................... 31-2
    INVEPT—Invalid Translations Derived from EPT ............................................................ 31-3
    INVVPID—Invalid Translations Based on VPID ............................................................. 31-6
    VMCALL—Call to VM Monitor ....................................................................................... 31-9
    VMCLEAR—Clear Virtual-Machine Control Structure .................................................... 31-11
    VMFUNC—Invoke VM function ..................................................................................... 31-13
    VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine ........................................ 31-14
    VMPTRLD—Load Pointer to Virtual-Machine Control Structure ..................................... 31-17
    VMREAD—Read Field from Virtual-Machine Control Structure ...................................... 31-19
    VMWRITE—Write Field to Virtual-Machine Control Structure ....................................... 31-21
    VMRESUME—Resume Virtual Machine ......................................................................... 31-23
    VMWRITE—Write Field to Virtual-Machine Control Structure ....................................... 31-24
    VMXOFF—Leave VMX Operation .................................................................................... 31-26
    VMXON—Enter VMX Operation ...................................................................................... 31-28
31.4 VM INSTRUCTION ERROR NUMBERS ...................................................................... 31-31

CHAPTER 32
SYSTEM MANAGEMENT MODE

32.1 SYSTEM MANAGEMENT MODE OVERVIEW .............................................................. 32-1
32.1.1 System Management Mode and VMX Operation ..................................................... 32-2
32.2 SYSTEM MANAGEMENT INTERRUPT (SMI) ............................................................ 32-2
32.3 SWITCHING BETWEEN SMM AND THE OTHER PROCESSOR OPERATING MODES . 32-2
32.3.1 Entering SMM .......................................................................................................... 32-2
32.3.2 Exiting From SMM .................................................................................................... 32-3
32.4 SMRAM ........................................................................................................................ 32-4
32.4.1 SMRAM State Save Map .......................................................................................... 32-4
32.4.1.1 SMRAM State Save Map and Intel 64 Architecture .............................................. 32-5
32.4.2 SMRAM Caching ....................................................................................................... 32-8
32.4.2.1 System Management Range Registers (SMRR) .................................................... 32-9
32.5 SMI HANDLER EXECUTION ENVIRONMENT ............................................................ 32-9
32.5.1 Initial SMM Execution Environment ......................................................................... 32-9
32.5.2 SMI Handler Operating Mode Switching .................................................................. 32-10
32.5.3 Control-flow Enforcement Technology Interactions ................................................. 32-11
32.6 EXCEPTIONS AND INTERRUPTS WITHIN SMM ....................................................... 32-11
32.7 MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS ................................................................. 32-12
32.7.1 I/O State Implementation ......................................................................................... 32-12
32.7.2 NMI HANDLING WHILE IN SMM ............................................................................ 32-13
32.7.3 SMM REVISON IDENTIFIER .................................................................................... 32-13
32.10 AUTO HALT RESTART ............................................................................................... 32-14
32.10.1 Executing the HLT Instruction in SMM 32-14
32.11 SMBASE RELOCATION 32-14
32.12 I/O INSTRUCTION RESTART 32-15
32.12.1 Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used 32-16
32.13 SMM MULTIPLE-PROCESSOR CONSIDERATIONS 32-16
32.14 DEFAULT TREATMENT OF SMIs AND SMM WITH VMX OPERATION AND SMX OPERATION 32-16
32.14.1 Default Treatment of SMI Delivery 32-17
32.14.2 Default Treatment of RSM 32-18
32.14.3 Protection of CR4.VMXE in SMM 32-19
32.14.4 VMXOFF and SMI Unblocking 32-19
32.15 DUAL-MONITOR TREATMENT OF SMIs AND SMM 32-19
32.15.1 Dual-Monitor Treatment Overview 32-19
32.15.2 SMM VM Exits 32-20
32.15.2.1 Architectural State Before a VM Exit 32-20
32.15.2.2 Updating the Current-VMCS and Executive-VMCS Pointers 32-20
32.15.2.3 Recording VM-Exit Information 32-20
32.15.2.4 Saving Guest State 32-21
32.15.2.5 Updating State 32-21
32.15.3 Operation of the SMM-Transfer Monitor 32-22
32.15.4 VM Entries that Return from SMM 32-22
32.15.4.1 Checks on the Executive-VMCS Pointer Field 32-22
32.15.4.2 Checks on VM-Execution Control Fields 32-22
32.15.4.3 Checks on VM-Entry Control Fields 32-23
32.15.4.4 Checks on the Guest State Area 32-23
32.15.4.5 Loading Guest State 32-23
32.15.4.6 VMX-Preemption Timer 32-23
32.15.4.7 Updating the Current-VMCS and SMM-Transfer VMCS Pointers 32-24
32.15.4.8 VM Exits Induced by VM Entry 32-24
32.15.4.9 SMI Blocking 32-24
32.15.4.10 Failures of VM Entries That Return from SMM 32-24
32.15.5 Enabling the Dual-Monitor Treatment 32-25
32.15.5.1 Initial Checks 32-26
32.15.5.2 Updating the Current-VMCS and Executive-VMCS Pointers 32-27
32.15.5.3 Saving Guest State 32-27
32.15.5.4 Saving MSRs 32-27
32.15.5.5 Loading Host State 32-27
32.15.5.6 Loading MSRs 32-29
32.15.5.7 Deactivating the Dual-Monitor Treatment 32-29
32.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT 32-29
32.17 MODEL-SPECIFIC SYSTEM MANAGEMENT ENHANCEMENT 32-30
32.17.1 SMM Handler Code Access Control 32-30
32.17.2 SMI Delivery Delay Reporting 32-30
32.17.3 Blocked SMI Reporting 32-30

CHAPTER 33
INTEL® PROCESSOR TRACE
33.1 OVERVIEW 33-1
33.1.1 Features and Capabilities 33-1
33.1.1.1 Packet Summary 33-1
33.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL 33-2
33.2.1 Change of Flow Instruction (COFI) Tracing 33-2
33.2.1.1 Direct Transfer COFI 33-3
33.2.1.2 Indirect Transfer COFI 33-3
33.2.1.3 Far Transfer COFI 33-4
33.2.2 Software Trace Instrumentation with PWRITE 33-4
33.2.3 Power Event Tracing 33-4
33.2.4 Event Tracing 33-5
33.2.5 Trace Filtering 33-5
33.2.5.1 Filtering by Current Privilege Level (CPL) 33-5
33.2.5.2 Filtering by CR3 33-5
33.2.5.3 Filtering by IP 33-6
33.2.6 Packet Generation Enable Controls 33-7
33.2.6.1 Packet Enable (PacketEn) 33-7
INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

CHAPTER 34

INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

34.1 OVERVIEW

34.2 ENCLAVE INTERACTION AND PROTECTION

34.3 ENCLAVE LIFE CYCLE
CHAPTER 35
ENCLAVE ACCESS CONTROL AND DATA STRUCTURES

35.1 OVERVIEW OF ENCLAVE EXECUTION ENVIRONMENT
35.2 TERMINOLOGY
35.3 ACCESS-CONTROL REQUIREMENTS
35.4 SEGMENT-BASED ACCESS CONTROL
35.5 PAGE-BASED ACCESS CONTROL
35.5.1 Access-control for Accesses that Originate from Non-SGX Instructions
35.5.2 Memory Accesses that Split Across ELRANGE
35.5.3 Implicit vs. Explicit Accesses
35.5.3.1 Explicit Accesses
35.5.3.2 Implicit Accesses
35.6 INTEL® SGX DATA STRUCTURES OVERVIEW
35.7 SGX ENCLAVE CONTROL STRUCTURE (SECS)
35.7.1 ATTRIBUTES
35.7.2 SECS.MISCSELECT Field
35.7.3 SECS.CET_ATTRIBUTES Field
35.8 THREAD CONTROL STRUCTURE (TCS)
35.8.1 TCS.FLAGS
35.8.2 State Save Area Offset (OSSA)
35.8.3 Current State Save Area Frame (CSSA)
35.8.4 Number of State Save Area Frames (NSSA)
35.9 STATE SAVE AREA (SSA) FRAME
35.9.1 GPRSGX Region
35.9.1.1 EXITINFO
35.9.1.2 VECTOR Field Definition
35.9.2 MISC Region
35.9.2.1 EXINFO Structure
35.9.2.2 Page Fault Error Code
35.10 CET STATE SAVE AREA FRAME
35.11 PAGE INFORMATION (PAGINFO)
35.12 SECURITY INFORMATION (SECINFO)
35.12.1 SECINFO_FLAGS
35.12.2 PAGE_TYPE Field Definition
35.13 PAGING CRYPTO METADATA (PCMD)
35.14 ENCLAVE SIGNATURE STRUCTURE (SIGSTRUCT)
35.15 EINIT TOKEN STRUCTURE (EINITTOKEN)
35.16 REPORT (REPORT)
35.16.1 REPORTDATA
35.17 REPORT TARGET INFO (TARGETINFO)
35.18 KEY REQUEST (KEYREQUEST)
35.18.1 KEY REQUEST KeyNames
35.18.2 Key Request Policy Structure
35.19 VERSION ARRAY (VA)
35.20 ENCLAVE PAGE CACHE MAP (EPCM)
35.21 READ INFO (RDINFO)
35.21.1 RDINFO Status Structure
35.21.2 RDINFO Flags Structure
CHAPTER 38
INTEL® SGX INSTRUCTION REFERENCES

38.1 INTEL® SGX INSTRUCTION SYNTAX AND OPERATION .......................................................... 38-1
  38.1.1 ENCLS Register Usage Summary .............................................................................. 38-1
  38.1.2 ENCLU Register Usage Summary ............................................................................. 38-2
  38.1.3 ENCLV Register Usage Summary .............................................................................. 38-2
  38.1.4 Information and Error Codes .................................................................................. 38-2
  38.1.5 Internal REGS ......................................................................................................... 38-3
  38.1.6 Concurrent Operation Restrictions ......................................................................... 38-4
  38.1.6.1 Concurrency Tables of Intel® SGX Instructions .................................................... 38-4
  38.2 INTEL® SGX INSTRUCTION REFERENCE .................................................................... 38-8
    ENCLS—Execute an Enclave System Function of Specified Leaf Number .................. 38-9
    ENCLU—Execute an Enclave User Function of Specified Leaf Number .................. 38-11
    ENCLV—Execute an Enclave VMM Function of Specified Leaf Number .................. 38-14
  38.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE ..................................................... 38-16
    EADD—Add a Page to an Uninitialized Enclave ......................................................... 38-17
    EAUG—Add a Page to an Initialized Enclave .............................................................. 38-22
    EBLOCK—Mark a page in EPC as Blocked .................................................................. 38-27
    ECREATE—Create an SECS page in the Enclave Page Cache ...................................... 38-30
    EDBGRD—Read From a Debug Enclave ..................................................................... 38-36
    EDBGWR—Write to a Debug Enclave ........................................................................ 38-40
    EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes ...................... 38-44
    EINIT—Initialize an Enclave for Execution .................................................................. 38-47
    ELDB/ELDU/ELDBC/ELDUC—Load an EPC Page and Mark its State ...................... 38-55
    EMODPR—Restrict the Permissions of an EPC Page ..................................................... 38-61
    EMODT—Change the Type of an EPC Page .................................................................. 38-64
    EPA—Add Version Array ............................................................................................ 38-67
    ERDINFO—Read Type and Status Information About an EPC Page ......................... 38-69
    ERREMOVE—Remove a page from the EPC ............................................................... 38-73
    ETRACK/ETRACT—Activates EBLOCK Checks .............................................................. 38-77
    ETRACKC—Activates EBLOCK Checks ...................................................................... 38-80
    EW—Invalidates an EPC Page and Write out to Main Memory ................................. 38-84
  38.4 INTEL® SGX USER LEAF FUNCTION REFERENCE ............................................................ 38-89
    EACCEPT—Accept Changes to an EPC Page .................................................................. 38-90
    EACCEPTCOPY—Initialize a Pending Page .................................................................. 38-95
    EDECCSSA—Decrements TCS.CSSA .......................................................................... 38-99
    ENTER—Enters an Enclave ......................................................................................... 38-103
    EEXIT—Exits an Enclave ............................................................................................. 38-112
    EGETKEY—Retrieves a Cryptographic Key ................................................................ 38-115
    EMODPE—Extend an EPC Page Permissions ............................................................... 38-125
    EREPORT—Create a Cryptographic Report of the Enclave ...................................... 38-128
    ERESUME—Re-Enters an Enclave ............................................................................. 38-133
  38.5 INTEL® SGX VIRTUALIZATION LEAF FUNCTION REFERENCE ..................................... 38-145
    EDEC/DEC/VIRT—Decrement VIRTCHILDCTN in SECS ............................................. 38-146
    EINCR/INC/VIRT—Increment VIRTCHILDCTN in SECS .............................................. 38-150
    ESETCONTEXT—Set the ENCLAVECONTEXT Field in SECS ..................................... 38-153

CHAPTER 39
INTEL® SGX INTERACTIONS WITH IA32 AND INTEL® 64 ARCHITECTURE

39.1 INTEL® SGX AVAILABILITY IN VARIOUS PROCESSOR MODES ........................................... 39-1
39.2 IA32_FEATURE_CONTROL ............................................................................................... 39-1
39.2.1 Availability of Intel SGX .......................................................................................... 39-1
39.2.2 Intel SGX Launch Control Configuration ............................................................... 39-1
39.3 INTERACTIONS WITH SEGMENTATION ......................................................................... 39-1
39.3.1 Scope of Interaction ............................................................................................... 39-1
39.3.2 Interactions of Intel® SGX Instructions with Segment, Operand, and Addressing Prefixes .................................................................................................................. 39-2
39.3.3 Interaction of Intel® SGX Instructions with Segmentation ..................................... 39-2
39.3.4 Interactions of Enclave Execution with Segmentation ............................................ 39-2
39.4 INTERACTIONS WITH Paging ....................................................................................... 39-2
39.5 INTERACTIONS WITH VMX ........................................................................................... 39-3
CONTENTS

39.5.1 VMM Controls to Configure Guest Support of Intel® SGX ...................................................... 39-3
39.5.2 Interactions with the Extended Page Table Mechanism (EPT) .............................................. 39-3
39.5.3 Interactions with APIC Virtualization ....................................................................................... 39-4
39.5.4 Interactions with VT and SGX concurrency ............................................................................ 39-4
39.5.5 Virtual Child Tracking .............................................................................................................. 39-5
39.5.6 Handling EPCM Entry Lock Conflicts .................................................................................... 39-5
39.5.7 Context Tracking .................................................................................................................... 39-6
39.6 INTEL® SGX INTERACTIONS WITH ARCHITECTURALLY-VISIBLE EVENTS ... 39-6
39.7 INTERACTIONS WITH THE PROCESSOR EXTENDED STATE AND MISCELLANEOUS STATE . 39-6
39.7.1 Requirements and Architecture Overview .............................................................................. 39-6
39.7.2 Relevant Fields in Various Data Structures ............................................................................. 39-7
39.7.2.1 SECS.ATTRIBUTES.XFRM ................................................................................................. 39-7
39.7.2.2 SECS.SSAFRAMESIZE ....................................................................................................... 39-8
39.7.2.3 XSAVE Area in SSA ........................................................................................................... 39-8
39.7.2.4 MISC Area in SSA ............................................................................................................ 39-8
39.7.2.5 SIGSTRUCT Fields ............................................................................................................ 39-8
39.7.2.6 REPORT.ATTRIBUTES.XFRM and REPORT.MISCESELECT ........................................... 39-9
39.7.2.7 KEYREQUEST ................................................................................................................... 39-9
39.7.3 Processor Extended States and ENCLU[CREATE] ................................................................. 39-9
39.7.4 Processor Extended States and ENCLU[ENTER] ................................................................. 39-9
39.7.4.1 Fault Checking .................................................................................................................. 39-9
39.7.4.2 State Loading .................................................................................................................. 39-9
39.7.5 Processor Extended States and AEX ....................................................................................... 39-10
39.7.5.1 State Saving ..................................................................................................................... 39-10
39.7.5.2 State Synthesis .................................................................................................................. 39-10
39.7.6 Processor Extended States and ENCLU[ERESUME] .......................................................... 39-10
39.7.6.1 Fault Checking .................................................................................................................. 39-10
39.7.6.2 State Loading .................................................................................................................. 39-10
39.7.7 Processor Extended States and ENCLU[EXIT] ...................................................................... 39-10
39.7.8 Processor Extended States and ENCLU[REPORT] ............................................................. 39-11
39.7.9 Processor Extended States and ENCLU[GETKEY] .............................................................. 39-11
39.8 INTERACTIONS WITH SMM .................................................................................................... 39-11
39.8.1 Availability of Intel® SGX instructions in SMM ................................................................. 39-11
39.8.2 SMI while Inside an Enclave ................................................................................................. 39-11
39.8.3 SMRAM Synthetic State of AEX Triggered by SMI ........................................................... 39-11
39.9 INTERACTIONS OF INIT, SIPI, AND WAIT-FOR-SIPI WITH INTEL® SGX . 39-12
39.10 INTERACTIONS WITH DMA .................................................................................................... 39-12
39.11 INTERACTIONS WITH TXT ....................................................................................................... 39-12
39.11.1 Enclaves Created Prior to Execution of GETSEC ............................................................... 39-12
39.11.2 Interaction of GETSEC with Intel® SGX ............................................................................ 39-12
39.11.3 Interactions with Authenticated Code Modules (ACMs) ...................................................... 39-13
39.12 INTERACTIONS WITH CACHING OF LINEAR-ADDRESS TRANSLATIONS .................................................................................................................. 39-13
39.13 INTERACTIONS WITH INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX) . 39-13
39.13.1 HLE and RTM Debug .......................................................................................................... 39-14
39.14 INTEL® SGX INTERACTIONS WITH S STATES ....................................................................... 39-14
39.14.1 Interactions with MCA Events .............................................................................................. 39-14
39.15 Machine Check Enables (IA32_MCI_CTL) ............................................................................. 39-14
39.15.3 CR4.MCE ............................................................................................................................. 39-14
39.16 INTEL® SGX INTERACTIONS WITH PROTECTED MODE VIRTUAL INTERRUPTS . 39-15
39.17 INTEL SGX INTERACTION WITH PROTECTION KEYS .......................................................... 39-15

CHAPTER 40

ENCLAVE CODE DEBUG AND PROFILING

40.1 CONFIGURATION AND CONTROLS ............................................................................................. 40-1
40.1.1 Debug Enclave vs. Production Enclave .................................................................................. 40-1
40.1.2 Tool-Chain Opt-In ................................................................................................................... 40-1
40.1.3 Debugging an Enclave That Uses Asynchronous Enclave Exit Notify .................................. 40-1
40.2 SINGLE STEP DEBUG ............................................................................................................... 40-1
40.2.1 Single Stepping ENCL executes Instruction Level ............................... 40-2
40.2.2 Single Stepping UNCL Instruction Level .............................................................................. 40-2
40.2.3 Single-Stepping Enclave Entry with Opt-out Enable .......................................................... 40-2
40.2.3.1 Single Stepping without AEX .......................................................................................... 40-2
40.2.3.2 Single Step Prempted by AEX Due to Non-SMI Event ..................................................... 40-2
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.3.4</td>
<td>32-Bit Host-State Field</td>
<td>B-9</td>
</tr>
<tr>
<td>B.4</td>
<td>NATURAL-WIDTH FIELDS</td>
<td>B-9</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Natural-Width Control Fields</td>
<td>B-9</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Natural-Width Read-Only Data Fields</td>
<td>B-10</td>
</tr>
<tr>
<td>B.4.3</td>
<td>Natural-Width Guest-State Fields</td>
<td>B-10</td>
</tr>
<tr>
<td>B.4.4</td>
<td>Natural-Width Host-State Fields</td>
<td>B-11</td>
</tr>
</tbody>
</table>

### APPENDIX C

**VMX BASIC EXIT REASONS**
<table>
<thead>
<tr>
<th>FIGURES</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1. Bit and Byte Order</td>
<td>1-8</td>
</tr>
<tr>
<td>Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation</td>
<td>1-9</td>
</tr>
<tr>
<td>Figure 2-1. IA-32 System-Level Registers and Data Structures</td>
<td>2-2</td>
</tr>
<tr>
<td>Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode</td>
<td>2-3</td>
</tr>
<tr>
<td>Figure 2-3. Transitions Among the Processor’s Operating Modes</td>
<td>2-8</td>
</tr>
<tr>
<td>Figure 2-4. IA32_EFER MSR Layout</td>
<td>2-9</td>
</tr>
<tr>
<td>Figure 2-5. System Flags in the EFLAGS Register</td>
<td>2-10</td>
</tr>
<tr>
<td>Figure 2-6. Memory Management Registers</td>
<td>2-12</td>
</tr>
<tr>
<td>Figure 2-7. Control Registers</td>
<td>2-14</td>
</tr>
<tr>
<td>Figure 2-8. XCR0</td>
<td>2-21</td>
</tr>
<tr>
<td>Figure 2-9. Format of Protection-Key Rights Registers</td>
<td>2-22</td>
</tr>
<tr>
<td>Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy</td>
<td>2-26</td>
</tr>
<tr>
<td>Figure 3-1. Segmentation and Paging</td>
<td>3-2</td>
</tr>
<tr>
<td>Figure 3-2. Flat Model</td>
<td>3-3</td>
</tr>
<tr>
<td>Figure 3-3. Protected Flat Model</td>
<td>3-4</td>
</tr>
<tr>
<td>Figure 3-4. Multi-Segment Model</td>
<td>3-5</td>
</tr>
<tr>
<td>Figure 3-5. Logical Address to Linear Address Translation</td>
<td>3-7</td>
</tr>
<tr>
<td>Figure 3-6. Segment Selector</td>
<td>3-7</td>
</tr>
<tr>
<td>Figure 3-7. Segment Registers</td>
<td>3-8</td>
</tr>
<tr>
<td>Figure 3-8. Segment Descriptor</td>
<td>3-10</td>
</tr>
<tr>
<td>Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear</td>
<td>3-11</td>
</tr>
<tr>
<td>Figure 3-10. Global and Local Descriptor Tables</td>
<td>3-15</td>
</tr>
<tr>
<td>Figure 3-11. Pseudo-Descriptor Formats</td>
<td>3-16</td>
</tr>
<tr>
<td>Figure 4-1. Enabling and Changing Paging Modes</td>
<td>4-4</td>
</tr>
<tr>
<td>Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging</td>
<td>4-10</td>
</tr>
<tr>
<td>Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging</td>
<td>4-11</td>
</tr>
<tr>
<td>Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging</td>
<td>4-11</td>
</tr>
<tr>
<td>Figure 4-5. Linear-Address Translation to a 4-KByte Page using PAE Paging</td>
<td>4-16</td>
</tr>
<tr>
<td>Figure 4-6. Linear-Address Translation to a 2-MByte Page using PAE Paging</td>
<td>4-17</td>
</tr>
<tr>
<td>Figure 4-7. Formats of CR3 and Paging-Structure Entries with PAE Paging</td>
<td>4-19</td>
</tr>
<tr>
<td>Figure 4-8. Linear-Address Translation to a 4-KByte Page Using 4-Level Paging</td>
<td>4-22</td>
</tr>
<tr>
<td>Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging</td>
<td>4-23</td>
</tr>
<tr>
<td>Figure 4-10. Linear-Address Translation to a 1-GByte Page Using 4-Level Paging</td>
<td>4-23</td>
</tr>
<tr>
<td>Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging</td>
<td>4-32</td>
</tr>
<tr>
<td>Figure 4-12. Page-Fault Error Code</td>
<td>4-37</td>
</tr>
<tr>
<td>Figure 4-13. Memory Management Convention That Assigns a Page Table to Each Segment</td>
<td>4-55</td>
</tr>
<tr>
<td>Figure 5-1. Descriptor Fields Used for Protection</td>
<td>5-3</td>
</tr>
<tr>
<td>Figure 5-2. Descriptor Fields with Flags used in IA-32e Mode</td>
<td>5-4</td>
</tr>
<tr>
<td>Figure 5-3. Protection Rings</td>
<td>5-7</td>
</tr>
<tr>
<td>Figure 5-4. Privilege Check for Data Access</td>
<td>5-8</td>
</tr>
<tr>
<td>Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels</td>
<td>5-9</td>
</tr>
<tr>
<td>Figure 5-6. Privilege Check for Control Transfer Without Using a Gate</td>
<td>5-11</td>
</tr>
<tr>
<td>Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels</td>
<td>5-12</td>
</tr>
<tr>
<td>Figure 5-8. Call-Gate Descriptor</td>
<td>5-13</td>
</tr>
<tr>
<td>Figure 5-9. Call-Gate Descriptor in IA-32e Mode</td>
<td>5-14</td>
</tr>
<tr>
<td>Figure 5-10. Call-Gate Mechananism</td>
<td>5-15</td>
</tr>
<tr>
<td>Figure 5-11. Privilege Check for Control Transfer with Call Gate</td>
<td>5-16</td>
</tr>
<tr>
<td>Figure 5-12. Example of Accessing Call Gates At Various Privilege Levels</td>
<td>5-17</td>
</tr>
<tr>
<td>Figure 5-13. Stack Switching During an Interprivilege-Level Call</td>
<td>5-19</td>
</tr>
<tr>
<td>Figure 5-14. MSRs Used by SYSCALL and SYSRET</td>
<td>5-23</td>
</tr>
<tr>
<td>Figure 5-15. Use of RPL to Weaken Privilege Level of Called Procedure</td>
<td>5-26</td>
</tr>
<tr>
<td>Figure 6-1. Relationship of the IDTR and IDT</td>
<td>6-10</td>
</tr>
<tr>
<td>Figure 6-2. IDT Gate Descriptors</td>
<td>6-11</td>
</tr>
<tr>
<td>Figure 6-3. Interrupt Procedure Call</td>
<td>6-12</td>
</tr>
<tr>
<td>Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines</td>
<td>6-13</td>
</tr>
<tr>
<td>Figure 6-5. Shadow Stack Usage on Transfers to Interrupt and Exception-Handling Routines</td>
<td>6-15</td>
</tr>
<tr>
<td>Figure 6-6. Interrupt Task Switch</td>
<td>6-18</td>
</tr>
<tr>
<td>Figure 6-7. Error Code</td>
<td>6-19</td>
</tr>
<tr>
<td>Figure 6-8. 64-Bit IDT Gate Descriptors</td>
<td>6-20</td>
</tr>
<tr>
<td>Figure 6-9. IA-32e Mode Stack Usage After Privilege Level Change</td>
<td>6-22</td>
</tr>
<tr>
<td>Figure 6-10. Interrupt Shadow Stack Table</td>
<td>6-23</td>
</tr>
<tr>
<td>Figure 6-11. Page-Fault Error Code</td>
<td>6-46</td>
</tr>
</tbody>
</table>
Figure 6-12. Exception Error Code Information .................................................. 6-56
Figure 8-1. Structure of a Task ....................................................................... 8-2
Figure 8-2. 32-Bit Task-State Segment (TSS). ............................................... 8-4
Figure 8-3. TSS Descriptor ........................................................................... 8-6
Figure 8-4. Format of TSS and LDTR Descriptors in 64-bit Mode .................. 8-7
Figure 8-5. Task Register ............................................................................. 8-8
Figure 8-6. Task-Gate Descriptor ................................................................. 8-8
Figure 8-7. Task Gates Referencing the Same Task ....................................... 8-9
Figure 8-8. Nested Tasks ............................................................................. 8-15
Figure 8-9. Overlapping Linear-to-Physical Mappings ................................. 8-17
Figure 8-10. 16-Bit TSS Format ................................................................... 8-19
Figure 8-11. 64-Bit TSS Format ................................................................... 8-20
Figure 9-1. Example of Write Ordering in Multiple-Processor Systems ....... 9-8
Figure 9-2. Interpretation of APIC ID in Early MP Systems ......................... 9-25
Figure 9-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology .................................................. 9-27
Figure 9-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology .................................................. 9-28
Figure 9-5. Generalized Seven-Domain Interpretation of the APIC ID .......... 9-35
Figure 9-6. Conceptual Six-Domain Topology and 32-bit APIC ID Composition ................................................................................................. 9-36
Figure 9-7. Topological Relationships Between Hierarchical IDs in a Hypothetical MP Platform .......................................................... 9-39
Figure 9-8. MP System With Multiple Pentium III Processors .................... 9-56
Figure 10-1. Contents of CR0 Register after Reset ....................................... 10-2
Figure 10-2. Version Information in the EDX Register after Reset .......... 10-5
Figure 10-3. Processor State After Reset ........................................................ 10-15
Figure 10-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File) .................................................. 10-23
Figure 10-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File) .................................................. 10-24
Figure 10-6. Task Switching (Lines 282-296 of List File) ................................ 10-25
Figure 10-7. Applying Microcode Updates .................................................... 10-28
Figure 10-8. Microcode Update Write Operation Flow [1] .............................. 10-45
Figure 10-9. Microcode Update Write Operation Flow [2] .............................. 10-46
Figure 11-1. Relationship of Local APIC and I/O APIC in Single-Processor Systems ....................................................................................... 11-2
Figure 11-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems ........................................ 11-3
Figure 11-3. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems ........................................ 11-3
Figure 11-4. Local APIC Structure ................................................................. 11-5
Figure 11-5. IA32_APIC_BASE MSR (APIC_BASE MSR in P6 Family) .......... 11-9
Figure 11-6. Local APIC ID Register ............................................................ 11-9
Figure 11-7. Local APIC Version Register .................................................... 11-11
Figure 11-8. Local Vector Table (LVT) .......................................................... 11-13
Figure 11-9. Error Status Register (ESR) ..................................................... 11-15
Figure 11-10. Divide Configuration Register ................................................. 11-17
Figure 11-11. Initial Count and Current Count Registers ............................ 11-17
Figure 11-12. Interrupt Command Register (ICR) ........................................ 11-19
Figure 11-13. Local Destination Register (LDR) .......................................... 11-24
Figure 11-14. Destination Format Register (DFR) ......................................... 11-24
Figure 11-15. Arbitration Priority Register (APR) ........................................ 11-25
Figure 11-16. Interrupt Acceptance Flow Chart for the Local APIC (Pentium 4 and Intel Xeon Processors) .................................................. 11-27
Figure 11-17. Interrupt Acceptance Flow Chart for the Local APIC (P6 Family and Pentium Processors) .................................................. 11-28
Figure 11-18. Task-Priority Register (TPR) .................................................... 11-29
Figure 11-19. Processor-Priority Register (PPR) ........................................... 11-29
Figure 11-20. IRR, ISR, and TMR Registers .................................................. 11-30
Figure 11-21. EOI Register ........................................................................ 11-31
Figure 11-22. CR8 Register ........................................................................ 11-32
Figure 11-23. Spurious-Interrupt Vector Register (SVR) .............................. 11-33
Figure 11-24. Layout of the MSI Message Address Register ........................ 11-34
Figure 11-25. Layout of the MSI Message Data Register ............................. 11-36
Figure 11-26. IA32_APIC_BASE MSR Supporting x2APIC ......................... 11-37
Figure 11-27. Local x2APIC State Transitions with IA32_APIC_BASE_INIT, and Reset .................................................. 11-42
Figure 11-28. Interrupt Command Register (ICR) in x2APIC Mode ............ 11-45
Figure 11-29. Logical Destination Register in x2APIC Mode ....................... 11-46
Figure 11-30. SELF IPI register .................................................................. 11-47
Figure 12-1. Cache Structure of the Pentium 4 and Intel Xeon Processors 12-1
Figure 12-2. Cache Structure of the Intel Core i7 Processors ....................... 12-2
Figure 12-3. Cache-Control Registers and Bits Available in Intel 64 and IA-32 Processors .................................................. 12-11
Figure 12-4. Mapping Physical Memory With MTRRs .............................. 12-21
Figure 12-5. IA32_MTRRCAP Register ...................................................... 12-22
Figure 12-6. IA32_MTRRCAP DEF_TYPE MSR .............................. 12-23
| Figure 34-1. | An Enclave Within the Application's Virtual Address Space | 34-1 |
| Figure 36-1. | Enclave Memory Layout | 36-1 |
| Figure 36-2. | Measurement Flow of Enclave Build Process | 36-7 |
| Figure 36-3. | SGX Local Attestation | 36-9 |
| Figure 37-1. | Exit Stack Just After Interrupt with Stack Switch | 37-1 |
| Figure 37-2. | The SSA Stack | 37-2 |
| Figure 38-1. | Relationships Between SECS, SIGSTRUCT, and EINITTOKEN | 38-47 |
| Figure 40-1. | Single Stepping with Opt-out Entry - No AEX | 40-2 |
| Figure 40-2. | Single Stepping with Opt-out Entry - AEX Due to Non-SMI Event Before Single-Step Boundary | 40-3 |
| Figure 40-3. | LBR Stack Interaction with Opt-in Entry | 40-6 |
| Figure 40-4. | LBR Stack Interaction with Opt-out Entry | 40-7 |
TABLES

Table 2-1. IA32_EFER MSR Information ............................................................. 2-9
Table 2-2. Action Taken by x87 FPU Instructions for Different Combinations of EM, MP, and TS .............................................................. 2-16
Table 2-3. Summary of System Instructions ................................................................ 2-23
Table 3-1. Code- and Data-Segment Types .................................................................. 3-12
Table 3-2. System-Segment and Gate-Descriptor Types .............................................. 3-14
Table 4-1. Properties of Different Paging Modes ...................................................... 4-2
Table 4-2. Paging Structures in the Different Paging Modes ....................................... 4-8
Table 4-3. Use of CR3 with 32-Bit Paging ............................................................... 4-12
Table 4-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page ........ 4-12
Table 4-5. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page ............... 4-13
Table 4-6. Format of a 32-Bit Page-Directory Entry that References a Page Table .... 4-13
Table 4-7. Use of CR3 with PAE Paging ............................................................... 4-14
Table 4-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE) .......... 4-15
Table 4-9. Format of a PAE Page-Directory Entry that Maps a 2-MByte Page ........... 4-17
Table 4-10. Format of a PAE Page-Directory Entry that References a Page Table ..... 4-18
Table 4-11. Format of a PEA Page-Directory Entry that Maps a 4-KByte Page ......... 4-18
Table 4-12. Use of CR3 with 4-Level Paging and 5-Level Paging and CR4.PCIDE = 0 ... 4-20
Table 4-13. Use of CR3 with 4-Level Paging and 5-Level Paging and CR4.PCIDE = 1 ... 4-21
Table 4-14. Format of a PML5 Entry (PML5E) that References a PML4 Table ......... 4-26
Table 4-15. Format of a PML4 Entry (PML4E) that References a Page-Directory-Pointer Table .................................................................................. 4-26
Table 4-16. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page ................................................................. 4-27
Table 4-17. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory ................................................................. 4-28
Table 4-18. Format of a Page-Directory Entry that Maps a 2-MByte Page .......... 4-29
Table 4-19. Format of a Page-Directory Entry that References a Page Table .......... 4-30
Table 4-20. Format of a Page-Directory Entry that Maps a 4-KByte Page .......... 4-31
Table 5-1. Privilege Check Rules for Call Gates .......................................................... 5-16
Table 5-2. 64-Bit-Mode Stack Layout After Far CALL with CPL Change ................. 5-19
Table 5-3. Combined Page-Directory and Page-Table Protection .......................... 5-29
Table 5-4. Extended Feature Enable MSR (IA32_EFER) .......................................... 5-30
Table 5-5. 4-KByte Page Level Protection Matrix with Execute-Disable Bit Capability with PAE Paging ................................................................. 5-31
Table 5-6. 2-MByte Page Level Protection Matrix with Execute-Disable Bit Capability with PAE Paging ................................................................. 5-31
Table 5-7. Page Level Protection Matrix with Execute-Disable Bit Capability with 4-Level Paging ............................................................... 5-31
Table 5-8. Page Level Protection Matrix with Execute-Disable Bit Capability Enabled ............................................................... 5-32
Table 5-9. Reserved Bit Checking with Execute-Disable Bit Capability Not Enabled .......................................................................................... 5-32
Table 5-10. Code- and Data-Segment Types ............................................................ 5-34
Table 5-11. Conditions for Generating a Double Fault ............................................. 5-34
Table 5-12. Invalid TSS Conditions ........................................................................ 5-36
Table 5-13. Alignment Requirements by Data Type ................................................ 6-50
Table 5-14. SIMD Floating-Point Exceptions Priority .......................................... 6-54
Table 5-15. Format of User Posted-Interrupt Descriptor — UPID ......................... 7-5
Table 8-1. Exception Conditions Checked During a Task Switch .......................... 8-13
Table 8-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag ................................................................. 8-15
Table 8-3. Broadcast INIT-SIPI-SIPI Sequence and Choice of Timeouts .............. 9-23
Table 9-1. IA-32 and Intel® 64 Processor States Following Power-up, Reset, or INIT .................................................................................. 10-2
Table 9-2. Variance of RESET Values in Selected Intel Architecture Processors .... 10-4
Table 9-3. Recommended Settings of EM and MP Flags on IA-32 Processors ......... 10-6
Table 9-4. Software Emulation Settings of EM, MP, and NE Flags ......................... 10-7
Table 9-5. Main Initialization Steps in STARTUP.ASM Source Listing ................. 10-16
Table 9-6. Relationship Between BLD Item and ASM Source File ................... 10-27
Table 9-7. Microcode Update Field Definitions .................................................... 10-28
Table 9-8. Microcode Update Format ................................................................. 10-30
Table 10-1. Microcode Update Format Table Header Structure ............................ 10-31
Table 20-47. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture) .......... 20-67
Table 20-46. Core PMU Summary of the Ice Lake Microarchitecture. .......................... 20-67
Table 20-48. MSR_OFFCORE_RSP_x Supplier Info Field Definition (Processors Based on Ice Lake Microarchitecture). 20-68
Table 20-49. MSR_OFFCORE_RSP_x Snoop Info Field Definition (Processors Based on Ice Lake Microarchitecture). 20-69
Table 20-50. Core PMU Summary of the Golden Cove Microarchitecture. ....................... 20-71
Table 20-51. Special Performance Monitoring Events with Counter Restrictions .................. 20-72
Table 20-52. Core PMU Summary of the Gracemont Microarchitecture .......................... 20-73
Table 20-53. E-core PEBS Memory Access Info Encoding ........................................... 20-73
Table 20-54. E-core PEBS Data Source Encodings ..................................................... 20-74
Table 20-55. MSR_OFFCORE_RSPx Request Type Definition ........................................ 20-75
Table 20-56. PEBS Performance Events for Knights Landing Microarchitecture .................... 20-77
Table 20-57. PEBS Record Format for Knights Landing Microarchitecture ........................ 20-77
Table 20-58. OffCore Response Event Encoding ......................................................... 20-78
Table 20-59. Bit fields of the MSR_OFFCORE_RESP [0,1] Registers .............................. 20-78
Table 20-60. PEBS Performance Events for the Silvermont Microarchitecture ...................... 20-81
Table 20-61. OffCore Response Event Encoding ......................................................... 20-82
Table 20-62. PEBS Record Format for the Silvermont Microarchitecture .......................... 20-82
Table 20-63. MSR_OFFCORE_RSPx Request_Type Field Definition ............................... 20-83
Table 20-64. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition .................... 20-84
Table 20-65. MSR_OFFCORE_RSPx Snoop Info Field Definition ................................... 20-84
Table 20-66. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures. 20-85
Table 20-67. Precise Events Supported by the Goldmont Microarchitecture ....................... 20-87
Table 20-68. PEBS Record Format for the Goldmont Microarchitecture ............................ 20-88
Table 20-69. MSR_OFFCORE_RSPx Request_Type Field Definition ............................... 20-89
Table 20-70. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests ...................... 20-90
Table 20-71. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures. 20-91
Table 20-72. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures. 20-92
Table 20-73. New Fields in IA32_PBS_ENABLE .......................................................... 20-93
Table 20-74. MSR_OFFCORE_RSPx Request_Type Definition ....................................... 20-94
Table 20-75. MSR_OFFCORE_RSPx Response_Type Definition ...................................... 20-95
Table 20-76. MSR_OFFCORE_RSPx Snoop Info Definition .......................................... 20-95
Table 20-77. Core Specificity Encoding within a Non-Architectural Umask ...................... 20-96
Table 20-78. Agent Specificity Encoding within a Non-Architectural Umask ..................... 20-97
Table 20-79. HW Prefetch Qualification Encoding within a Non-Architectural Umask .......... 20-97
Table 20-80. MESI Qualification Definitions within a Non-Architectural Umask ............... 20-97
Table 20-81. Bus Snoop Qualification Definitions within a Non-Architectural Umask .......... 20-98
Table 20-82. Snoop Type Qualification Definitions within a Non-Architectural Umask ........ 20-98
Table 20-83. At-Retirement Performance Events for Intel Core Microarchitecture .......... 20-101
Table 20-84. PEBS Performance Events for Intel Core Microarchitecture ....................... 20-102
Table 20-85. Requirements to Program PEBS ............................................................ 20-103
Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) ......................................................... 20-104
Table 20-87. Event Example ....................................................................................... 20-111
Table 20-88. CCR Names and Bit Positions ............................................................... 20-115
Table 20-89. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events .......................................................... 20-123
Table 20-90. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events .......................................................... 20-124
Table 20-91. Nominal Core Crystal Clock Frequency ................................................... 20-140
Table 20-92. Basic Info Group ..................................................................................... 20-146
Table 20-93. Memory Access Info Group ..................................................................... 20-147
Table 20-94. Updated Memory Access Info Group ....................................................... 20-148
Table 20-95. GPRs in Ice Lake Microarchitecture ....................................................... 20-149
Table 20-96. XMMs .................................................................................................... 20-149
Table 20-97. LBRs ........................................................................................................ 20-150
Table 20-98. MSR_PBS_CFG Programming ................................................................. 20-151
Table 20-99. PEBS Record Example 1 ....................................................................... 20-151
Table 20-100. PEBS Record Example 2 ..................................................................... 20-152
Table 20-101. Data Source Encoding for Memory Acceses (Ice Lake and Later Microarchitectures) ............................................................ 20-155
Table 21-1. Real-Address Mode Exceptions and Interrupts .......................................... 21-6
Table 21-2. Software Interrupt Handling Methods while in Virtual-8086 Mode ............. 21-17
Table 21-3. Characteristics of 16-Bit and 32-Bit Program Modules ............................. 22-1
Table 21-4. New Instruction in the Pentium Miroprocessor and Later IA-32 Processors ..... 23-3
Table 23-3. EM and MP Flag Interpretation .................................................................. 23-17
Table 23-2. Recommended Values of the EM, MP, and NE Flags for Intel486 SX Microprocessor/Intel 487 SX Math Coprocessor System .................. 23-17
Table 23-4. Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment .......................... 23-22
Table 33-2. IP Filtering Packet Example ................................................................. 33-7
Table 33-3. ToPA Table Entry Fields ................................................................. 33-12
Table 33-4. Algorithm to Manage Intel PT ToPA PMI and XSAVE/XRSTORS ........... 33-15
Table 33-5. Behavior on Restricted Memory Access ........................................... 33-16
Table 33-6. IA32_RTIT_CTL MSR ................................................................. 33-17
Table 33-7. IA32_RTIT_STATUS MSR ............................................................ 33-21
Table 33-8. IA32_RTIT_OUTPUT_BASE MSR .................................................. 33-23
Table 33-9. TSX Packet Scenarios with BranchEn=1 .......................................... 33-24
Table 33-10. IA32_RTIT_OUTPUT_MASK_PTRS MSR ........................................ 33-24
Table 33-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities .... 33-27
Table 33-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities ................................................................. 33-29
Table 33-13. An Illustrative CYC Packet Example ............................................. 33-34
Table 33-14. Compound Packet Event Summary ............................................... 33-37
Table 33-15. Packets Forbidden Between BBP and BEP ..................................... 33-37
Table 33-16. TNT Packet Definition ................................................................. 33-39
Table 33-17. IP Packet Definition ................................................................. 33-40
Table 33-18. FUP/TIP IP Reconstruction ......................................................... 33-41
Table 33-19. TNT Examples with Deferred TIPs .............................................. 33-42
Table 33-20. TIP.PGE Packet Definition .......................................................... 33-43
Table 33-21. TIP.PGP Packet Definition .......................................................... 33-44
Table 33-22. FUP Packet Definition ............................................................... 33-45
Table 33-23. FUP Cases and IP Payload ......................................................... 33-46
Table 33-24. PIP Packet Definition ............................................................... 33-47
Table 33-25. General Form of MODE Packets ................................................. 33-48
Table 33-26. MODE.Exec Packet Definition ...................................................... 33-49
Table 33-27. MODE.TSX Packet Definition .................................................... 33-50
Table 33-28. TraceStop Packet Definition ....................................................... 33-50
Table 33-29. CBR Packet Definition ............................................................... 33-50
Table 33-30. TSC Packet Definition ............................................................... 33-51
Table 33-31. MTC Packet Definition ............................................................... 33-52
Table 33-32. TMA Packet Definition ............................................................... 33-53
Table 33-33. Cycle Count Packet Definition .................................................... 33-54
Table 33-34. VMCS Packet Definition ............................................................. 33-55
Table 33-35. OVF Packet Definition ............................................................... 33-56
Table 33-36. PSB Packet Definition ............................................................... 33-56
Table 33-37. PSBEND Packet Definition .......................................................... 33-57
Table 33-38. MNT Packet Definition ............................................................... 33-58
Table 33-39. PAD Packet Definition ............................................................... 33-58
Table 33-40. PTW Packet Definition ............................................................... 33-59
Table 33-41. EXSTOP Packet Definition ......................................................... 33-60
Table 33-42. MWAIT Packet Definition ........................................................... 33-61
Table 33-43. PWR Packet Definition ............................................................. 33-62
Table 33-44. PVRX Packet Definition ............................................................. 33-63
Table 33-45. Block Begin Packet Definition ..................................................... 33-64
Table 33-46. Block Item Packet Definition ...................................................... 33-65
Table 33-47. BIP Encodings ......................................................................... 33-65
Table 33-48. Block End Packet Definition ....................................................... 33-70
Table 33-49. Control Flow Event Packet Definition .......................................... 33-71
Table 33-50. CFE Packet Type and VectorFields Details .................................... 33-71
Table 33-51. Event Data Packet Definition ....................................................... 33-73
Table 33-52. VMX Controls For Intel Processor Trace ....................................... 33-74
Table 33-53. Packets on VMX Transitions (System-Wide Tracing) ....................... 33-75
Table 33-54. Packets on a Failed VM Entry ..................................................... 33-76
Table 33-55. Packet Generation under Different Example Operations ................. 33-77
Table 33-56. Packet Generation with Operations That Alter the Value of PacketEn ................................................................. 33-78
Table 33-57. Examples of PTWR when TriggerEn & PTWEn is True ...................... 33-79
Table 33-58. Examples of Power Event Trace when TriggerEn & PwEvEn is True .... 33-79
Table 33-59. Event Trace Examples when TriggerEn & ContextEn & EventEn is True ................................................................. 33-80
Table 34-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1 ................................................................. 34-3
Table 34-2. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX2 ................................................................. 34-4
Table 34-3. VMX Operation and Supervisor Mode Enclave Instruction Leaf Functions in Long-Form of OVERSUB ................................................................. 34-4
Table 34-4. Intel® SGX Opt-in and Enabling Behavior ........................................ 34-5
Table 34-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities .... 34-5
Table 34-6. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities .... 34-6
Table 34-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources ................................................................. 34-6
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-1</td>
<td>Encoding for 32-Bit Host-State Field (0100_11xx_xxxx_xxx0B)</td>
<td>38-64</td>
</tr>
<tr>
<td>B-2</td>
<td>Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-3</td>
<td>Encodings for 32-Bit Read-Only Data Fields (0100_01xx_xxxx_xxx0B)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-4</td>
<td>Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-5</td>
<td>Encoding for 64-Bit Host-State Field (0010_11xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-6</td>
<td>Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-7</td>
<td>Encodings for 64-Bit Read-Only Data Field (0010_01xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-8</td>
<td>Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-9</td>
<td>Encoding for 32-Bit Read-Only Data Field (0100_11xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-10</td>
<td>Encoding for 32-Bit Control Fields (0100_10xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>B-11</td>
<td>Encoding for 32-Bit Host-State Field (0100_01xx_xxxx_xxxxxAb)</td>
<td>38-70</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>B-12</td>
<td>Encodings for Natural-Width Control Fields (0110_00xx_xxxx_xxx0B)</td>
<td>B-9</td>
</tr>
<tr>
<td>B-13</td>
<td>Encodings for Natural-Width Read-Only Data Fields (0110_01xx_xxxx_xxx0B)</td>
<td>B-10</td>
</tr>
<tr>
<td>B-14</td>
<td>Encodings for Natural-Width Guest-State Fields (0110_10xx_xxxx_xxx0B)</td>
<td>B-10</td>
</tr>
<tr>
<td>B-15</td>
<td>Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xxx0B)</td>
<td>B-11</td>
</tr>
<tr>
<td>C-1</td>
<td>Basic Exit Reasons</td>
<td>C-1</td>
</tr>
</tbody>
</table>

- The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4: Model-Specific Registers (order number 335592).

The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C, & 3D, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, and Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, address the programming environment for classes of software that host operating systems. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, describes the model-specific registers of Intel 64 and IA-32 processors.

### 1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™ 2 Duo processor
- Intel® Core™ 2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
• Intel® Core™ 2 Extreme processor X7000 and X6800 series
• Intel® Core™ 2 Extreme QX6000 series
• Intel® Xeon® processor 7100 series
• Intel® Pentium® Dual-Core processor
• Intel® Xeon® processor 7200, 7300 series
• Intel® Core™ 2 Extreme QX9000 series
• Intel® Xeon® processor 5200, 5400, 7400 series
• Intel® Core™ 2 Extreme processor QX9000 and X9000 series
• Intel® Core™ 2 Quad processor Q9000 series
• Intel® Core™ 2 Duo processor E8000, T9000 series
• Intel® Core™ i7 processor
• Intel® Core™ i5 processor
• Intel® Xeon® processor E7-8800/4800/2800 product families
• Intel® Core™ i7-3930K processor
• 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
• Intel® Xeon® processor E3-1200 product family
• Intel® Xeon® processor E5-2400/1400 product family
• Intel® Xeon® processor E5-4600/2600/1600 product family
• 3rd generation Intel® Core™ processors
• Intel® Xeon® processor E3-1200 v2 product family
• Intel® Xeon® processor E5-2400/1400 v2 product families
• Intel® Xeon® processor E5-4600/2600/1600 v2 product families
• Intel® Xeon® processor E7-8800/4800/2800 v2 product families
• 4th generation Intel® Core™ processors
• The Intel® Core™ M processor family
• Intel® Core™ i7-59xx Processor Extreme Edition
• Intel® Core™ i7-49xx Processor Extreme Edition
• Intel® Xeon® processor E3-1200 v3 product family
• Intel® Xeon® processor E5-2600/1600 v3 product families
• 5th generation Intel® Core™ processors
• Intel® Xeon® processor E3-1200 v5 product family
• Intel® Xeon® processor E5 v4 family
• Intel® Xeon® processor X7-Z8000 and X5-Z8000 series
• Intel Atom® processor Z3400 series
• Intel Atom® processor Z3500 series
• 6th generation Intel® Core™ processors
• Intel® Xeon® processor E3-1500m v5 product family
• 7th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
• Intel® Xeon® Scalable Processor Family
• 8th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
• Intel® Xeon® E processors
• 9th generation Intel® Core™ processors
• 2nd generation Intel® Xeon® Scalable Processor Family
• 10th generation Intel® Core™ processors
• 11th generation Intel® Core™ processors
• 3rd generation Intel® Xeon® Scalable Processor Family
• 12th generation Intel® Core™ processors
• 13th generation Intel® Core™ processors
• 4th generation Intel® Xeon® Scalable Processor Family
• 5th generation Intel® Xeon® Scalable Processor Family
• Intel® Core™ Ultra 7 processors

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.


The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™ 2 Duo, Intel® Core™ 2 Quad, and Intel® Core™ 2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™ 2 Quad processor Q9000 series, and Intel® Core™ 2 Extreme processors QX9000, X9000 series, Intel® Core™ 2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel Atom® processor Z5xx series support IA-32 architecture. The Intel® Xeon processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™ 2 Duo, Intel® Core™ 2 Extreme, Intel® Core™ 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™ 2 Duo, Intel® Core™ 2 Extreme, Intel® Core™ 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor support Intel 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.
The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel Atom® processor Z8000 series is based on the Airmont microarchitecture.

The Intel Atom® processor Z3400 series and the Intel Atom® processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel ® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Scalable Processor Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Core™ Ultra 7 processor is based on Meteor Lake hybrid architecture and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.
Chapter 1 — About This Manual. Gives an overview of all volumes of the Intel® 64 and IA-32 Architectures Software Developer’s Manual. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — System Architecture Overview. Describes the modes of operation used by Intel 64 and IA-32 processors and the mechanisms provided by the architectures to support operating systems and executives, including the system-oriented registers and data structures and the system-oriented instructions. The steps necessary for switching between real-address and protected modes are also identified.

Chapter 3 — Protected-Mode Memory Management. Describes the data structures, registers, and instructions that support segmentation and paging. The chapter explains how they can be used to implement a “flat” (unsegmented) memory model or a segmented memory model.

Chapter 4 — Paging. Describes the paging modes supported by Intel 64 and IA-32 processors.

Chapter 5 — Protection. Describes the support for page and segment protection provided in the Intel 64 and IA-32 architectures. This chapter also explains the implementation of privilege rules, stack switching, pointer validation, user mode, and supervisor mode.

Chapter 6 — Interrupt and Exception Handling. Describes the basic interrupt mechanisms defined in the Intel 64 and IA-32 architectures, shows how interrupts and exceptions relate to protection, and describes how the architecture handles each exception type. Reference information for each exception is given in this chapter. Includes programming the LINT0 and LINT1 inputs and gives an example of how to program the LINT0 and LINT1 pins for specific interrupt vectors.

Chapter 7 — User Interrupts. Describes user interrupts supported by Intel 64 and IA-32 processors.

Chapter 8 — Task Management. Describes mechanisms the Intel 64 and IA-32 architectures provide to support multitasking and inter-task protection.

Chapter 9 — Multiple-Processor Management. Describes the instructions and flags that support multiple processors with shared memory, memory ordering, and Intel® Hyper-Threading Technology. Includes MP initialization for P6 family processors and gives an example of how to use the MP protocol to boot P6 family processors in an MP system.

Chapter 10 — Processor Management and Initialization. Defines the state of an Intel 64 or IA-32 processor after reset initialization. This chapter also explains how to set up an Intel 64 or IA-32 processor for real-address mode operation and protected- mode operation, and how to switch between modes.

Chapter 11 — Advanced Programmable Interrupt Controller (APIC). Describes the programming interface to the local APIC and gives an overview of the interface between the local APIC and the I/O APIC. Includes APIC bus message formats and describes the message formats for messages transmitted on the APIC bus for P6 family and Pentium processors.

Chapter 12 — Memory Cache Control. Describes the general concept of caching and the caching mechanisms supported by the Intel 64 or IA-32 architectures. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory. Information on using the new cache control and memory streaming instructions introduced with the Pentium III, Pentium 4, and Intel Xeon processors is also given.

Chapter 13 — Intel® MMX™ Technology System Programming. Describes those aspects of the Intel® MMX™ technology that must be handled and considered at the system programming level, including: task switching, exception handling, and compatibility with existing system environments.

Chapter 14 — System Programming For Instruction Set Extensions And Processor Extended States. Describes the operating system requirements to support SSE/SSE2/SSE3/SSSE3/SSE4 extensions, including task switching, exception handling, and compatibility with existing system environments. The latter part of this chapter describes the extensible framework of operating system requirements to support processor extended states. Processor extended state may be required by instruction set extensions beyond those of SSE/SSE2/SSE3/SSSE3/SSE4 extensions.

Chapter 15 — Power and Thermal Management. Describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

1. Model-Specific Registers have been moved out of this volume and into a separate volume: Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
Chapter 16 — Machine-Check Architecture. Describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, and P6 family processors. Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

Chapter 17 — Interpreting Machine-Check Error Codes. Gives an example of how to interpret the error codes for a machine-check error that occurred on a P6 family processor.

Chapter 18 — Debug, Branch Profile, TSC, and Resource Monitoring Features. Describes the debugging registers and other debug mechanism provided in Intel 64 or IA-32 processors. This chapter also describes the time-stamp counter.

Chapter 19 — Last Branch Records. Describes the Last Branch Records (architectural feature).

Chapter 20 — Performance Monitoring. Describes the Intel 64 and IA-32 architectures' facilities for monitoring performance.

Chapter 21 — 8086 Emulation. Describes the real-address and virtual-8086 modes of the IA-32 architecture.

Chapter 22 — Mixing 16-Bit and 32-Bit Code. Describes how to mix 16-bit and 32-bit code modules within the same program or task.


Chapter 24 — Introduction to Virtual Machine Extensions. Describes the basic elements of virtual machine architecture and the virtual machine extensions for Intel 64 and IA-32 Architectures.

Chapter 25 — Virtual Machine Control Structures. Describes components that manage VMX operation. These include the working-VMCS pointer and the controlling-VMCS pointer.

Chapter 26 — VMX Non-Root Operation. Describes the operation of a VMX non-root operation. Processor operation in VMX non-root mode can be restricted programatically such that certain operations, events or conditions can cause the processor to transfer control from the guest (running in VMX non-root mode) to the monitor software (running in VMX root mode).

Chapter 27 — VM Entries. Describes VM entries. VM entry transitions the processor from the VMM running in VMX root-mode to a VM running in VMX non-root mode. VM-Entry is performed by the execution of VMLAUNCH or VMRESUME instructions.

Chapter 28 — VM Exits. Describes VM exits. Certain events, operations or situations while the processor is in VMX non-root operation may cause VM-exit transitions. In addition, VM exits can also occur on failed VM entries.

Chapter 29 — VMX Support for Address Translation. Describes virtual-machine extensions that support address translation and the virtualization of physical memory.

Chapter 30 — APIC Virtualization and Virtual Interrupts. Describes the VMCS including controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

Chapter 31 — VMX Instruction Reference. Describes the virtual-machine extensions (VMX). VMX is intended for a system executive to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.

Chapter 32 — System Management Mode. Describes Intel 64 and IA-32 architectures' system management mode (SMM) facilities.

Chapter 33 — Intel® Processor Trace. Describes details of Intel® Processor Trace.

Chapter 34 — Introduction to Intel® Software Guard Extensions. Provides an overview of the Intel® Software Guard Extensions (Intel® SGX) set of instructions.

Chapter 35 — Enclave Access Control and Data Structures. Describes Enclave Access Control procedures and defines various Intel SGX data structures.

Chapter 36 — Enclave Operation. Describes enclave creation and initialization, adding pages and measuring an enclave, and enclave entry and exit.

Chapter 37 — Enclave Exiting Events. Describes enclave-exiting events (EEE) and asynchronous enclave exit (AEX).

Chapter 38 — SGX Instruction References. Describes the supervisor and user level instructions provided by Intel SGX.
Chapter 39 — Intel® SGX Interactions with IA32 and Intel® 64 Architecture. Describes the Intel SGX collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel 64 architectures.

Chapter 40 — Enclave Code Debug and Profiling. Describes enclave code debug processes and options.

Appendix A — VMX Capability Reporting Facility. Describes the VMX capability MSRs. Support for specific VMX features is determined by reading capability MSRs.

Appendix B — Field Encoding in VMCS. Enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).

Appendix C — VM Basic Exit Reasons. Describes the 32-bit fields that encode reasons for a VM exit. Examples of exit reasons include, but are not limited to: software interrupts, processor exceptions, software traps, NMIs, external interrupts, and triple faults.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as reserved. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.
1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The “B” designation is only used in situations where confusion as to the type of number might arise.

1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.
The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

### 1.3.6 Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a single syntax to represent this type of information. See Figure 1-2.

![Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation](image)
1.3.7 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:


See also:

- The latest security information on Intel® products:
- Software developer resources, guidance, and insights for security advisories:
  https://software.intel.com/security-software-guidance/
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
- Intel® Fortran Compiler documentation and online help:
- Intel® Software Development Tools:
- Intel® 64 and IA-32 Architectures Software Developer’s Manual (in one, four or ten volumes):
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:
- Intel® Software Guard Extensions (Intel® SGX) Information
- Developing Multi-threaded Applications: A Platform Consistent Approach:
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  https://software.intel.com/sites/default/files/article/22/30/25602
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference

More relevant links are:
• Intel® Developer Zone:
  https://software.intel.com/en-us
• Developer centers:
• Processor support general link:
  http://www.intel.com/support/processors/
• Intel® Hyper-Threading Technology (Intel® HT Technology):
IA-32 architecture (beginning with the Intel386 processor family) provides extensive support for operating-system and system-development software. This support offers multiple modes of operation, which include:

- Real mode, protected mode, virtual 8086 mode, and system management mode. These are sometimes referred to as legacy modes.

Intel 64 architecture supports almost all the system programming facilities available in IA-32 architecture and extends them to a new operating mode (IA-32e mode) that supports a 64-bit programming environment. IA-32e mode allows software to operate in one of two sub-modes:

- 64-bit mode supports 64-bit OS and 64-bit applications
- Compatibility mode allows most legacy software to run; it co-exists with 64-bit applications under a 64-bit OS.

The IA-32 system-level architecture includes features to assist in the following operations:

- Memory management.
- Protection of software modules.
- Multitasking.
- Exception and interrupt handling.
- Multiprocessing.
- Cache management.
- Hardware resource and power management.
- Debugging and performance monitoring.

This chapter provides a description of each part of this architecture. It also describes the system registers that are used to set up and control the processor at the system level and gives a brief overview of the processor’s system-level (operating system) instructions.

Many features of the system-level architecture are used only by system programmers. However, application programmers may need to read this chapter and the following chapters in order to create a reliable and secure environment for application programs.

This overview and most subsequent chapters of this book focus on protected-mode operation of the IA-32 architecture. IA-32e mode operation of the Intel 64 architecture, as it differs from protected mode operation, is also described.

All Intel 64 and IA-32 processors enter real-address mode following a power-up or reset (see Chapter 10, “Processor Management and Initialization”). Software then initiates the switch from real-address mode to protected mode. If IA-32e mode operation is desired, software also initiates a switch from protected mode to IA-32e mode.

2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE

System-level architecture consists of a set of registers, data structures, and instructions designed to support basic system-level operations such as memory management, interrupt and exception handling, task management, and control of multiple processors.

Figure 2-1 provides a summary of system registers and data structures that applies to 32-bit modes. System registers and data structures that apply to IA-32e mode are shown in Figure 2-2.
Figure 2-1. IA-32 System-Level Registers and Data Structures
2.1.1 Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2-1. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments as well as access rights, type, and usage information.
Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information.

To access a byte in a segment, a segment selector and an offset must be supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

See Figure 2-1. The solid arrows in the figure indicate a linear address, dashed lines indicate a segment selector, and the dotted arrows indicate a physical address. For simplicity, many of the segment selectors are shown as direct pointers to a segment. However, the actual path from a segment selector to its associated segment is always through a GDT or LDT.

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR).

2.1.1.1 Global and Local Descriptor Tables in IA-32e Mode
GDTR and LDTR registers are expanded to 64-bits wide in both IA-32e sub-modes (64-bit mode and compatibility mode). For more information: see Section 3.5.2, "Segment Descriptor Tables in IA-32e Mode."

Global and local descriptor tables are expanded in 64-bit mode to support 64-bit base addresses, (16-byte LDT descriptors hold a 64-bit base address and various attributes). In compatibility mode, descriptors are not expanded.

2.1.2 System Segments, Segment Descriptors, and Gates
Besides code, data, and stack segments that make up the execution environment of a program or procedure, the architecture defines two system segments: the task-state segment (TSS) and the LDT. The GDT is not considered a segment because it is not accessed by means of a segment selector and segment descriptor. TSSs and LDTs have segment descriptors defined for them.

The architecture also defines a set of special descriptors called gates (call gates, interrupt gates, trap gates, and task gates). These provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures. For example, a CALL to a call gate can provide access to a procedure in a code segment that is at the same or a numerically lower privilege level (more privileged) than the current code segment. To access a procedure through a call gate, the calling procedure1 supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate.

If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task. Gates also facilitate transitions between 16-bit and 32-bit code segments, and vice versa.

2.1.2.1 Gates in IA-32e Mode
In IA-32e mode, the following descriptors are 16-byte descriptors (expanded to allow a 64-bit base): LDT descriptors, 64-bit TSSs, call gates, interrupt gates, and trap gates.

Call gates facilitate transitions between 64-bit mode and compatibility mode. Task gates are not supported in IA-32e mode. On privilege level changes, stack segment selectors are not read from the TSS. Instead, they are set to NULL.

1. The word “procedure” is commonly used in this document as a general term for a logical unit or block of code (such as a program, procedure, function, or routine).
2.1.3 Task-State Segments and Task Gates

The TSS (see Figure 2-1) defines the state of the execution environment for a task. It includes the state of general-purpose registers, segment registers, the EFLAGS register, the EIP register, and segment selectors with stack pointers for three stack segments (one stack for each privilege level). The TSS also includes the segment selector for the LDT associated with the task and the base address of the paging-structure hierarchy.

All program execution in protected mode happens within the context of a task (called the current task). The segment selector for the TSS for the current task is stored in the task register. The simplest method for switching to a task is to make a call or jump to the new task. Here, the segment selector for the TSS of the new task is given in the CALL or JMP instruction. In switching tasks, the processor performs the following actions:

1. Stores the state of the current task in the current TSS.
2. Loads the task register with the segment selector for the new task.
3. Accesses the new TSS through a segment descriptor in the GDT.
4. Loads the state of the new task from the new TSS into the general-purpose registers, the segment registers, the LDTR, control register CR3 (base address of the paging-structure hierarchy), the EFLAGS register, and the EIP register.
5. Begins execution of the new task.

A task can also be accessed through a task gate. A task gate is similar to a call gate, except that it provides access (through a segment selector) to a TSS rather than a code segment.

2.1.3.1 Task-State Segments in IA-32e Mode

Hardware task switches are not supported in IA-32e mode. However, TSSs continue to exist. The base address of a TSS is specified by its descriptor.

A 64-bit TSS holds the following information that is important to 64-bit operation:

- Stack pointer addresses for each privilege level.
- Pointer addresses for the interrupt stack table.
- Offset address of the IO-permission bitmap (from the TSS base).

The task register is expanded to hold 64-bit base addresses in IA-32e mode. See also: Section 8.7, "Task Management in 64-bit Mode."

2.1.4 Interrupt and Exception Handling

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. Like the GDT, the IDT is not a segment. The linear address for the base of the IDT is contained in the IDT register (IDTR).

Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector from internal hardware, an external interrupt controller, or from software by means of an INT n, INTO, INT3, INT1, or BOUND instruction. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

2.1.4.1 Interrupt and Exception Handling IA-32e Mode

In IA-32e mode, interrupt gate descriptors are expanded to 16 bytes to support 64-bit base addresses. This is true for 64-bit mode and compatibility mode.

The IDTR register is expanded to hold a 64-bit base address. Task gates are not supported.
2.1.5 Memory Management

System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address. When paging is used: all code, data, stack, and system segments (including the GDT and IDT) can be paged with only the most recently accessed pages being held in physical memory.

The location of pages (sometimes called page frames) in physical memory is contained in the paging structures. These structures reside in physical memory (see Figure 2-1 for the case of 32-bit paging).

The base physical address of the paging-structure hierarchy is contained in control register CR3. The entries in the paging structures determine the physical address of the base of a page frame, access rights and memory management information.

To use this paging mechanism, a linear address is broken into parts. The parts provide separate offsets into the paging structures and the page frame. A system can have a single hierarchy of paging structures or several. For example, each task can have its own hierarchy.

2.1.5.1 Memory Management in IA-32e Mode

In IA-32e mode, physical memory pages are managed by a set of system data structures. In both compatibility mode and 64-bit mode, four or five levels of system data structures are used (see Chapter 4, "Paging"). These include the following:

- **The page map level 5 (PML5)** — An entry in the PML5 table contains the physical address of the base of a PML4 table, access rights, and memory management information. The base physical address of the PML5 table is stored in CR3. The PML5 table is used only with 5-level paging.

- **A page map level 4 (PML4)** — An entry in a PML4 table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. With 4-level paging, there is only one PML4 table and its base physical address is stored in CR3.

- **A set of page directory pointer tables** — An entry in a page directory pointer table contains the physical address of the base of a page directory table, access rights, and memory management information.

- **Sets of page directories** — An entry in a page directory table contains the physical address of the base of a page table, access rights, and memory management information.

- **Sets of page tables** — An entry in a page table contains the physical address of a page frame, access rights, and memory management information.

2.1.6 System Registers

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the EFLAGS register and several system registers:

- The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights. See also: Section 2.3, "System Flags and Fields in the EFLAGS Register."

- The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system or executive. See also: Chapter 2, "Control Registers," and Section 2.6, "Extended Control Registers (Including XCR0)."

- The debug registers (not shown in Figure 2-1) allow the setting of breakpoints for use in debugging programs and systems software. See also: Chapter 18, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features."

- The GDTR, LDTR, and IDTR registers contain the linear addresses and sizes (limits) of their respective tables. See also: Section 2.4, "Memory-Management Registers."

- The task register contains the linear address and size of the TSS for the current task. See also: Section 2.4, "Memory-Management Registers."

- Model-specific registers (not shown in Figure 2-1).
The model-specific registers (MSRs) are a group of registers available primarily to operating-system or executive procedures (that is, code running at privilege level 0). These registers control items such as the debug extensions, the performance-monitoring counters, the machine-check architecture, and the memory type ranges (MTRRs).

The number and function of these registers varies among different members of the Intel 64 and IA-32 processor families. See also: Section 10.4, “Model-Specific Registers (MSRs),” and Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

Most systems restrict access to system registers (other than the EFLAGS register) by application programs. Systems can be designed, however, where all programs and procedures run at the most privileged level (privilege level 0). In such a case, application programs would be allowed to modify the system registers.

### 2.1.6.1 System Registers in IA-32e Mode

In IA-32e mode, the four system-descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded in hardware to hold 64-bit base addresses. EFLAGS becomes the 64-bit RFLAGS register. CR0–CR4 are expanded to 64 bits. CR8 becomes available. CR8 provides read-write access to the task priority register (TPR) so that the operating system can control the priority classes of external interrupts.

In 64-bit mode, debug registers DR0–DR7 are 64 bits. In compatibility mode, address-matching in DR0–DR3 is also done at 64-bit granularity.

On systems that support IA-32e mode, the extended feature enable register (IA32_EFER) is available. This model-specific register controls activation of IA-32e mode and other IA-32e mode operations. In addition, there are several model-specific registers that govern IA-32e mode instructions:

- **IA32_KERNEL_GS_BASE** — Used by SWAPGS instruction.
- **IA32_LSTAR** — Used by SYSCALL instruction.
- **IA32_FMASK** — Used by SYSCALL instruction.
- **IA32_STAR** — Used by SYSCALL and SYSRET instruction.

### 2.1.7 Other System Resources

Besides the system registers and data structures described in the previous sections, system architecture provides the following additional resources:

- Operating system instructions (see also: Section 2.8, “System Instruction Summary”).
- Performance-monitoring counters (not shown in Figure 2-1).
- Internal caches and buffers (not shown in Figure 2-1).

Performance-monitoring counters are event counters that can be programmed to count processor events such as the number of instructions decoded, the number of interrupts received, or the number of cache loads.

The processor provides several internal caches and buffers. The caches are used to store both data and instructions. The buffers are used to store things like decoded addresses to system and application segments and write operations waiting to be performed. See also: Chapter 12, “Memory Cache Control.”

### 2.2 Modes of Operation

The IA-32 architecture supports three operating modes and one quasi-operating mode:

- **Protected mode** — This is the native operating mode of the processor. It provides a rich set of architectural features, flexibility, high performance and backward compatibility to existing software base.
- **Real-address mode** — This operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode).
- **System management mode (SMM)** — SMM is a standard architectural feature in all IA-32 processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management
interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.

• **Virtual-8086 mode** — In protected mode, the processor supports a quasi-operating mode known as virtual-8086 mode. This mode allows the processor execute 8086 software in a protected, multitasking environment.

Intel 64 architecture supports all operating modes of IA-32 architecture and IA-32e modes:

• **IA-32e mode** — In IA-32e mode, the processor supports two sub-modes: compatibility mode and 64-bit mode. 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GBytes. Compatibility mode allows most legacy protected-mode applications to run unchanged.

Figure 2-3 shows how the processor moves between operating modes.

The processor is placed in real-address mode following power-up or a reset. The PE flag in control register CR0 then controls whether the processor is operating in real-address or protected mode. See also: Section 10.9, “Mode Switching,” and Section 4.1.2, “Paging-Mode Enabling.”

The VM flag in the EFLAGS register determines whether the processor is operating in protected mode or virtual-8086 mode. Transitions between protected mode and virtual-8086 mode are generally carried out as part of a task switch or a return from an interrupt or exception handler. See also: Section 21.2.5, “Entering Virtual-8086 Mode.”

The LMA bit (IA32_EFER.LMA[bit 10]) determines whether the processor is operating in IA-32e mode. When running in IA-32e mode, 64-bit or compatibility sub-mode operation is determined by CS.L bit of the code segment. The processor enters into IA-32e mode from protected mode by enabling paging and setting the LME bit (IA32_EFER.LME[bit 8]). See also: Chapter 10, “Processor Management and Initialization.”

The processor switches to SMM whenever it receives an SMI while the processor is in real-address, protected, virtual-8086, or IA-32e modes. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.
2.2.1 **Extended Feature Enable Register**

The IA32_EFER MSR provides several fields related to IA-32e mode enabling and operation. It also provides one field that relates to page-access right modification (see Section 4.6, "Access Rights"). The layout of the IA32_EFER MSR is shown in Figure 2-4.

![Figure 2-4. IA32_EFER MSR Layout](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SYSCALL Enable: IA32_EFER.SCE (R/W)</td>
</tr>
<tr>
<td></td>
<td>Enables SYSCALL/SYSRET instructions in 64-bit mode.</td>
</tr>
<tr>
<td>7:1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>8</td>
<td>IA-32e Mode Enable: IA32_EFER.LME (R/W)</td>
</tr>
<tr>
<td></td>
<td>Enables IA-32e mode operation.</td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
</tr>
<tr>
<td>10</td>
<td>IA-32e Mode Active: IA32_EFER.LMA (R)</td>
</tr>
<tr>
<td></td>
<td>Indicates IA-32e mode is active when set.</td>
</tr>
<tr>
<td>11</td>
<td>Execute Disable Bit Enable: IA32_EFER.NXE (R/W)</td>
</tr>
<tr>
<td></td>
<td>Enables page access restriction by preventing instruction fetches from PAE pages with the XD bit set (See Section 4.6).</td>
</tr>
<tr>
<td>63:12</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

2.3 **SYSTEM FLAGS AND FIELDS IN THE EFLAGS REGISTER**

The system flags and IOPL field of the EFLAGS register control I/O, maskable hardware interrupts, debugging, task switching, and the virtual-8086 mode (see Figure 2-5). Only privileged code (typically operating system or executive code) should be allowed to modify these bits.

The system flags and IOPL are:

**TF Trap (bit 8)** — Set to enable single-step mode for debugging; clear to disable single-step mode. In single-step mode, the processor generates a debug exception after each instruction. This allows the execution state of a program to be inspected after each instruction. If an application program sets the TF flag using a
POPF, POPFD, or IRET instruction, a debug exception is generated after the instruction that follows the POPF, POPFD, or IRET.

**IF**  
**Interrupt enable (bit 9)** — Controls the response of the processor to maskable hardware interrupt requests (see also: Section 6.3.2, “Maskable Hardware Interrupts”). The flag is set to respond to maskable hardware interrupts; cleared to inhibit maskable hardware interrupts. The IF flag does not affect the generation of exceptions or nonmaskable interrupts (NMI interrupts). The CPL, IOPL, and the state of the VME flag in control register CR4 determine whether the IF flag can be modified by the CLI, STI, POPF, POPFD, and IRET.

**IOPL**  
**I/O privilege level field (bits 12 and 13)** — Indicates the I/O privilege level (IOPL) of the currently running program or task. The CPL of the currently running program or task must be less than or equal to the IOPL to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.

The IOPL is also one of the mechanisms that controls the modification of the IF flag and the handling of interrupts in virtual-8086 mode when virtual mode extensions are in effect (when CR4.VME = 1). See also: Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

**NT**  
**Nested task (bit 14)** — Controls the chaining of interrupted and called tasks. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. It examines and modifies this flag on returns from a task initiated with the IRET instruction. The flag can be explicitly set or cleared with the POPF/POPFD instructions; however, changing to the state of this flag can generate unexpected exceptions in application programs.

See also: Section 8.4, “Task Linking.”

**RF**  
**Resume (bit 16)** — Controls the processor’s response to instruction-breakpoint conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints (although other exception conditions can cause an exception to be generated). When clear, instruction breakpoints will generate debug exceptions.

The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debug software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with IRETD (to prevent the instruction breakpoint from causing another debug exception). The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.

See also: Section 18.3.1.1, “Instruction-Breakpoint Exception Condition.”

**VM**  
**Virtual-8086 mode (bit 17)** — Set to enable virtual-8086 mode; clear to return to protected mode.
See also: Section 21.2.1, “Enabling Virtual-8086 Mode.”

**AC**  **Alignment check or access control (bit 18)** — If the AM bit is set in the CR0 register, alignment checking of user-mode data accesses is enabled if and only if this flag is 1. An alignment-check exception is generated when reference is made to an unaligned operand, such as a word at an odd byte address or a doubleword at an address which is not an integral multiple of four. Alignment-check exceptions are generated only in user mode (privilege level 3). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate this exception even when caused by instructions executed in user-mode.

The alignment-check exception can be used to check alignment of data. This is useful when exchanging data with processors which require all data to be aligned. The alignment-check exception can also be used by interpreters to flag some pointers as special by misaligning the pointer. This eliminates overhead of checking each pointer and only handles the special pointer when used.

If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1. See Section 4.6, “Access Rights.”

**VIF**  **Virtual Interrupt (bit 19)** — Contains a virtual image of the IF flag. This flag is used in conjunction with the VIP flag. The processor only recognizes the VIF flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. (The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.)

See also: Section 21.3.3.5, “Method 6: Software Interrupt Handling,” and Section 21.4, “Protected-Mode Virtual Interrupts.”

**VIP**  **Virtual interrupt pending (bit 20)** — Set by software to indicate that an interrupt is pending; cleared to indicate that no interrupt is pending. This flag is used in conjunction with the VIF flag. The processor reads this flag but never modifies it. The processor only recognizes the VIP flag when either the VME flag or the PVI flag in control register CR4 is set and the IOPL is less than 3. The VME flag enables the virtual-8086 mode extensions; the PVI flag enables the protected-mode virtual interrupts.

See Section 21.3.3.5, “Method 6: Software Interrupt Handling,” and Section 21.4, “Protected-Mode Virtual Interrupts.”

**ID**  **Identification (bit 21)** — The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

### 2.3.1 System Flags and Fields in IA-32e Mode

In 64-bit mode, the RFLAGS register expands to 64 bits with the upper 32 bits reserved. System flags in RFLAGS (64-bit mode) or EFLAGS (compatibility mode) are shown in Figure 2-5.

In IA-32e mode, the processor does not allow the VM bit to be set because virtual-8086 mode is not supported (attempts to set the bit are ignored). Also, the processor will not set the NT bit. The processor does, however, allow software to set the NT bit (note that an IRET causes a general protection fault in IA-32e mode if the NT bit is set).

In IA-32e mode, the SYSCALL/SYSRET instructions have a programmable method of specifying which bits are cleared in RFLAGS/EFLAGS. These instructions save/restore EFLAGS/RFLAGS.

### 2.4 MEMORY-MANAGEMENT REGISTERS

The processor provides four memory-management registers (GDTR, LDTR, IDTR, and TR) that specify the locations of the data structures which control segmented memory management (see Figure 2-6). Special instructions are provided for loading and storing these registers.
2.4.1 Global Descriptor Table Register (GDTR)

The GDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and the 16-bit table limit for the GDT. The base address specifies the linear address of byte 0 of the GDT; the table limit specifies the number of bytes in the table.

The LGDT and SGDT instructions load and store the GDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. A new base address must be loaded into the GDTR as part of the processor initialization process for protected-mode operation.

See also: Section 3.5.1, “Segment Descriptor Tables.”

2.4.2 Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the LDT. The base address specifies the linear address of byte 0 of the LDT segment; the segment limit specifies the number of bytes in the segment. See also: Section 3.5.1, “Segment Descriptor Tables.”

The LLDT and SLDT instructions load and store the segment selector part of the LDTR register, respectively. The segment that contains the LDT must have a segment descriptor in the GDT. When the LLDT instruction loads a segment selector in the LDTR: the base address, limit, and descriptor attributes from the LDT descriptor are automatically loaded in the LDTR.

When a task switch occurs, the LDTR is automatically loaded with the segment selector and descriptor for the LDT for the new task. The contents of the LDTR are not automatically saved prior to writing the new LDT information into the register.

On power up or reset of the processor, the segment selector and base address are set to the default value of 0 and the limit is set to 0FFFFH.

2.4.3 IDTR Interrupt Descriptor Table Register

The IDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and 16-bit table limit for the IDT. The base address specifies the linear address of byte 0 of the IDT; the table limit specifies the number of bytes in the table. The LIDT and SIDT instructions load and store the IDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. The base address and limit in the register can then be changed as part of the processor initialization process.

See also: Section 6.10, ”Interrupt Descriptor Table (IDT).”
2.4.4 Task Register (TR)

The task register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the TSS of the current task. The selector references the TSS descriptor in the GDT. The base address specifies the linear address of byte 0 of the TSS; the segment limit specifies the number of bytes in the TSS. See also: Section 8.2.4, “Task Register.”

The LTR and STR instructions load and store the segment selector part of the task register, respectively. When the LTR instruction loads a segment selector in the task register, the base address, limit, and descriptor attributes from the TSS descriptor are automatically loaded into the task register. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH.

When a task switch occurs, the task register is automatically loaded with the segment selector and descriptor for the TSS for the new task. The contents of the task register are not automatically saved prior to writing the new TSS information into the register.

2.5 CONTROL REGISTERS

Control registers (CR0, CR1, CR2, CR3, and CR4; see Figure 2-7) determine operating mode of the processor and the characteristics of the currently executing task. These registers are 32 bits in all 32-bit modes and compatibility mode.

In 64-bit mode, control registers are expanded to 64 bits. The MOV CRn instructions are used to manipulate the register bits. Operand-size prefixes for these instructions are ignored. The following is also true:

• The control registers can be read and loaded (or modified) using the move-to-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating-system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.
• Some of the bits in CR0 and CR4 are reserved and must be written with zeros. Attempting to set any reserved bits in CR0[31:0] is ignored. Attempting to set any reserved bits in CR0[63:32] results in a general-protection exception, #GP(0). Attempting to set any reserved bits in CR4 results in a general-protection exception, #GP(0).
• All 64 bits of CR2 are writable by software.
• Reserved bits in CR3[63:MAXPHYADDR] must be zero. Attempting to set any of them results in #GP(0).
• The MOV CR2 instruction does not check that address written to CR2 is canonical.
• A 64-bit capable processor will retain the upper 32 bits of each control register when transitioning out of IA-32e mode.
• On a 64-bit capable processor, an execution of MOV to CR outside of 64-bit mode zeros the upper 32 bits of the control register.
• Register CR8 is available in 64-bit mode only.

The control registers are summarized below, and each architecturally defined control field in these control registers is described individually. In Figure 2-7, the width of the register in 64-bit mode is indicated in parenthesis (except for CR0).

• CR0 — Contains system control flags that control operating mode and states of the processor.
• CR1 — Reserved.
• CR2 — Contains the page-fault linear address (the linear address that caused a page fault).
• CR3 — Contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor’s internal data caches (they do not control TLB caching of page-directory information).

When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table. With 4-level paging and 5-level paging, the CR3 register contains the base address of the PML4.
table and PML5 table, respectively. If PCIDs are enabled, CR3 has a format different from that illustrated in Figure 2-7. See Section 4.5, “4-Level Paging and 5-Level Paging.”

See also: Chapter 4, “Paging.”

- **CR4** — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. Bits CR4[63:32] can only be used for IA-32e mode only features that are enabled after entering 64-bit mode. Bits CR4[63:32] do not have any effect outside of IA-32e mode.

- **CR8** — Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.
The flags in control registers are:

CR0.PG

**Paging (bit 31 of CR0)** — Enables paging when set; disables paging when clear. When paging is disabled, all linear addresses are treated as physical addresses. The PG flag has no effect if the PE flag (bit 0 of register CR0) is not also set; setting the PG flag when the PE flag is clear causes a general-protection exception (#GP). See also: Chapter 4, “Paging.”

On Intel 64 processors, enabling and disabling IA-32e mode operation also requires modifying CR0.PG.

CR0.CD

**Cache Disable (bit 30 of CR0)** — When the CD and NW flags are clear, caching of memory locations for the whole of physical memory in the processor’s internal (and external) caches is enabled. When the CD flag is set, caching is restricted as described in Table 12-5. To prevent the processor from accessing and updating its caches, the CD flag must be set and the caches must be invalidated so that no cache hits can occur.

See also: Section 12.5.3, “Preventing Caching,” and Section 12.5, “Cache Control.”

CR0.NW

**Not Write-through (bit 29 of CR0)** — When the NW and CD flags are clear, write-back (for Pentium 4, Intel Xeon, P6 family, and Pentium processors) or write-through (for Intel486 processors) is enabled for writes that hit the cache and invalidation cycles are enabled. See Table 12-5 for detailed information about the effect of the NW flag on caching for other settings of the CD and NW flags.

CR0.AM

**Alignment Mask (bit 18 of CR0)** — Enables automatic alignment checking when set; disables alignment checking when clear. Alignment checking is performed only when the AM flag is set, the AC flag in the EFLAGS register is set, CPL is 3, and the processor is operating in either protected or virtual-8086 mode.

CR0.WP

**Write Protect (bit 16 of CR0)** — When set, inhibits supervisor-level procedures from writing into read-only pages; when clear, allows supervisor-level procedures to write into read-only pages (regardless of the U/S bit setting; see Section 4.1.3 and Section 4.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX. This flag must be set before software can set CR4.CET, and it cannot be cleared as long as CR4.CET = 1 (see below).

CR0.NE

**Numeric Error (bit 5 of CR0)** — Enables the native (internal) mechanism for reporting x87 FPU errors when set; enables the PC-style x87 FPU error reporting mechanism when clear. When the NE flag is clear and the IGNNE# input is asserted, x87 FPU errors are ignored. When the NE flag is clear and the IGNNE# input is deasserted, an unmasked x87 FPU error causes the processor to assert the FERR# pin to generate an external interrupt and to stop instruction execution immediately before executing the next waiting floating-point instruction or WAIT/FWAIT instruction.

The FERR# pin is intended to drive an input to an external interrupt controller (the FERR# pin emulates the ERROR# pin of the Intel 287 and Intel 387 DX math coprocessors). The NE flag, IGNNE# pin, and FERR# pin are used with external logic to implement PC-style error reporting. Using FERR# and IGNNE# to handle floating-point exceptions is deprecated by modern operating systems; this non-native approach also limits newer processors to operate with one logical processor active.


CR0.ET

**Extension Type (bit 4 of CR0)** — Reserved in the Pentium 4, Intel Xeon, P6 family, and Pentium processors. In the Pentium 4, Intel Xeon, and P6 family processors, this flag is hardcoded to 1. In the Intel386 and Intel486 processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.

CR0.TS

**Task Switched (bit 3 of CR0)** — Allows the saving of the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction is
actually executed by the new task. The processor sets this flag on every task switch and tests it when executing x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

- If the TS flag is set and the EM flag (bit 2 of CR0) is clear, a device-not-available exception (#NM) is raised prior to the execution of any x87 FPU/MMX/SSE2/SSE3/SSSE3/SSE4 instruction; with the exception of PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. See the paragraph below for the special case of the WAIT/FWAIT instructions.

- If the TS flag is set and the MP flag (bit 1 of CR0) and EM flag are clear, an #NM exception is not raised prior to the execution of an x87 FPU WAIT/FWAIT instruction.

- If the EM flag is set, the setting of the TS flag has no effect on the execution of x87 FPU/MMX/SSE2/SSE3/SSSE3/SSE4 instructions.

Table 2-2 shows the actions taken when the processor encounters an x87 FPU instruction based on the settings of the TS, EM, and MP flags. Table 13-1 and 14-1 show the actions taken when the processor encounters an MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction.

The processor does not automatically save the context of the x87 FPU, XMM, and MXCSR registers on a task switch. Instead, it sets the TS flag, which causes the processor to raise an #NM exception whenever it encounters an x87 FPU/MMX/SSE2/SSE3/SSSE3/SSE4 instruction in the instruction stream for the new task (with the exception of the instructions listed above).

The fault handler for the #NM exception can then be used to clear the TS flag (with the CLTS instruction) and save the context of the x87 FPU, XMM, and MXCSR registers. If the task never encounters an x87 FPU/MMX/SSE2/SSE3/SSSE3/SSE4 instruction, the x87 FPU/MMX/SSE2/SSE3/SSSE3/SSE4 context is never saved.

**Table 2-2. Action Taken By x87 FPU Instructions for Different Combinations of EM, MP, and TS**

<table>
<thead>
<tr>
<th>CR0 Flags</th>
<th>x87 FPU Instruction Type</th>
<th>WAIT/FWAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM MP TS</td>
<td>Floating-Point</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>0 0 1</td>
<td># NM Exception</td>
<td>Execute</td>
</tr>
<tr>
<td>0 1 0</td>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>0 1 1</td>
<td># NM Exception</td>
<td># NM exception</td>
</tr>
<tr>
<td>1 0 0</td>
<td># NM Exception</td>
<td>Execute</td>
</tr>
<tr>
<td>1 0 1</td>
<td># NM Exception</td>
<td>Execute</td>
</tr>
<tr>
<td>1 1 0</td>
<td># NM Exception</td>
<td>Execute</td>
</tr>
<tr>
<td>1 1 1</td>
<td># NM Exception</td>
<td># NM exception</td>
</tr>
</tbody>
</table>

**CR0.EM Emulation (bit 2 of CR0)** — Indicates that the processor does not have an internal or external x87 FPU when set; indicates an x87 FPU is present when clear. This flag also affects the execution of MMX/SSE2/SSE3/SSSE3/SSE4 instructions.

When the EM flag is set, execution of an x87 FPU instruction generates a device-not-available exception (#NM). This flag must be set when the processor does not have an internal x87 FPU or is not connected to an external math coprocessor. Setting this flag forces all floating-point instructions to be handled by software emulation. Table 10-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the EM, MP, and TS flags.

Also, when the EM flag is set, execution of an MMX instruction causes an invalid-opcode exception (#UD) to be generated (see Table 13-1). Thus, if an IA-32 or Intel 64 processor incorporates MMX technology, the EM flag must be set to 0 to enable execution of MMX instructions.

Similarly for SSE/SSE2/SSE3/SSSE3/SSE4 extensions, when the EM flag is set, execution of most SSE/SSE2/SSE3/SSSE3/SSE4 instructions causes an invalid opcode exception (#UD) to be generated (see...
If an IA-32 or Intel 64 processor incorporates the SSE/SSE2/SSE3/SSSE3/SSE4 extensions, the EM flag must be set to 0 to enable execution of these extensions. SSE/SSE2/SSE3/SSSE3/SSE4 instructions not affected by the EM flag include: PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

**Monitor Coprocessor (bit 1 of CR0)** — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CR0). If the MP flag is set, a WAIT instruction generates a device-not-available exception (#NM) if the TS flag is also set. If the MP flag is clear, the WAIT instruction ignores the setting of the TS flag. Table 10-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the MP, EM, and TS flags.

**Protection Enable (bit 0 of CR0)** — Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set.

See also: Section 10.9, “Mode Switching.”

**Page-level Cache Disable (bit 4 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, “Paging and Memory Typing.” This bit is not used if paging is disabled, with PAE paging, or with 4-level paging or 5-level paging if CR4.PCIDE=1.

**Page-level Write-Through (bit 3 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, “Paging and Memory Typing.” This bit is not used if paging is disabled, with PAE paging, or with 4-level paging or 5-level paging if CR4.PCIDE=1.

**Virtual-8086 Mode Extensions (bit 0 of CR4)** — Enables interrupt- and exception-handling extensions in virtual-8086 mode when set; disables the extensions when clear. Use of the virtual mode extensions can improve the performance of virtual-8086 applications by eliminating the overhead of calling the virtual-8086 monitor to handle interrupts and exceptions that occur while executing an 8086 program and, instead, redirecting the interrupts and exceptions back to the 8086 program’s handlers. It also provides hardware support for a virtual interrupt flag (VIF) to improve reliability of running 8086 programs in multi-tasking and multiple-processor environments.

See also: Section 21.3, “Interrupt and Exception Handling in Virtual-8086 Mode.”

**Protected-Mode Virtual Interrupts (bit 1 of CR4)** — Enables hardware support for a virtual interrupt flag (VIF) in protected mode when set; disables the VIF flag in protected mode when clear.

See also: Section 21.4, “Protected-Mode Virtual Interrupts.”

**Time Stamp Disable (bit 2 of CR4)** — Restricts the execution of the RDTSC instruction to procedures running at privilege level 0 when set; allows RDTSC instruction to be executed at any privilege level when clear. This bit also applies to the RDTSCP instruction if supported (if CPUID.80000001H:EDX[27] = 1).

**Debugging Extensions (bit 3 of CR4)** — References to debug registers DR4 and DR5 cause an undefined opcode (#UD) exception to be generated when set; when clear, processor aliases references to registers DR4 and DR5 for compatibility with software written to run on earlier IA-32 processors.

See also: Section 18.2.2, “Debug Registers DR4 and DR5.”

**Page Size Extensions (bit 4 of CR4)** — Enables 4-MByte pages with 32-bit paging when set; restricts 32-bit paging to pages of 4 KBytes when clear.

See also: Section 4.3, “32-Bit Paging.”

---

2. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.
CR4.PAE

**Physical Address Extension (bit 5 of CR4)** — When set, enables paging to produce physical addresses with more than 32 bits. When clear, restricts physical addresses to 32 bits. PAE must be set before entering IA-32e mode.

See also: Chapter 4, “Paging.”

CR4.MCE

**Machine-Check Enable (bit 6 of CR4)** — Enables the machine-check exception when set; disables the machine-check exception when clear.

See also: Chapter 16, “Machine-Check Architecture.”

CR4.PGE

**Page Global Enable (bit 7 of CR4)** — (Introduced in the P6 family processors.) Enables the global page feature when set; disables the global page feature when clear. The global page feature allows frequently used or shared pages to be marked as global to all users (done with the global flag, bit 8, in a page-directory-pointer-table entry, a page-directory entry, or a page-table entry). Global pages are not flushed from the translation-lookaside buffer (TLB) on a task switch or a write to register CR3.

When enabling the global page feature, paging must be enabled (by setting the PG flag in control register CR0) before the PGE flag is set. Reversing this sequence may affect program correctness, and processor performance will be impacted.

See also: Section 4.10, “Caching Translation Information.”

CR4.PCE

**Performance-Monitoring Counter Enable (bit 8 of CR4)** — Enables execution of the RDPMC instruction for programs or procedures running at any protection level when set; RDPMC instruction can be executed only at protection level 0 when clear.

CR4.OSFXSR

**Operating System Support for FXSAVE and FXRSTOR instructions (bit 9 of CR4)** — When set, this flag: (1) indicates to software that the operating system supports the use of the FXSAVE and FXRSTOR instructions, (2) enables the FXSAVE and FXRSTOR instructions to save and restore the contents of the XMM and MXCSR registers along with the contents of the x87 FPU and MMX registers, and (3) enables the processor to execute SSE/SSE2/SSE3/SSSE3/SSE4 instructions, with the exception of the PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

If this flag is clear, the FXSAVE and FXRSTOR instructions will save and restore the contents of the x87 FPU and MMX registers, but they may not save and restore the contents of the XMM and MXCSR registers. Also, the processor will generate an invalid opcode exception (#UD) if it attempts to execute any SSE/SSE2/SSE3 instruction, with the exception of PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. The operating system or executive must explicitly set this flag.

**NOTE**

CPUID feature flag FXSR indicates availability of the FXSAVE/FXRSTOR instructions. The OSFXSR bit provides operating system software with a means of enabling FXSAVE/FXRSTOR to save/restore the contents of the X87 FPU, XMM, and MXCSR registers. Consequently OSFXSR bit indicates that the operating system provides context switch support for SSE/SSE2/SSE3/SSSE3/SSE4.

CR4.OSXMMEXCPT

**Operating System Support for Unmasked SIMD Floating-Point Exceptions (bit 10 of CR4)** — When set, indicates that the operating system supports the handling of unmasked SIMD floating-point exceptions through an exception handler that is invoked when a SIMD floating-point exception (#XM) is generated. SIMD floating-point exceptions are only generated by SSE/SSE2/SSE3/SSE4.1 SIMD floating-point instructions.

The operating system or executive must explicitly set this flag. If this flag is not set, the processor will generate an invalid opcode exception (#UD) whenever it detects an unmasked SIMD floating-point exception.
CR4.UMIP
User-Mode Instruction Prevention (bit 11 of CR4) — When set, the following instructions cannot be executed if CPL > 0: SGDT, SIDT, SLDT, SMSW, and STR. An attempt at such execution causes a general-protection exception (#GP).

CR4.LA57
57-bit linear addresses (bit 12 of CR4) — When set in IA-32e mode, the processor uses 5-level paging to translate 57-bit linear addresses. When clear in IA-32e mode, the processor uses 4-level paging to translate 48-bit linear addresses. This bit cannot be modified in IA-32e mode.

See also: Chapter 4, “Paging.”

CR4.VMXE
VMX-Enable Bit (bit 13 of CR4) — Enables VMX operation when set. See Chapter 24, “Introduction to Virtual Machine Extensions.”

CR4.SMXE

CR4.FSGSBASE
FSGSBASE-Enable Bit (bit 16 of CR4) — Enables the instructions RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE.

CR4.PCIDE
PCID-Enable Bit (bit 17 of CR4) — Enables process-context identifiers (PCIDs) when set. See Section 4.10.1, “Process-Context Identifiers (PCIDs).” Applies only in IA-32e mode (if IA32_EFER.LMA = 1).

CR4.OSXSAVE
XSAXE and Processor Extended States-Enable Bit (bit 18 of CR4) — When set, this flag: (1) indicates (via CPUID.01H:ECX.OSXSAVE[bit 27]) that the operating system supports the use of the XGETBV, XSAVE, and XRSTOR instructions by general software; (2) enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), the SSE state (XMM registers and MXCSR), along with other processor extended states enabled in XCR0; (3) enables the processor to execute XGETBV and XSETBV instructions in order to read and write XCR0. See Section 2.6 and Chapter 14, “System Programming for Instruction Set Extensions and Processor Extended States.”

CR4.KL
Key-Locker-Enable Bit (bit 19 of CR4) — When set, the LOADIWKEY instruction is enabled; in addition, if support for the AES Key Locker instructions has been activated by system firmware, CPUID.19H:EBX.AESKLE[bit 0] is enumerated as 1 and the AES Key Locker instructions are enabled. When clear, CPUID.19H:EBX.AESKLE[bit 0] is enumerated as 0 and execution of any Key Locker instruction causes an invalid-opcode exception (#UD).

CR4.SMEP
SMEP-Enable Bit (bit 20 of CR4) — Enables supervisor-mode execution prevention (SMEP) when set. See Section 4.6, “Access Rights.”

CR4.SMAP
SMAP-Enable Bit (bit 21 of CR4) — Enables supervisor-mode access prevention (SMAP) when set. See Section 4.6, “Access Rights.”

CR4.PKE
Enable protection keys for user-mode pages (bit 22 of CR4) — 4-level paging and 5-level paging associate each user-mode linear address with a protection key. When set, this flag indicates (via CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4]) that the operating system supports use of the PKRU register to specify, for each protection key, whether user-mode linear addresses with that protection key can be read or written. This bit also enables access to the PKRU register using the RDPKRU and WRPKRU instructions.

3. Software can check CPUID.19H:EBX.AESKLE[bit 0] after setting CR4.KL to determine whether the AES Key Locker instructions have been enabled. Note that some processors may allow enabling of those instructions without activation by system firmware. Some processors may not support use of the AES Key Locker instructions in system-management mode (SMM). Those processors enumerate CPUID.19H:EBX.AESKLE[bit 0] as 0 in SMM regardless of the setting of CR4.KL.
CR4.CET

**Control-flow Enforcement Technology (bit 23 of CR4)** — Enables control-flow enforcement technology when set. See Chapter 17, “Control-flow Enforcement Technology (CET),” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 1*. This flag can be set only if CR0.WP is set, and it must be clear before CR0.WP can be cleared (see below).

CR4.PKS

**Enable protection keys for supervisor-mode pages (bit 24 of CR4)** — 4-level paging and 5-level paging associate each supervisor-mode linear address with a protection key. When set, this flag allows use of the IA32_PKRS MSR to specify, for each protection key, whether supervisor-mode linear addresses with that protection key can be read or written.

CR4.UINTR

**User Interrupts Enable Bit (bit 25 of CR4)** — Enables user interrupts when set, including user-interrupt delivery, user-interrupt notification identification, and the user-interrupt instructions.

CR8.TPL

**Task Priority Level (bit 3:0 of CR8)** — This sets the threshold value corresponding to the highest-priority interrupt to be blocked. A value of 0 means all interrupts are enabled. This field is available in 64-bit mode. A value of 15 means all interrupts will be disabled.

### 2.5.1 CPUID Qualification of Control Register Flags

Not all flags in control register CR4 are implemented on all processors. With the exception of the PCE flag, they can be qualified with the CPUID instruction to determine if they are implemented on the processor before they are used.

The CR8 register is available on processors that support Intel 64 architecture.

### 2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0)

If CPUID.01H:ECX.XSAVE[bit 26] is 1, the processor supports one or more **extended control registers** (XCRs). Currently, the only such register defined is XCR0. This register specifies the set of processor state components for which the operating system provides context management, e.g., x87 FPU state, SSE state, AVX state. The OS programs XCR0 to reflect the features for which it provides context management.
Software can access XCR0 only if CR4.OSXSAVE[bit 18] = 1. (This bit is also readable as CPUID.01H:ECX.OSXSAVE[bit 27].) Software can use CPUID leaf function 0DH to enumerate the bits in XCR0 that the processor supports (see CPUID instruction in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). Each supported state component is represented by a bit in XCR0. System software enables state components by loading an appropriate bit mask value into XCR0 using the XSETBV instruction.

As each bit in XCR0 (except bit 63) corresponds to a processor state component, XCR0 thus provides support for up to 63 sets of processor state components. Bit 63 of XCR0 is reserved for future expansion and will not represent a processor state component.

Currently, XCR0 defines support for the following state components:

- XCR0.X87 (bit 0): This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.
- XCR0.SSE (bit 1): If 1, the XSAVE feature set can be used to manage MXCSR and the XMM registers (XMM0-XMM15 in 64-bit mode; otherwise XMM0-XMM7).
- XCR0.AVX (bit 2): If 1, Intel AVX instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the YMM registers (YMM0-YMM15 in 64-bit mode; otherwise YMM0-YMM7).
- XCR0.BNDREG (bit 3): If 1, Intel MPX instructions can be executed and the XSAVE feature set can be used to manage the bounds registers BND0–BND3.
- XCR0.BNDCSR (bit 4): If 1, Intel MPX instructions can be executed and the XSAVE feature set can be used to manage the BNDCFGU and BNDSTATUS registers.
- XCR0.opmask (bit 5): If 1, Intel AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the opmask registers k0–k7.
- XCR0.ZMM_Hi256 (bit 6): If 1, Intel AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper halves of the lower ZMM registers (ZMM0-ZMM15 in 64-bit mode; otherwise ZMM0-ZMM7).
- XCR0.Hi16_ZMM (bit 7): If 1, Intel AVX-512 instructions can be executed and the XSAVE feature set can be used to manage the upper ZMM registers (ZMM16-ZMM31, only in 64-bit mode).
- XCR0.PKRU (bit 9): If 1, the XSAVE feature set can be used to manage the PKRU register (see Section 2.7).
- XCR0.TILECFG (bit 17): If 1, and if XCR0.TILEDATA is also 1, Intel AMX instructions can be executed and the XSAVE feature set can be used to manage TILECFG.
SYSTEM ARCHITECTURE OVERVIEW

- XCR0.TILEDATA (bit 18): If 1, and if XCR0.TILECFG is also 1, Intel AMX instructions can be executed and the XSAVE feature set can be used to manage TILEDATA.

An attempt to use XSETBV to write to XCR0 results in general-protection exceptions (#GP) if it would do any of the following:

- Set a bit reserved in XCR0 for a given processor (as determined by the contents of EAX and EDX after executing CPUID with EAX=0DH, ECX= 0H).
- Clear XCR0.x87.
- Clear XCR0.SSE and set XCR0.AVX.
- Clear XCR0.AVX and set any of XCR0.opmask, XCR0.ZMM_Hi256, or XCR0.Hi16_ZMM.
- Set either XCR0.BNDREG or XCR0.BNDCSR while not setting the other.
- Set any of XCR0.opmask, XCR0.ZMM_Hi256, and XCR0.Hi16_ZMM while not setting all of them.
- Set either XCR0.TILECFG or XCR0.TILEDATA while not setting the other.

After reset, all bits (except bit 0) in XCR0 are cleared to zero; XCR0[0] is set to 1.

2.7 PROTECTION-KEY RIGHTS REGISTERS (PKRU AND IA32_PKRS)

Processors may support either or both of two protection-key rights registers: PKRU for user-mode pages and the IA32_PKRS MSR (MSR index 6E1H) for supervisor-mode pages. 4-level paging and 5-level paging associate a 4-bit protection key with each page. The protection-key rights registers determine accessibility based on a page’s protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, the processor supports the protection-key feature for user-mode pages. When CR4.PKE = 1, software can use the protection-key rights register for user pages (PKRU) to specify the access rights for user-mode pages for each protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, the processor supports the protection-key feature for supervisor-mode pages. When CR4.PKS = 1, software can use the protection-key rights register for supervisor pages (the IA32_PKRS MSR) to specify the access rights for supervisor-mode pages for each protection key.

![Figure 2-9. Format of Protection-Key Rights Registers](image)

The format of each protection-key rights register is given in Figure 2-9. Each contains 16 pairs of disable controls to prevent data accesses to linear addresses (user-mode or supervisor-mode, depending on the register) based on their protection keys. Each protection key \( i \) (0 ≤ \( i \) ≤ 15) is associated with two bits in each protection-key rights register:

- Bit 2\( i \), shown as “AD\( i \)” (access disable): if set, the processor prevents any data accesses to linear addresses (user-mode or supervisor-mode, depending on the register) with protection key \( i \).
- Bit 2\( i+1 \), shown as “WD\( i \)” (write disable): if set, the processor prevents write accesses to linear addresses (user-mode or supervisor-mode, depending on the register) with protection key \( i \).

(Bits 63:32 of the IA32_PKRS MSR are reserved and must be zero.)

See Section 4.6.2, “Protection Keys,” for details of how the processor uses the protection-key rights registers to control accesses to linear addresses.

Software can read and write PKRU using the RDPKRU and WRPKRU instructions. The IA32_PKRS MSR can be read and written with the RDMSR and WRMSR instructions. Writes to the IA32_PKRS MSR using WRMSR are not serializing.
2.8 SYSTEM INSTRUCTION SUMMARY

System instructions handle system-level functions such as loading system registers, managing the cache, managing interrupts, or setting up the debug registers. Many of these instructions can be executed only by operating-system or executive procedures (that is, procedures running at privilege level 0). Others can be executed at any privilege level and are thus available to application programs.

Table 2-3 lists the system instructions and indicates whether they are available and useful for application programs. These instructions are described in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Use useful to Application?</th>
<th>Protected from Application?</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLDT</td>
<td>Load LDT Register</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SLDT</td>
<td>Store LDT Register</td>
<td>No</td>
<td>If CR4.UMIP = 1</td>
</tr>
<tr>
<td>LGDT</td>
<td>Load GDT Register</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SGDT</td>
<td>Store GDT Register</td>
<td>No</td>
<td>If CR4.UMIP = 1</td>
</tr>
<tr>
<td>LTR</td>
<td>Load Task Register</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>STR</td>
<td>Store Task Register</td>
<td>No</td>
<td>If CR4.UMIP = 1</td>
</tr>
<tr>
<td>LIDT</td>
<td>Load IDT Register</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SIDT</td>
<td>Store IDT Register</td>
<td>No</td>
<td>If CR4.UMIP = 1</td>
</tr>
<tr>
<td>MOV CRn</td>
<td>Load and store control registers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SMSW</td>
<td>Store MSW</td>
<td>Yes</td>
<td>If CR4.UMIP = 1</td>
</tr>
<tr>
<td>LMSW</td>
<td>Load MSW</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CLTS</td>
<td>Clear TS flag in CR0</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ARPL</td>
<td>Adjust RPL</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LAR</td>
<td>Load Access Rights</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LSL</td>
<td>Load Segment Limit</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VERR</td>
<td>Verify for Reading</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VERW</td>
<td>Verify for Writing</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MOV DRn</td>
<td>Load and store debug registers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>INVD</td>
<td>Invalidate cache, no writeback</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>WBINVD</td>
<td>Invalidate cache, with writeback</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>INVLPG</td>
<td>Invalidate TLB entry</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>HLT</td>
<td>Halt Processor</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCK (Prefix)</td>
<td>Bus Lock</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>RSM</td>
<td>Return from system management mode</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RDMSR³</td>
<td>Read Model-Specific Registers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>WRMSR³</td>
<td>Write Model-Specific Registers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RDPMC⁴</td>
<td>Read Performance-Monitoring Counter</td>
<td>Yes</td>
<td>Yes²</td>
</tr>
<tr>
<td>RDTSC³</td>
<td>Read Time-Stamp Counter</td>
<td>Yes</td>
<td>Yes²</td>
</tr>
<tr>
<td>RDTSCP⁷</td>
<td>Read Serialized Time-Stamp Counter</td>
<td>Yes</td>
<td>Yes²</td>
</tr>
<tr>
<td>XGETBV</td>
<td>Return the state of XCRO</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>XSETBV</td>
<td>Enable one or more processor extended states</td>
<td>No⁶</td>
<td>Yes</td>
</tr>
</tbody>
</table>
2.8.1 Loading and Storing System Registers

The GDTR, LDTR, IDTR, and TR registers each have a load and store instruction for loading data into and storing data from the register:

- **LGDT (Load GDTR Register)** — Loads the GDT base address and limit from memory into the GDTR register.
- **SGDT (Store GDTR Register)** — Stores the GDT base address and limit from the GDTR register into memory.
- **LIDT (Load IDTR Register)** — Loads the IDT base address and limit from memory into the IDTR register.
- **SIDT (Store IDTR Register)** — Stores the IDT base address and limit from the IDTR register into memory.
- **LLDT (Load LDTR Register)** — Loads the LDT segment selector and segment descriptor from memory into the LDTR. (The segment selector operand can also be located in a general-purpose register.)
- **SLDT (Store LDTR Register)** — Stores the LDT segment selector from the LDTR register into memory or a general-purpose register.
- **LTR (Load Task Register)** — Loads segment selector and segment descriptor for a TSS from memory into the task register. (The segment selector operand can also be located in a general-purpose register.)
- **STR (Store Task Register)** — Stores the segment selector for the current task TSS from the task register into memory or a general-purpose register.

The LMSW (load machine status word) and SMSW (store machine status word) instructions operate on bits 0 through 15 of control register CR0. These instructions are provided for compatibility with the 16-bit Intel 286 processor. Programs written to run on 32-bit IA-32 processors should not use these instructions. Instead, they should access the control register CR0 using the MOV CR instruction.

The CLTS (clear TS flag in CR0) instruction is provided for use in handling a device-not-available exception (#NM) that occurs when the processor attempts to execute a floating-point instruction when the TS flag is set. This instruction allows the TS flag to be cleared after the x87 FPU context has been saved, preventing further #NM exceptions. See Section 2.5, “Control Registers,” for more information on the TS flag.

The control registers (CR0, CR1, CR2, CR3, CR4, and CR8) are loaded using the MOV instruction. The instruction loads a control register from a general-purpose register or stores the content of a control register in a general-purpose register.

2.8.2 Verifying Access Privileges

The processor provides several instructions for examining segment selectors and segment descriptors to determine if access to their associated segments is allowed. These instructions duplicate some of the automatic access rights and type checking done by the processor, thus allowing operating-system or executive software to prevent exceptions from being generated.

The ARPL (adjust RPL) instruction adjusts the RPL (requestor privilege level) of a segment selector to match that of the program or procedure that supplied the segment selector. See Section 5.10.4, “Checking Caller Access Privileges (ARPL Instruction),” for a detailed explanation of the function and use of this instruction. Note that ARPL is not supported in 64-bit mode.
The LAR (load access rights) instruction verifies the accessibility of a specified segment and loads access rights information from the segment’s segment descriptor into a general-purpose register. Software can then examine the access rights to determine if the segment type is compatible with its intended use. See Section 5.10.1, “Checking Access Rights (LAR Instruction),” for a detailed explanation of the function and use of this instruction.

The LSL (load segment limit) instruction verifies the accessibility of a specified segment and loads the segment limit from the segment’s segment descriptor into a general-purpose register. Software can then compare the segment limit with an offset into the segment to determine whether the offset lies within the segment. See Section 5.10.3, “Checking That the Pointer Offset Is Within Limits (LSL Instruction),” for a detailed explanation of the function and use of this instruction.

The VERR (verify for reading) and VERW (verify for writing) instructions verify if a selected segment is readable or writable, respectively, at a given CPL. See Section 5.10.2, “Checking Read/Write Rights (VERR and VERW Instructions),” for a detailed explanation of the function and use of these instructions.

2.8.3 Loading and Storing Debug Registers

Internal debugging facilities in the processor are controlled by a set of 8 debug registers (DR0-DR7). The MOV instruction allows setup data to be loaded to and stored from these registers.

On processors that support Intel 64 architecture, debug registers DR0-DR7 are 64 bits. In 32-bit modes and compatibility mode, writes to a debug register fill the upper 32 bits with zeros. Reads return the lower 32 bits. In 64-bit mode, the upper 32 bits of DR6-DR7 are reserved and must be written with zeros. Writing one to any of the upper 32 bits causes an exception, #GP(0).

In 64-bit mode, MOV DRn instructions read or write all 64 bits of a debug register (operand-size prefixes are ignored). All 64 bits of DR0-DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0-DR3 are in the limits of the implementation. Address matching is supported only on valid addresses generated by the processor implementation.

2.8.4 Invalidating Caches and TLBs

The processor provides several instructions for use in explicitly invalidating its caches and TLB entries. The INVD (invalidate cache with no writeback) instruction invalidates all data and instruction entries in the internal caches and sends a signal to the external caches indicating that they should also be invalidated.

The WBINVD (invalidate cache with writeback) instruction performs the same function as the INVD instruction, except that it writes back modified lines in its internal caches to memory before it invalidates the caches. After invalidating the caches local to the executing logical processor or processor core, WBINVD signals caches higher in the cache hierarchy (caches shared with the invalidating logical processor or core) to write back any data they have in modified state at the time of instruction execution and to invalidate their contents.

Note, non-shared caches may not be written back nor invalidated. In Figure 2-10 below, if code executing on either LP0 or LP1 were to execute a WBINVD, the shared L1 and L2 for LP0/LP1 will be written back and invalidated as will the shared L3. However, the L1 and L2 caches not shared with LP0 and LP1 will not be written back nor invalidated.
The INVLPG (invalidate TLB entry) instruction invalidates (flushes) the TLB entry for a specified page.

2.8.5 **Controlling the Processor**

The HLT (halt processor) instruction stops the processor until an enabled interrupt (such as NMI or SMI, which are normally enabled), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal is received. The processor generates a special bus cycle to indicate that the halt mode has been entered.

Hardware may respond to this signal in a number of ways. An indicator light on the front panel may be turned on. An NMI interrupt for recording diagnostic information may be generated. Reset initialization may be invoked (note that the BINIT# pin was introduced with the Pentium Pro processor). If any non-wake events are pending during shutdown, they will be handled after the wake event from shutdown is processed (for example, A20M# interrupts).

The LOCK prefix invokes a locked (atomic) read-modify-write operation when modifying a memory operand. This mechanism is used to allow reliable communications between processors in multiprocessor systems, as described below:

- In the Pentium processor and earlier IA-32 processors, the LOCK prefix causes the processor to assert the LOCK# signal during the instruction. This always causes an explicit bus lock to occur.
- In the Pentium 4, Intel Xeon, and P6 family processors, the locking operation is handled with either a cache lock or bus lock. If a memory access is cacheable and affects only a single cache line, a cache lock is invoked and the system bus and the actual memory location in system memory are not locked during the operation. Here, other Pentium 4, Intel Xeon, or P6 family processors on the bus write-back any modified data and invalidate their caches as necessary to maintain system memory coherency. If the memory access is not cacheable and/or it crosses a cache line boundary, the processor's LOCK# signal is asserted and the processor does not respond to requests for bus control during the locked operation.

The RSM (return from SMM) instruction restores the processor (from a context dump) to the state it was in prior to a system management mode (SMM) interrupt.

2.8.6 **Reading Performance-Monitoring and Time-Stamp Counters**

The RDPMC (read performance-monitoring counter) and RDTSC (read time-stamp counter) instructions allow application programs to read the processor’s performance-monitoring and time-stamp counters, respectively. Processors based on Intel NetBurst® microarchitecture have eighteen 40-bit performance-monitoring counters; P6 family processors have two 40-bit counters. Intel Atom® processors and most of the processors based on the Intel Core microarchitecture support two types of performance monitoring counters: programmable performance counters similar to those available in the P6 family, and three fixed-function performance monitoring counters. Details
of programmable and fixed-function performance monitoring counters for each processor generation are described in Chapter 20, “Performance Monitoring.”

The programmable performance counters can support counting either the occurrence or duration of events. Events that can be monitored on programmable counters generally are model specific (except for architectural performance events enumerated by CPUID leaf 0AH); they may include the number of instructions decoded, interrupts received, or the number of cache loads. Individual counters can be set up to monitor different events. Use the system instruction WRMSR to set up values in one of the IA32_PERFEVTSELx MSR, in one of the 45 ESCRs and one of the 18 CCCR MSRs (for Pentium 4 and Intel Xeon processors); or in the PerfEvtSel0 or the PerfEvtSel1 MSR (for the P6 family processors). The RDPMC instruction loads the current count from the selected counter into the EDX:EAX registers.

Fixed-function performance counters record only specific events that are defined at: https://perfmonevents.intel.com/, and the width/number of fixed-function counters are enumerated by CPUID leaf 0AH.

The time-stamp counter is a model-specific 64-bit counter that is reset to zero each time the processor is reset. If not reset, the counter will increment ~9.5 x 10^16 times per year when the processor is operating at a clock rate of 3GHz. At this clock frequency, it would take over 190 years for the counter to wrap around. The RDTSC instruction loads the current count of the time-stamp counter into the EDX:EAX registers.


The RDTSC instruction was introduced into the IA-32 architecture with the Pentium processor. The RDPMC instruction was introduced into the IA-32 architecture with the Pentium Pro processor and the Pentium processor with MMX technology. Earlier Pentium processors have two performance-monitoring counters, but they can be read only with the RDMSR instruction, and only at privilege level 0.

2.8.6.1 Reading Counters in 64-Bit Mode

In 64-bit mode, RDTSC operates the same as in protected mode. The count in the time-stamp counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

RDPMC requires an index to specify the offset of the performance-monitoring counter. In 64-bit mode for Pentium 4 or Intel Xeon processor families, the index is specified in ECX[30:0]. The current count of the performance-monitoring counter is stored in EDX:EAX (or RDX[31:0]:RAX[31:0] with RDX[63:32]:RAX[63:32] cleared).

2.8.7 Reading and Writing Model-Specific Registers

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions allow a processor’s 64-bit model-specific registers (MSRs) to be read and written, respectively. The MSR to be read or written is specified by the value in the ECX register.

RDMSR reads the value from the specified MSR to the EDX:EAX registers; WRMSR writes the value in the EDX:EAX registers to the specified MSR. RDMSR and WRMSR were introduced into the IA-32 architecture with the Pentium processor.

See Section 10.4, “Model-Specific Registers (MSRs),” for more information.

2.8.7.1 Reading and Writing Model-Specific Registers in 64-Bit Mode

RDMSR and WRMSR require an index to specify the address of an MSR. In 64-bit mode, the index is 32 bits; it is specified using ECX.

2.8.8 Enabling Processor Extended States

The XSETBV instruction is required to enable OS support of individual processor extended states in XCR0 (see Section 2.6).
This chapter describes the Intel 64 and IA-32 architecture’s protected-mode memory management facilities, including the physical memory requirements, segmentation mechanism, and paging mechanism.

See also: Chapter 5, “Protection,” (for a description of the processor’s protection mechanism) and Chapter 21, “8086 Emulation,” (for a description of memory addressing protection in real-address and virtual-8086 modes).

3.1 MEMORY MANAGEMENT OVERVIEW

The memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another. Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program’s execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks. When operating in protected mode, some form of segmentation must be used. **There is no mode bit to disable segmentation.** The use of paging, however, is optional.

These two mechanisms (segmentation and paging) can be configured to support simple single-program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

As shown in Figure 3-1, segmentation provides a mechanism for dividing the processor’s addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**. Segments can be used to hold the code, data, and stack for a program or to hold system data structures (such as a TSS or LDT). If more than one program (or task) is running on a processor, each program can be assigned its own set of segments. The processor then enforces the boundaries between these segments and ensures that one program does not interfere with the execution of another program by writing into the other program’s segments. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

All the segments in a system are contained in the processor’s linear address space. To locate a byte in a particular segment, a **logical address** (also called a far pointer) must be provided. A logical address consists of a segment selector and an offset. The segment selector is a unique identifier for a segment. Among other things it provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor’s linear address space.
If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (typically 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location.

If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program.

When paging is implemented properly in the operating-system or executive, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit IA-32 processors can be paged (transparently) when they are run in virtual-8086 mode.

### 3.2 USING SEGMENTS

The segmentation mechanism supported by the IA-32 architecture can be used to implement a wide variety of system designs. These designs range from flat models that make only minimal use of segmentation to protect...
programs to multi-segmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

The following sections give several examples of how segmentation can be employed in a system to improve memory management performance and reliability.

### 3.2.1 Basic Flat Model

The simplest memory model for a system is the basic “flat model,” in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment (see Figure 3-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

### 3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

![Figure 3-2. Flat Model](image-url)
More complexity can be added to this protected flat model to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other. Similar designs are used by several popular multitasking operating systems.

### 3.2.3 Multi-Segment Model

A multi-segment model (such as the one shown in Figure 3-4) uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

3.2.5 Paging and Segmentation

Paging can be used with any of the segmentation models described in Figures 3-2, 3-3, and 3-4. The processor’s paging mechanism divides the linear address space (into which segments are mapped) into pages (as shown in Figure 3-1). These linear-address-space pages are then mapped to pages in the physical address space. The paging mechanism offers several page-level protection facilities that can be used with or instead of the segment-
PROTECTED-MODE MEMORY MANAGEMENT

Protection facilities. For example, it lets read-write protection be enforced on a page-by-page basis. The paging mechanism also provides two-level user-supervisor protection that can also be specified on a page-by-page basis.

3.3 PHYSICAL ADDRESS SPACE

In protected mode, the IA-32 architecture provides a normal physical address space of 4 GBytes ($2^{32}$ bytes). This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFFFFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

Starting with the Pentium Pro processor, the IA-32 architecture also supports an extension of the physical address space to $2^{36}$ bytes (64 GBytes); with a maximum physical address of FFFFFFFFH. This extension is invoked in either of two ways:

- Using the physical address extension (PAE) flag, located in bit 5 of control register CR4.
- Using the 36-bit page size extension (PSE-36) feature (introduced in the Pentium III processors).

Physical address support has since been extended beyond 36 bits. See Chapter 4, “Paging,” for more information about 36-bit physical addressing.

3.3.1 Intel® 64 Processors and Physical Address Space

On processors that support Intel 64 architecture (CPUID.80000001H:EDX[29] = 1), the size of the physical address range is implementation-specific and indicated by CPUID.80000008H:EAX[bits 7-0].

For the format of information returned in EAX, see “CPUID—CPU Identification” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. See also: Chapter 4, “Paging.”

3.4 LOGICAL AND LINEAR ADDRESSES

At the system-architecture level in protected mode, the processor uses two stages of address translation to arrive at a physical address: logical-address translation and linear address space paging.

Even with the minimum use of segments, every byte in the processor’s address space is accessed with a logical address. A logical address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5). The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte in the segment relative to the base address of the segment.

The processor translates every logical address into a linear address. A linear address is a 32-bit address in the processor’s linear address space. Like the physical address space, the linear address space is a flat (unsegmented), $2^{32}$-byte address space, with addresses ranging from 0 to FFFFFFFFH. The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor. (This step is needed only when a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to ensure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.
If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor’s address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

See also: Chapter 4, “Paging.”

### 3.4.1 Logical Address Translation in IA-32e Mode

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit code by code segment.

### 3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

- **Index** (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

- **TI (table indicator) flag** (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

---

![Figure 3-5. Logical Address to Linear Address Translation](image)

![Figure 3-6. Segment Selector](image)
**Requested Privilege Level (RPL)**

(Bits 0 and 1) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level. See Section 5.5, “Privilege Levels,” for a description of the relationship of the RPL to the CPL of the executing program (or task) and the descriptor privilege level (DPL) of the descriptor the segment selector points to.

The first entry of the GDT is not used by the processor. A segment selector that points to this entry of the GDT (that is, a segment selector with an index of 0 and the TI flag set to 0) is used as a “null segment selector.” The processor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. A null selector can be used to initialize unused segment registers. Loading the CS or SS register with a null segment selector causes a general-protection exception (#GP) to be generated.

Segment selectors are visible to application programs as part of a pointer variable, but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs.

### 3.4.3 Segment Registers

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 3-7). Each of these segment registers support a specific kind of memory reference (code, stack, or data). For virtually any kind of program execution to take place, at least the code-segment (CS), data-segment (DS), and stack-segment (SS) registers must be loaded with valid segment selectors. The processor also provides three additional data-segment registers (ES, FS, and GS), which can be used to make additional data segments available to the currently executing program (or task).

For a program to access a segment, the segment selector for the segment must have been loaded in one of the segment registers. So, although a system can define thousands of segments, only 6 can be available for immediate use. Other segments can be made available by loading their segment selectors into these registers during program execution.

![Figure 3-7. Segment Registers](image-url)

Every segment register has a “visible” part and a “hidden” part. (The hidden part is sometimes referred to as a “descriptor cache” or a “shadow register.”) When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Two kinds of load instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, POP, LDS, LES, LSS, LGS, and LFS instructions. These instructions explicitly reference the segment registers.

2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions, the SYSENTER and SYSEXIT instructions, and the IRET, INT n, INTO, INT3, and INT1 instructions. These instructions change
the contents of the CS register (and sometimes other segment registers) as an incidental part of their operation.

The MOV instruction can also be used to store the visible part of a segment register in a general-purpose register.

### 3.4.4 Segment Loading Instructions in IA-32e Mode

Because ES, DS, and SS segment registers are not used in 64-bit mode, their fields (base, limit, and attribute) in segment descriptor registers are ignored. Some forms of segment load instructions are also invalid (for example, LDS, POP ES). Address calculations that reference the ES, DS, or SS segments are treated as if the segment base is zero.

The processor checks that all linear-address references are in canonical form instead of performing limit checks. Mode switching does not change the contents of the segment registers or the associated descriptor registers. These registers are also not changed during 64-bit mode execution, unless explicit segment loads are performed.

In order to set up compatibility mode for an application, segment-load instructions (MOV to Sreg, POP Sreg) work normally in 64-bit mode. An entry is read from the system descriptor table (GDT or LDT) and is loaded in the hidden portion of the segment register. The descriptor-register base, limit, and attribute fields are all loaded. However, the contents of the data and stack segment selector and the descriptor registers are ignored.

When FS and GS segment overrides are used in 64-bit mode, their respective base addresses are used in the linear address calculation: $(FS$ or $GS).base + index + displacement$. $FS.base$ and $GS.base$ are then expanded to the full linear-address size supported by the implementation. The resulting effective address calculation can wrap across positive and negative addresses; the resulting linear address must be canonical.

In 64-bit mode, memory accesses using FS-segment and GS-segment overrides are not checked for a runtime limit nor subjected to attribute-checking. Normal segment loads (MOV to Sreg and POP Sreg) into FS and GS load a standard 32-bit base value in the hidden portion of the segment register. The base address bits above the standard 32 bits are cleared to 0 to allow consistency for implementations that use less than 64 bits.

The hidden descriptor register fields for $FS.base$ and $GS.base$ are physically mapped to MSRs in order to load all address bits supported by a 64-bit implementation. Software with CPL = 0 (privileged software) can load all supported linear-address bits into $FS.base$ or $GS.base$ using WRMSR. Addresses written into the 64-bit $FS.base$ and $GS.base$ registers must be in canonical form. A WRMSR instruction that attempts to write a non-canonical address to those registers causes a #GP fault.

When in compatibility mode, FS and GS overrides operate as defined by 32-bit mode behavior regardless of the value loaded into the upper 32 linear-address bits of the hidden descriptor register base field. Compatibility mode ignores the upper 32 bits when calculating an effective address.

A new 64-bit mode instruction, SWAPGS, can be used to load GS base. SWAPGS exchanges the kernel data structure pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access the kernel data structures. An attempt to write a non-canonical value (using WRMSR) to the IA32_KERNEL_GS_BASE MSR causes a #GP fault.

### 3.4.5 Segment Descriptors

A segment descriptor is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information. Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs. Figure 3-8 illustrates the general descriptor format for all types of segment descriptors.
The flags and fields in a segment descriptor are as follows:

**Segment limit field**

Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, “Code- and Data-Segment Descriptor Types,” for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segments other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment’s address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

**Base address fields**

Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

**Type field**

Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, “Code- and Data-Segment Descriptor Types,” for a description of how this field is used to specify code and data-segment types.
S (descriptor type) flag
Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

DPL (descriptor privilege level) field
Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment. See Section 5.5, “Privilege Levels,” for a description of the relationship of the DPL to the CPL of the executing code segment and the RPL of a segment selector.

P (segment-present) flag
Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register. Memory management software can use this flag to control which segments are actually loaded into physical memory at a given time. It offers a control in addition to paging for managing virtual memory.

Figure 3-9 shows the format of a segment descriptor when the segment-present flag is clear. When this flag is clear, the operating system or executive is free to use the locations marked “Available” to store its own data, such as information regarding the whereabouts of the missing segment.

D/B (default operation size/default stack pointer size and/or upper bound) flag
Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

- Executable code segment. The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed. The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.

- Stack segment (data segment pointed to by the SS register). The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.

- Expand-down data segment. The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).

G (granularity) flag
Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units. (This flag does not affect the granularity of the base address; it is always byte granular.) When the granularity flag is set, the twelve least significant bits of an offset are not tested when checking the
offset against the segment limit. For example, when the granularity flag is set, a limit of 0 results in valid offsets from 0 to 4095.

**L (64-bit code segment) flag**

In IA-32e mode, bit 21 of the second doubleword of the segment descriptor indicates whether a code segment contains native 64-bit code. A value of 1 indicates instructions in this code segment are executed in 64-bit mode. A value of 0 indicates the instructions in this code segment are executed in compatibility mode. If the L-bit is set, then the D-bit must be cleared. Bit 21 is not used outside IA-32e mode (or for data segments). Because an attempt to activate IA-32e mode will fault if the current CS has the L-bit set (see Section 10.8.5), software operating outside IA-32e mode should avoid loading CS from a descriptor that sets the L-bit.

**Available and reserved bits**

Bit 20 of the second doubleword of the segment descriptor is available for use by system software.

### 3.4.5.1 Code- and Data-Segment Descriptor Types

When the S (descriptor type) flag in a segment descriptor is set, the descriptor is for either a code or a data segment. The highest order bit of the type field (bit 11 of the second double word of the segment descriptor) then determines whether the descriptor is for a data segment (clear) or a code segment (set).

For data segments, the three low-order bits of the type field (bits 8, 9, and 10) are interpreted as accessed (A), write-enable (W), and expansion-direction (E). See Table 3-1 for a description of the encoding of the bits in the type field for code and data segments. Data segments can be read-only or read/write segments, depending on the setting of the write-enable bit.

<table>
<thead>
<tr>
<th>Type Field</th>
<th>Descriptor Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>11 10 9 8 A</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0 0</td>
<td>Data Read-Only</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 1</td>
<td>Data Read-Only, accessed</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 1 0</td>
<td>Data Read/Write</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1 1</td>
<td>Data Read/Write, accessed</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0 0</td>
<td>Data Read-Only, expand-down</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1 1</td>
<td>Data Read-Only, expand-down, accessed</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0 0</td>
<td>Data Read/Write, expand-down</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1 1</td>
<td>Data Read/Write, expand-down, accessed</td>
</tr>
<tr>
<td>C</td>
<td>R A</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0 0</td>
<td>Code Execute-Only</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 0 1</td>
<td>Code Execute-Only, accessed</td>
</tr>
<tr>
<td>10</td>
<td>1 0 0 1 0</td>
<td>Code Execute/Read</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1 1</td>
<td>Code Execute/Read, accessed</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0 0</td>
<td>Code Execute-Only, conforming</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1 1</td>
<td>Code Execute-Only, conforming, accessed</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0 0</td>
<td>Code Execute/Read, conforming</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1 1</td>
<td>Code Execute/Read, conforming, accessed</td>
</tr>
</tbody>
</table>

Stack segments are data segments which must be read/write segments. Loading the SS register with a segment selector for a nonwritable data segment generates a general-protection exception (#GP). If the size of a stack segment needs to be changed dynamically, the stack segment can be an expand-down data segment (expansion-
direction flag set). Here, dynamically changing the segment limit causes stack space to be added to the bottom of
the stack. If the size of a stack segment is intended to remain static, the stack segment may be either an expand-
up or expand-down type.

The accessed bit indicates whether the segment has been accessed since the last time the operating-system or
executive cleared the bit. The processor sets this bit whenever it loads a segment selector for the segment into a
segment register, assuming that the type of memory that contains the segment descriptor supports processor
writes. The bit remains set until explicitly cleared. This bit can be used both for virtual memory management and
for debugging.

For code segments, the three low-order bits of the type field are interpreted as accessed (A), read enable (R), and
conforming (C). Code segments can be execute-only or execute/read, depending on the setting of the read-enable
bit. An execute/read segment might be used when constants or other static data have been placed with instruction
code in a ROM. Here, data can be read from the code segment either by using an instruction with a CS override
prefix or by loading a segment selector for the code segment in a data-segment register (the DS, ES, FS, or GS
registers). In protected mode, code segments are not writable.

Code segments can be either conforming or nonconforming. A transfer of execution into a more-privileged
conforming segment allows execution to continue at the current privilege level. A transfer into a nonconforming
segment at a different privilege level results in a general-protection exception (#GP), unless a call gate or task gate
is used (see Section 5.8.1, “Direct Calls or Jumps to Code Segments,” for more information on conforming and
nonconforming code segments). System utilities that do not access protected facilities and handlers for some types
of exceptions (such as, divide error or overflow) may be loaded in conforming code segments. Utilities that need to
be protected from less privileged programs and procedures should be placed in nonconforming code segments.

NOTE

Execution cannot be transferred by a call or a jump to a less-privileged (numerically higher
privilege level) code segment, regardless of whether the target segment is a conforming or
nonconforming code segment. Attempting such an execution transfer will result in a general-
protection exception.

All data segments are nonconforming, meaning that they cannot be accessed by less privileged programs or proce-
dures (code executing at numerically higher privilege levels). Unlike code segments, however, data segments can
be accessed by more privileged programs or procedures (code executing at numerically lower privilege levels)
without using a special access gate.

If the segment descriptors in the GDT or an LDT are placed in ROM, the processor can enter an indefinite loop if
software or the processor attempts to update (write to) the ROM-based segment descriptors. To prevent this
problem, set the accessed bits for all segment descriptors placed in a ROM. Also, remove operating-system or
executive code that attempts to modify segment descriptors located in ROM.

3.5 SYSTEM DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The
processor recognizes the following types of system descriptors:

• Local descriptor-table (LDT) segment descriptor.
• Task-state segment (TSS) descriptor.
• Call-gate descriptor.
• Interrupt-gate descriptor.
• Trap-gate descriptor.
• Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors. System-
segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves
“gates,” which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which
hold segment selectors for TSS’s (task gates).
Table 3-2 shows the encoding of the type field for system-segment descriptors and gate descriptors. Note that system descriptors in IA-32e mode are 16 bytes instead of 8 bytes.

**Table 3-2. System-Segment and Gate-Descriptor Types**

<table>
<thead>
<tr>
<th>Type Field</th>
<th>Description</th>
<th>32-Bit Mode</th>
<th>IA-32e Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>16-bit TSS (Available)</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>LDT</td>
<td>LDT</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>16-bit TSS (Busy)</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>16-bit Call Gate</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>Task Gate</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>16-bit Interrupt Gate</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>16-bit Trap Gate</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>32-bit TSS (Available)</td>
<td>64-bit TSS (Available)</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>32-bit TSS (Busy)</td>
<td>64-bit TSS (Busy)</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>32-bit Call Gate</td>
<td>64-bit Call Gate</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>32-bit Interrupt Gate</td>
<td>64-bit Interrupt Gate</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>32-bit Trap Gate</td>
<td>64-bit Trap Gate</td>
</tr>
</tbody>
</table>

See also: Section 3.5.1, “Segment Descriptor Tables,” and Section 8.2.2, “TSS Descriptor,” (for more information on the system-segment descriptors); see Section 5.8.3, “Call Gates,” Section 6.11, “IDT Descriptors,” and Section 8.2.5, “Task-Gate Descriptor,” (for more information on the gate descriptors).

### 3.5.1 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors (see Figure 3-10). A descriptor table is variable in length and can contain up to 8192 \(2^{13}\) 8-byte descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT).
- The local descriptor tables (LDT).
Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register (see Section 2.4, “Memory-Management Registers”). The base address of the GDT should be aligned on an eight-byte boundary to yield the best processor performance. The limit value for the GDT is expressed in bytes. As with segments, the limit value is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes long, the GDT limit should always be one less than an integral multiple of eight (that is, $8N - 1$).

The first descriptor in the GDT is not used by the processor. A segment selector to this “null descriptor” does not generate an exception when loaded into a data-segment register (DS, ES, FS, or GS), but it always generates a general-protection exception (#GP) when an attempt is made to access memory using the descriptor. By initializing the segment registers with this segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

The LDT is located in a system segment of the LDT type. The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. The segment descriptor for an LDT can be located anywhere in the GDT. See Section 3.5, “System Descriptor Types,” for information on the LDT segment-descriptor type.

An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register (see Section 2.4, “Memory-Management Registers”).

When the GDTR register is stored (using the SGDT instruction), a 48-bit “pseudo-descriptor” is stored in memory (see top diagram in Figure 3-11). To avoid alignment check faults in user mode (privilege level 3), the pseudo-descriptor should be located at an odd word address (that is, address MOD 4 is equal to 2). This causes the
processor to store an aligned word, followed by an aligned doubleword. User-mode programs normally do not store pseudo-descriptors, but the possibility of generating an alignment check fault can be avoided by aligning pseudo-descriptors in this way. The same alignment should be used when storing the IDTR register using the SIDT instruction. When storing the LDTR or task register (using the SLDT or STR instruction, respectively), the pseudo-descriptor should be located at a doubleword address (that is, address MOD 4 is equal to 0).

3.5.2 Segment Descriptor Tables in IA-32e Mode

In IA-32e mode, a segment descriptor table can contain up to 8192 \(2^{13}\) 8-byte descriptors. An entry in the segment descriptor table can be 8 bytes. System descriptors are expanded to 16 bytes (occupying the space of two entries).

GDTR and LDTR registers are expanded to hold 64-bit base address. The corresponding pseudo-descriptor is 80 bits. (see the bottom diagram in Figure 3-11).

The following system descriptors expand to 16 bytes:

- Call gate descriptors (see Section 5.8.3.1, "IA-32e Mode Call Gates").
- IDT gate descriptors (see Section 6.14.1, "64-Bit Mode IDT").
- LDT and TSS descriptors (see Section 8.2.3, "TSS Descriptor in 64-bit mode").

Figure 3-11. Pseudo-Descriptor Formats
Chapter 3 explains how segmentation converts logical addresses to linear addresses. **Paging** (or linear-address translation) is the process of translating linear addresses so that they can be used to access memory or I/O devices. Paging translates each linear address to a **physical address** and determines, for each translation, what accesses to the linear address are allowed (the address's **access rights**) and the type of caching used for such accesses (the address's **memory type**).

Intel-64 processors support four different paging modes. These modes are identified and defined in Section 4.1. Section 4.2 gives an overview of the translation mechanism that is used in all modes. Section 4.3, Section 4.4, and Section 4.5 discuss the four paging modes in detail.

Section 4.6 details how paging determines and uses access rights. Section 4.7 discusses exceptions that may be generated by paging (page-fault exceptions). Section 4.8 considers data which the processor writes in response to linear-address accesses (accessed and dirty flags).

Section 4.9 describes how paging determines the memory types used for accesses to linear addresses. Section 4.10 provides details of how a processor may cache information about linear-address translation. Section 4.11 outlines interactions between paging and certain VMX features. Section 4.12 gives an overview of how paging can be used to implement virtual memory.

## 4.1 Paging Modes and Control Bits

Paging behavior is controlled by the following control bits:

- The WP and PG flags in control register CR0 (bit 16 and bit 31, respectively).
- The PSE, PAE, PGE, LA57, PCIDE, SMEP, SMAP, PKE, CET, and PKS flags in control register CR4 (bit 4, bit 5, bit 7, bit 12, bit 17, bit 20, bit 21, bit 22, bit 23, and bit 24, respectively).
- The LME and NXE flags in the IA32_EFER MSR (bit 8 and bit 11, respectively).
- The AC flag in the EFLAGS register (bit 18).
- The "enable HLAT" VM-execution control (tertiary processor-based VM-execution control bit 1; see Section 25.6.2, "Processor-Based VM-Execution Controls," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C).

Software enables paging by using the MOV to CR0 instruction to set CR0.PG. Before doing so, software should ensure that control register CR3 contains the physical address of the first paging structure that the processor will use for linear-address translation (see Section 4.2) and that that structure is initialized as desired. See Table 4-3, Table 4-7, and Table 4-12 for the use of CR3 in the different paging modes.

Section 4.1.1 describes how the values of CR0.PG, CR4.PAE, CR4.LA57, and IA32_EFER.LME determine whether paging is enabled and, if so, which of four paging modes is in use. Section 4.1.2 explains how to manage these bits to establish or make changes in paging modes. Section 4.1.3 discusses how CR0.WP, CR4.PSE, CR4.PGE, CR4.PCEDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE modify the operation of the different paging modes.

### 4.1.1 Four Paging Modes

If CR0.PG = 0, paging is not used. The logical processor treats all linear addresses as if they were physical addresses. CR4.PAE, CR4.LA57, and IA32_EFER.LME are ignored by the processor, as are CR0.WP, CR4.PSE, CR4.PGE, CR4.SMEP, CR4.SMAP, and IA32_EFER.NXE. (CR4.CET is also ignored insofar as it affects linear-address access rights.)

Paging is enabled if CR0.PG = 1. Paging can be enabled only if protection is enabled (CR0.PE = 1). If paging is enabled, one of four paging modes is used. The values of CR4.PAE, CR4.LA57, and IA32_EFER.LME determine which paging mode is used:
• If CR4.PAE = 0, 32-bit paging is used. 32-bit paging is detailed in Section 4.3. 32-bit paging uses CR0.WP, CR4.PSE, CR4.PGE, CR4.SMEP, CR4.SMAP, and CR4.CET as described in Section 4.1.3 and Section 4.6.

• If CR4.PAE = 1 and IA32_EFER.LME = 0, PAE paging is used. PAE paging is detailed in Section 4.4. PAE paging uses CR0.WP, CR4.PGE, CR4.SMEP, CR4.SMAP, CR4.CET, and IA32_EFER.NXE as described in Section 4.1.3 and Section 4.6.

• If CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 0, 4-level paging is used. 4-level paging is detailed in Section 4.5 (along with 5-level paging). 4-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE as described in Section 4.1.3 and Section 4.6.

• If CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 1, 5-level paging is used. 5-level paging is detailed in Section 4.5 (along with 4-level paging). 5-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE as described in Section 4.1.3 and Section 4.6.

NOTE

32-bit paging and PAE paging can be used only in legacy protected mode (IA32_EFER.LME = 0). In contrast, 4-level paging and 5-level paging can be used only IA-32e mode (IA32_EFER.LME = 1).

The four paging modes differ with regard to the following details:

• Linear-address width. The size of the linear addresses that can be translated.

• Physical-address width. The size of the physical addresses produced by paging.

• Page size. The granularity at which linear addresses are translated. Linear addresses on the same page are translated to corresponding physical addresses on the same page.

• Support for execute-disable access rights. In some paging modes, software can be prevented from fetching instructions from pages that are otherwise readable.

• Support for PCIDs. With 4-level paging and 5-level paging, software can enable a facility by which a logical processor caches information for multiple linear-address spaces. The processor may retain cached information when software switches between different linear-address spaces.

• Support for protection keys. With 4-level paging and 5-level paging, each linear address is associated with a protection key. Software can use the protection-key rights registers to disable, for each protection key, how certain accesses to linear addresses associated with that protection key.

Table 4-1 illustrates the principal differences between the four paging modes.

Table 4-1. Properties of Different Paging Modes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>02</td>
<td>N/A</td>
<td>32</td>
<td>Up to 403</td>
<td>4 KB 4 MB4</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PAE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N/A</td>
<td>32</td>
<td>Up to 52</td>
<td>4 KB 2 MB</td>
<td>Yes5</td>
<td>No</td>
</tr>
<tr>
<td>4-level</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>48</td>
<td>Up to 52</td>
<td>4 KB 2 MB 1 GB6</td>
<td>Yes5</td>
<td>Yes7</td>
</tr>
</tbody>
</table>

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

2. The LMA flag in the IA32_EFER MSR (bit 10) is a status bit that indicates whether the logical processor is in IA-32e mode (and thus uses either 4-level paging or 5-level paging). The processor always sets IA32_EFER.LMA to CR0.PG & IA32_EFER.LME. Software cannot directly modify IA32_EFER.LMA; an execution of WRMSR to the IA32_EFER MSR ignores bit 10 of its source operand.
Because 32-bit paging and PAE paging are used only in legacy protected mode and because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

4-level paging and 5-level paging are used only in IA-32e mode. IA-32e mode has two sub-modes:

- Compatibility mode. This sub-mode uses only 32-bit linear addresses. In this sub-mode, 4-level paging and 5-level paging treat bits 63:32 of such an address as all 0.
- 64-bit mode. While this sub-mode produces 64-bit linear addresses, the processor enforces \textit{canonicality}, meaning that the upper bits of such an address are identical: bits 63:47 for 4-level paging and bits 63:56 for 5-level paging. 4-level paging (respectively, 5-level paging) does not use bits 63:48 (respectively, bits 63:57) of such addresses.

### 4.1.2 Paging-Mode Enabling

If CR0.PG = 1, a logical processor is in one of four paging modes, depending on the values of CR4.PAE, IA32_EFER.LME, and CR4.LA57. Figure 4-1 illustrates how software can enable these modes and make transitions between them. The following items identify certain limitations and other details:

- IA32_EFER.LME cannot be modified while paging is enabled (CR0.PG = 1). Attempts to do so using WRMSR cause a general-protection exception (#GP(0)).
- Paging cannot be enabled (by setting CR0.PG to 1) while CR4.PAE = 0 and IA32_EFER.LME = 1. Attempts to do so using MOV to CR0 cause a general-protection exception (#GP(0)).
- One node in Figure 4-1 is labeled "IA-32e mode." This node represents either 4-level paging (if CR4.LA57 = 0) or 5-level paging (if CR4.LA57 = 1). As noted in the following items, software cannot modify CR4.LA57 (affecting transition between 4-level paging and 5-level paging) without first disabling paging.
- CR4.PAE and CR4.LA57 cannot be modified while either 4-level paging or 5-level paging is in use (when CR0.PG = 1 and IA32_EFER.LME = 1). Attempts to do so using MOV to CR4 cause a general-protection exception (#GP(0)).
- Regardless of the current paging mode, software can disable paging by clearing CR0.PG with MOV to CR0.\footnote{If the logical processor is in 64-bit mode or if CR4.PCIDE = 1, an attempt to clear CR0.PG causes a general-protection exception (#GP). Software should transition to compatibility mode and clear CR4.PCIDE before attempting to disable paging.}
- Software can transition between 32-bit paging and PAE paging by changing the value of CR4.PAE with MOV to CR4.
Software cannot transition directly between 4-level paging (or 5-level paging) and any of other paging mode. It must first disable paging (by clearing CR0.PG with MOV to CR0), then set CR4.PAE, IA32_EFER.LME, and CR4.LA57 to the desired values (with MOV to CR4 and WRMSR), and then re-enable paging (by setting CR0.PG with MOV to CR0). As noted earlier, an attempt to modify CR4.PAE, IA32_EFER.LME, or CR.LA57 while 4-level paging or 5-level paging is enabled causes a general-protection exception (#GP(0)).

VMX transitions allow transitions between paging modes that are not possible using MOV to CR or WRMSR. This is because VMX transitions can load CR0, CR4, and IA32_EFER in one operation. See Section 4.11.1.

4.1.3 Paging-ModeModifiers

Details of how each paging mode operates are determined by the following control bits:

- The WP flag in CR0 (bit 16).
- The PSE, PGE, PCIDE, SMEP, SMAP, PKE, CET, and PKS flags in CR4 (bit 4, bit 7, bit 17, bit 20, bit 21, bit 22, bit 23, and bit 24, respectively).
- The NXE flag in the IA32_EFER MSR (bit 11).
- The “enable HLAT” VM-execution control (tertiary processor-based VM-execution control bit 1).

CR0.WP allows pages to be protected from supervisor-mode writes. If CR0.WP = 0, supervisor-mode write accesses are allowed to linear addresses with read-only access rights; if CR0.WP = 1, they are not. (User-mode write accesses are never allowed to linear addresses with read-only access rights, regardless of the value of
Section 4.6 explains how access rights are determined, including the definition of supervisor-mode and user-mode accesses.

CR4.PSE enables 4-MByte pages for 32-bit paging. If CR4.PSE = 0, 32-bit paging can only use 4-KByte pages; if CR4.PSE = 1, 32-bit paging can use both 4-KByte pages and 4-MByte pages. See Section 4.3 for more information. (PAE paging, 4-level paging, and 5-level paging can use multiple page sizes regardless of the value of CR4.PSE.)

CR4.PGE enables global pages. If CR4.PGE = 0, no translations are shared across address spaces; if CR4.PGE = 1, specified translations may be shared across address spaces. See Section 4.10.2.4 for more information.

CR4.PCIDE enables process-context identifiers (PCIDs) for 4-level paging and 5-level paging. PCIDs allow a logical processor to cache information for multiple linear-address spaces. See Section 4.10.1 for more information.

CR4.SMEP allows pages to be protected from supervisor-mode instruction fetches. If CR4.SMEP = 1, software operating in supervisor mode cannot fetch instructions from linear addresses that are accessible in user mode. Section 4.6 explains how access rights are determined, including the definition of supervisor-mode accesses and user-mode accessibility.

CR4.SMAP allows pages to be protected from supervisor-mode data accesses. If CR4.SMAP = 1, software operating in supervisor mode cannot access data at linear addresses that are accessible in user mode. Software can override this protection by setting EFLAGS.AC. Section 4.6 explains how access rights are determined, including the definition of supervisor-mode accesses and user-mode accessibility.

CR4.PKE and CR4.PKS enable specification of access rights based on protection keys. 4-level paging and 5-level paging associate each linear address with a protection key. When CR4.PKE = 1, the PKRU register specifies, for each protection key, whether user-mode linear addresses with that protection key can be read or written. When CR4.PKS = 1, the IA32_PKRS MSR does the same for supervisor-mode linear addresses. See Section 4.6 for more information.

CR4.CET enables control-flow enforcement technology, including the shadow-stack feature. If CR4.CET = 1, certain memory accesses are identified as shadow-stack accesses and certain linear addresses translate to shadow-stack pages. Section 4.6 explains how access rights are determined for these accesses and pages. (The processor allows CR4.CET to be set only if CR0.WP is also set.)

IA32_EFER.NXE enables execute-disable access rights for PAE paging, 4-level paging, and 5-level paging. If IA32_EFER.NXE = 1, instruction fetches can be prevented from specified linear addresses (even if data reads from the addresses are allowed). Section 4.6 explains how access rights are determined. (IA32_EFER.NXE has no effect with 32-bit paging. Software that wants to use this feature to limit instruction fetches from readable pages must use PAE paging, 4-level paging, or 5-level paging.)

The “enable HLAT” VM-execution control enables HLAT paging for 4-level paging and 5-level paging. HLAT paging does not use control register CR3 to identify the address of the first paging structure used for linear-address translation; instead, that structure is located using a field in the virtual-machine control structure (VMCS). In addition, HLAT paging interprets certain bits in paging-structure entries differently than ordinary paging. See Section 4.5 for details.

### 4.1.4 Enumeration of Paging Features by CPUID

Software can discover support for different paging features using the CPUID instruction:

- **PSE**: page-size extensions for 32-bit paging.
  
  - If CPUID.01H:EDX.PSE [bit 3] = 1, CR4.PSE may be set to 1, enabling support for 4-MByte pages with 32-bit paging (see Section 4.3).

- **PAE**: physical-address extension.
  
  - If CPUID.01H:EDX.PAE [bit 6] = 1, CR4.PAE may be set to 1, enabling PAE paging (this setting is also required for 4-level paging and 5-level paging).

- **PGE**: global-page support.
  
  - If CPUID.01H:EDX.PGE [bit 13] = 1, CR4.PGE may be set to 1, enabling the global-page feature (see Section 4.10.2.4).

- **PAT**: page-attribute table.
  
  - If CPUID.01H:EDX.PAT [bit 16] = 1, the 8-entry page-attribute table (PAT) is supported. When the PAT is
supported, three bits in certain paging-structure entries select a memory type (used to determine type of caching used) from the PAT (see Section 4.9.2).

- **PSE-36**: page-size extensions with 40-bit physical-address extension.
  If CPUID.01H:EDX.PSE-36 [bit 17] = 1, the PSE-36 mechanism is supported, indicating that translations using 4-MByte pages with 32-bit paging may produce physical addresses with up to 40 bits (see Section 4.3).

- **PCID**: process-context identifiers.
  If CPUID.01H:ECX.PCID [bit 17] = 1, CR4.PCIDE may be set to 1, enabling process-context identifiers (see Section 4.10.1).

- **SMEP**: supervisor-mode execution prevention.
  If CPUID.(EAX=07H,ECX=0H):EBX.SMEP [bit 7] = 1, CR4.SMEP may be set to 1, enabling supervisor-mode execution prevention (see Section 4.6).

- **SMAP**: supervisor-mode access prevention.
  If CPUID.(EAX=07H,ECX=0H):EBX.SMAP [bit 20] = 1, CR4.SMAP may be set to 1, enabling supervisor-mode access prevention (see Section 4.6).

- **PKU**: protection keys for user-mode pages.
  If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, CR4.PKE may be set to 1, enabling protection keys for user-mode pages (see Section 4.6).

- **OSPKE**: enabling of protection keys for user-mode pages.
  CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4] returns the value of CR4.PKE. Thus, protection keys for user-mode pages are enabled if this flag is 1 (see Section 4.6).

- **CET**: control-flow enforcement technology.
  If CPUID.(EAX=07H,ECX=0H):ECX.CET_SS [bit 7] = 1, CR4.CET may be set to 1, enabling shadow-stack pages (see Section 4.6).

- **LA57**: 57-bit linear addresses and 5-level paging.
  If CPUID.(EAX=07H,ECX=0H):ECX.LA57 [bit 16] = 1, CR4.LA57 may be set to 1, enabling 5-level paging.

- **PKS**: protection keys for supervisor-mode pages.
  If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, CR4.PKS may be set to 1, enabling protection keys for supervisor-mode pages (see Section 4.6).

- **NX**: execute disable.
  If CPUID.80000001H:EDX.NX [bit 20] = 1, IA32_EFER.NXE may be set to 1, allowing software to disable execute access to selected pages (see Section 4.6). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.NXE to be set to 1.)

- **Page1GB**: 1-GByte pages.
  If CPUID.80000001H:EDX.Page1GB [bit 26] = 1, 1-GByte pages may be supported with 4-level paging and 5-level paging (see Section 4.5).

- **LM**: IA-32e mode support.
  If CPUID.80000001H:EDX.LM [bit 29] = 1, IA32_EFER.LME may be set to 1, enabling IA-32e mode (with either 4-level paging or 5-level paging). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.LME to be set to 1.)

- **CPUID.80000008H:EAX[7:0]** reports the physical-address width supported by the processor. (For processors that do not support CPUID function 80000008H, the width is generally 36 if CPUID.01H:EDX.PAE [bit 6] = 1 and 32 otherwise. This width is referred to as MAXPHYADDR. MAXPHYADDR is at most 52.

- **CPUID.80000008H:EAX[15:8]** reports the linear-address width supported by the processor. Generally, this value is reported as follows:
  - If CPUID.80000001H:EDX.LM [bit 29] = 0, the value is reported as 32.
  - If CPUID.80000001H:EDX.LM [bit 29] = 1 and CPUID.(EAX=07H,ECX=0H):ECX.LA57 [bit 16] = 0, the value is reported as 48.
  - If CPUID.(EAX=07H,ECX=0H):ECX.LA57 [bit 16] = 1, the value is reported as 57.
  (Processors that do not support CPUID function 80000008H, support a linear-address width of 32.)
4.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW

All four paging modes translate linear addresses using hierarchical paging structures. This section provides an overview of their operation. Section 4.3, Section 4.4, Section 4.5, and Section 4.6 provide details for the four paging modes.

Every paging structure is 4096 Bytes in size and comprises a number of individual entries. With 32-bit paging, each entry is 32 bits (4 bytes); there are thus 1024 entries in each structure. With the other paging modes, each entry is 64 bits (8 bytes); there are thus 512 entries in each structure. (PAE paging includes one exception, a paging structure that is 32 bytes in size, containing 4 64-bit entries.)

The processor uses the upper portion of a linear address to identify a series of paging-structure entries. The last of these entries identifies the physical address of the region to which the linear address translates (called the page frame). The lower portion of the linear address (called the page offset) identifies the specific address within that region to which the linear address translates.

Each paging-structure entry contains a physical address, which is either the address of another paging structure or the address of a page frame. In the first case, the entry is said to reference the other paging structure; in the latter, the entry is said to map a page.

The first paging structure used for any translation is located at the physical address in CR3. A linear address is translated using the following iterative procedure. A portion of the linear address (initially the uppermost bits) selects an entry in a paging structure (initially the one located using CR3). If that entry references another paging structure, the process continues with that paging structure and with the portion of the linear address immediately below that just used. If instead the entry maps a page, the process completes: the physical address in the entry is that of the page frame and the remaining lower portion of the linear address is the page offset.

The following items give an example for each of the four paging modes (each example locates a 4-KByte page frame):

1. With 32-bit paging, each paging structure comprises 1024 = 2^10 entries. For this reason, the translation process uses 10 bits at a time from a 32-bit linear address. Bits 31:22 identify the first paging-structure entry and bits 21:12 identify a second. The latter identifies the page frame. Bits 11:0 of the linear address are the page offset within the 4-KByte page frame. (See Figure 4-2 for an illustration.)
2. With PAE paging, the first paging structure comprises only 4 = 2^2 entries. Translation thus begins by using bits 31:30 from a 32-bit linear address to identify the first paging-structure entry. Other paging structures comprise 512 = 2^9 entries, so the process continues by using 9 bits at a time. Bits 29:21 identify a second paging-structure entry and bits 20:12 identify a third. This last identifies the page frame. (See Figure 4-5 for an illustration.)
3. With 4-level paging, each paging structure comprises 512 = 2^9 entries and translation uses 9 bits at a time from a 48-bit linear address. Bits 47:39 identify the first paging-structure entry, bits 38:30 identify a second, bits 29:21 a third, and bits 20:12 identify a fourth. Again, the last identifies the page frame. (See Figure 4-8 for an illustration.)
4. 5-level paging is similar to 4-level paging except that 5-level paging translates 57-bit linear addresses. Bits 56:48 identify the first paging-structure entry, while the remaining bits are used as with 4-level paging.

The translation process in each of the examples above completes by identifying a page frame; the page frame is part of the translation of the original linear address. In some cases, however, the paging structures may be configured so that the translation process terminates before identifying a page frame. This occurs if the process encounters a paging-structure entry that is marked “not present” (because its P flag — bit 0 — is clear) or in which a reserved bit is set. In this case, there is no translation for the linear address; an access to that address causes a page-fault exception (see Section 4.7).

In the examples above, a paging-structure entry maps a page with a 4-KByte page frame when only 12 bits remain in the linear address; entries identified earlier always reference other paging structures. That may not apply in other cases. The following items identify when an entry maps a page and when it references another paging structure:

1. If more than 12 bits remain in the linear address, bit 7 (PS — page size) of the current paging-structure entry is consulted. If the bit is 0, the entry references another paging structure; if the bit is 1, the entry maps a page.

1. If HLAT paging is in use, a different mechanism is used to identify the first paging structure. See Section 4.5 for more information.
• If only 12 bits remain in the linear address, the current paging-structure entry always maps a page (bit 7 is used for other purposes).

If a paging-structure entry maps a page when more than 12 bits remain in the linear address, the entry identifies a page frame larger than 4 KBytes. For example, 32-bit paging uses the upper 10 bits of a linear address to locate the first paging-structure entry; 22 bits remain. If that entry maps a page, the page frame is $2^{22}$ Bytes = 4 MBytes. 32-bit paging can use 4-MByte pages if CR4.PSE = 1. The other paging modes can use 2-MByte pages (regardless of the value of CR4.PSE). 4-level paging and 5-level paging can use 1-GByte pages if the processor supports them (see Section 4.1.4).

Paging structures are given different names based on their uses in the translation process. Table 4-2 gives the names of the different paging structures. It also provides, for each structure, the source of the physical address used to locate it (CR3 or a different paging-structure entry); the bits in the linear address used to select an entry from the structure; and details of whether and how such an entry can map a page.

Table 4-2. Paging Structures in the Different Paging Modes

<table>
<thead>
<tr>
<th>Paging Structure</th>
<th>Entry Name</th>
<th>Paging Mode</th>
<th>Physical Address of Structure</th>
<th>Bits Selecting Entry</th>
<th>Page Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>PML5 table</td>
<td>PML5E</td>
<td>32-bit, PAE, 4-level</td>
<td>N/A</td>
<td></td>
<td>N/A (PS must be 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5-level</td>
<td>CR3</td>
<td>56:48</td>
<td>N/A (PS must be 0)</td>
</tr>
<tr>
<td>PML4 table</td>
<td>PML4E</td>
<td>32-bit, PAE</td>
<td>N/A</td>
<td></td>
<td>N/A (PS must be 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-level</td>
<td>CR3</td>
<td>47:39</td>
<td>N/A (PS must be 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5-level</td>
<td>PML5E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page-directory-pointer table</td>
<td>PDTE</td>
<td>32-bit</td>
<td>N/A</td>
<td></td>
<td>N/A (PS must be 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAE</td>
<td>CR3</td>
<td>31:30</td>
<td>N/A (PS must be 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-level, 5-level</td>
<td>PML4E</td>
<td>38:30</td>
<td>1-GByte page if PS=1</td>
</tr>
<tr>
<td>Page directory</td>
<td>PDE</td>
<td>32-bit</td>
<td>CR3</td>
<td>31:22</td>
<td>4-MByte page if PS=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAE, 4-level, 5-level</td>
<td>PDPTE</td>
<td>29:21</td>
<td>2-MByte page if PS=1</td>
</tr>
<tr>
<td>Page table</td>
<td>PTE</td>
<td>32-bit</td>
<td>PDE</td>
<td>21:12</td>
<td>4-KByte page</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAE, 4-level, 5-level</td>
<td>PDE</td>
<td>20:12</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. If HLAT paging is in use, a different mechanism is used to identify the first paging structure. See Section 4.5 for more information.
2. Not all processors support 1-GByte pages; see Section 4.1.4.
3. 32-bit paging ignores the PS flag in a PDE (and uses the entry to reference a page table) unless CR4.PSE = 1. Not all processors support 4-MByte pages with 32-bit paging; see Section 4.1.4.
4.3 32-BIT PAGING

A logical processor uses 32-bit paging if CR0.PG = 1 and CR4.PAE = 0. 32-bit paging translates 32-bit linear addresses to 40-bit physical addresses.\(^1\) Although 40 bits corresponds to 1 TByte, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

32-bit paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the page directory. Table 4-3 illustrates how CR3 is used with 32-bit paging.

32-bit paging may map linear addresses to either 4-KByte pages or 4-MByte pages. Figure 4-2 illustrates the translation process when it uses a 4-KByte page; Figure 4-3 covers the case of a 4-MByte page. The following items describe the 32-bit paging process in more detail as well as how the page size is determined:

- A 4-KByte naturally aligned page directory is located at the physical address specified in bits 31:12 of CR3 (see Table 4-3). A page directory comprises 1024 32-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from CR3.
  - Bits 11:2 are bits 31:22 of the linear address.
  - Bits 1:0 are 0.

Because a PDE is identified using bits 31:22 of the linear address, it controls access to a 4-Mbyte region of the linear-address space. Use of the PDE depends on CR4.PSE and the PDE’s PS flag (bit 7):

- If CR4.PSE = 1 and the PDE’s PS flag is 1, the PDE maps a 4-MByte page (see Table 4-4). The final physical address is computed as follows:
  - Bits 39:32 are bits 20:13 of the PDE.
  - Bits 31:22 are bits 31:22 of the PDE.\(^2\)
  - Bits 21:0 are from the original linear address.

- If CR4.PSE = 0 or the PDE’s PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 31:12 of the PDE (see Table 4-5). A page table comprises 1024 32-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from the PDE.
  - Bits 11:2 are bits 21:12 of the linear address.
  - Bits 1:0 are 0.

- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-6). The final physical address is computed as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from the PTE.
  - Bits 11:0 are from the original linear address.

If a paging-structure entry’s P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 4.7).

---

1. Bits in the range 39:32 are 0 in any physical address used by 32-bit paging except those used to map 4-MByte pages. If the processor does not support the PSE-36 mechanism, this is true also for physical addresses used to map 4-MByte pages. If the processor does support the PSE-36 mechanism and MAXPHYADDR < 40, bits in the range 39:MAXPHYADDR are 0 in any physical address used to map a 4-MByte page. (The corresponding bits are reserved in PDEs.) See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

2. The upper bits in the final physical address do not all come from corresponding positions in the PDE; the physical-address bits in the PDE are not all contiguous.
With 32-bit paging, there are reserved bits only if CR4.PSE = 1:

- If the P flag and the PS flag (bit 7) of a PDE are both 1, the bits reserved depend on MAXPHYADDR, and whether the PSE-36 mechanism is supported:\textsuperscript{1}
  - If the PSE-36 mechanism is not supported, bits 21:13 are reserved.
  - If the PSE-36 mechanism is supported, bits 21:\(M-19\) are reserved, where \(M\) is the minimum of 40 and MAXPHYADDR.
- If the PAT is not supported:\textsuperscript{2}
  - If the P flag of a PTE is 1, bit 7 is reserved.
  - If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

(If CR4.PSE = 0, no bits are reserved with 32-bit paging.)

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

---

\textsuperscript{1} See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

\textsuperscript{2} See Section 4.1.4 for how to determine whether the PAT is supported.

---

1. See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.
2. See Section 4.1.4 for how to determine whether the PAT is supported.
Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

![Diagram of Linear-Address Translation to a 4-MByte Page using 32-Bit Paging](image)

**Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging**

<table>
<thead>
<tr>
<th>CR3</th>
<th>Bit 31:22 of address of 4MB page frame</th>
<th>Reserved (must be 0)</th>
<th>Bits 39:32 of address2</th>
<th>Address of page table</th>
<th>Ignored</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>Ignored</th>
<th>P</th>
<th>W</th>
<th>T</th>
<th>Ignored</th>
<th>CR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE: 4MB page</td>
<td>Bits 31:22 of address of 4MB page frame</td>
<td>Reserved (must be 0)</td>
<td>Bits 39:32 of address2</td>
<td>Address of page directory</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>CR3</td>
</tr>
<tr>
<td>PDE: page table</td>
<td>Address of page directory</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
</tr>
<tr>
<td>PDE: not present</td>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
</tr>
<tr>
<td>PTE: not present</td>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
</tr>
<tr>
<td>PTE: 4KB page</td>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
</tr>
<tr>
<td>PDE: not present</td>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
</tr>
<tr>
<td>PTE: not present</td>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

**Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging**

**NOTES:**
1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.
### Table 4-3. Use of CR3 with 32-Bit Paging

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)</td>
</tr>
<tr>
<td>11:5</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:12</td>
<td>Physical address of the 4-KByte aligned page directory used for linear-address translation</td>
</tr>
<tr>
<td>63:32</td>
<td>Ignored (these bits exist only on processors supporting the Intel-64 architecture)</td>
</tr>
</tbody>
</table>

### Table 4-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-MByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-MByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-MByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-MByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 1 (otherwise, this entry references a page table; see Table 4-5)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>12 (PAT)</td>
<td>If the PAT is supported, indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)¹</td>
</tr>
<tr>
<td>(M-20):13</td>
<td>Bits (M-1):32 of physical address of the 4-MByte page referenced by this entry²</td>
</tr>
<tr>
<td>21:(M-19)</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>31:22</td>
<td>Bits 31:22 of physical address of the 4-MByte page referenced by this entry</td>
</tr>
</tbody>
</table>

**NOTES:**

1. See Section 4.1.4 for how to determine whether the PAT is supported.
2. If the PSE-36 mechanism is not supported, M is 32, and this row does not apply. If the PSE-36 mechanism is supported, M is the minimum of 40 and MAXPHYADDR (this row does not apply if MAXPHYADDR = 32). See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.
### Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page table</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
<tr>
<td>6</td>
<td>Ignored</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored</td>
</tr>
<tr>
<td>11:8</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:12</td>
<td>Physical address of 4-KByte aligned page table referenced by this entry</td>
</tr>
</tbody>
</table>

### Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-KByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PAT)</td>
<td>If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0) [^{1}]</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:12</td>
<td>Physical address of the 4-KByte page referenced by this entry</td>
</tr>
</tbody>
</table>

**NOTES:**
1. See Section 4.1.4 for how to determine whether the PAT is supported.
4.4 PAE PAGING

A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0. PAE paging translates 32-bit linear addresses to 52-bit physical addresses.1 Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

With PAE paging, a logical processor maintains a set of four (4) PDPT registers, which are loaded from an address in CR3. Linear address are translated using 4 hierarchies of in-memory paging structures, each located using one of the PDPT registers. (This is different from the other paging modes, in which there is one hierarchy referenced by CR3.)

Section 4.4.1 discusses the PDPT registers. Section 4.4.2 describes linear-address translation with PAE paging.

4.4.1 PDPT Registers

When PAE paging is used, CR3 references the base of a 32-Byte page-directory-pointer table. Table 4-7 illustrates how CR3 is used with PAE paging.

Table 4-7. Use of CR3 with PAE Paging

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:0</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:5</td>
<td>Physical address of the 32-Byte aligned table</td>
</tr>
<tr>
<td>63:32</td>
<td>Ignored (these bits exist only on processors</td>
</tr>
<tr>
<td></td>
<td>supporting the Intel-64 architecture)</td>
</tr>
</tbody>
</table>

The page-directory-pointer-table comprises four (4) 64-bit entries called PDPT registers. Each PDPT controls access to a 1-GByte region of the linear-address space. Corresponding to the PDPTs, the logical processor maintains a set of four (4) internal, non-architectural PDPT registers, called PDPT0, PDPT1, PDPT2, and PDPT3. The logical processor loads these registers from the PDPTs in memory as part of certain operations:

- If PAE paging would be in use following an execution of MOV to CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is modifying any of CR0.CD, CR0.NW, CR0.PG, CR4.PAE, CR4.PGE, CR4.PSE, or CR4.SMEP; then the PDPTs are loaded from the address in CR3.
- If MOV to CR3 is executed while the logical processor is using PAE paging, the PDPTs are loaded from the address being loaded into CR3.
- If PAE paging is in use and a task switch changes the value of CR3, the PDPTs are loaded from the address in the new CR3 value.
- Certain VMX transitions load the PDPT registers. See Section 4.11.1.

Table 4-8 gives the format of a PDPT. If any of the PDPTs sets both the P flag (bit 0) and any reserved bit, the MOV to CR instruction causes a general-protection exception (#GP(0)) and the PDPTs are not loaded.2 As shown in Table 4-8, bits 2:1, 8:5, and 63:MAXPHYADDR are reserved in the PDPTs.

---

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by PAE paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.
2. On some processors, reserved bits are checked even in PDPTs in which the P flag (bit 0) is 0.
4.4.2 Linear-Address Translation with PAE Paging

PAE paging may map linear addresses to either 4-KByte pages or 2-MByte pages. Figure 4-5 illustrates the translation process when it produces a 4-KByte page; Figure 4-6 covers the case of a 2-MByte page. The following items describe the PAE paging process in more detail as well how the page size is determined:

- Bits 31:30 of the linear address select a PDPTE register (see Section 4.4.1); this is PDPTE\textsubscript{i}, where \(i\) is the value of bits 31:30.1 Because a PDPTE register is identified using bits 31:30 of the linear address, it controls access to a 1-GByte region of the linear-address space. If the P flag (bit 0) of PDPTE\textsubscript{i} is 0, the processor ignores bits 63:1, and there is no mapping for the 1-GByte region controlled by PDPTE\textsubscript{i}. A reference using a linear address in this region causes a page-fault exception (see Section 4.7).

- If the P flag of PDPTE\textsubscript{i} is 1, 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of PDPTE\textsubscript{i} (see Table 4-8 in Section 4.4.1). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 51:12 are from PDPTE\textsubscript{i}.
  - Bits 11:3 are bits 29:21 of the linear address.
  - Bits 2:0 are 0.

Because a PDE is identified using bits 31:21 of the linear address, it controls access to a 2-Mbyte region of the linear-address space. Use of the PDE depends on its PS flag (bit 7):

- If the PDE’s PS flag is 1, the PDE maps a 2-MByte page (see Table 4-9). The final physical address is computed as follows:
  - Bits 51:21 are from the PDE.
  - Bits 20:0 are from the original linear address.

- If the PDE’s PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 4-10). A page table comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDE.

1. With PAE paging, the processor does not use CR3 when translating a linear address (as it does in the other paging modes). It does not access the PDPTEs in the page-directory-pointer table during linear-address translation.

---

**Table 4-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)**

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page directory</td>
</tr>
<tr>
<td>2:1</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>8:5</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>(M-1):12</td>
<td>Physical address of 4-KByte aligned page directory referenced by this entry(^1)</td>
</tr>
<tr>
<td>63:M</td>
<td>Reserved (must be 0)</td>
</tr>
</tbody>
</table>

**NOTES:**
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.
Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-11). The final physical address is computed as follows:

- Bits 51:12 are from the PTE.
- Bits 11:0 are from the original linear address.

If the P flag (bit 0) of a PDE or a PTE is 0 or if a PDE or a PTE sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 4.7).

The following bits are reserved with PAE paging:

- If the P flag (bit 0) of a PDE or a PTE is 1, bits 62:MAXPHYADDR are reserved.
- If the P flag and the PS flag (bit 7) of a PDE are both 1, bits 20:13 are reserved.
- If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.
- If the PAT is not supported: 1
  - If the P flag of a PTE is 1, bit 7 is reserved.
  - If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

---

1. See Section 4.1.4 for how to determine whether the PAT is supported.
Table 4-9. Format of a PAE Page-Directory Entry that Maps a 2-MByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 2-MByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 1 (otherwise, this entry references a page table; see Table 4-10)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>12 (PAT)</td>
<td>If the PAT is supported, indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)¹</td>
</tr>
<tr>
<td>20:13</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>(M-1):21</td>
<td>Physical address of the 2-MByte page referenced by this entry</td>
</tr>
<tr>
<td>62:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>

NOTES:
1. See Section 4.1.4 for how to determine whether the PAT is supported.

¹ This bit is not applicable in the context of Table 4-9.
### Table 4-10. Format of a PAE Page-Directory Entry that References a Page Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page table</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
<tr>
<td>6</td>
<td>Ignored</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 4-9)</td>
</tr>
<tr>
<td>11:8</td>
<td>Ignored</td>
</tr>
<tr>
<td>(M–1):12</td>
<td>Physical address of 4-KByte aligned page table referenced by this entry</td>
</tr>
<tr>
<td>62:63</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>

### Table 4-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-KByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PAT)</td>
<td>If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
</tbody>
</table>
Table 4-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>(M-1):12</td>
<td>Physical address of the 4-KByte page referenced by this entry</td>
</tr>
<tr>
<td>63:63</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>

NOTES:
1. See Section 4.1.4 for how to determine whether the PAT is supported.

Figure 4-7 gives a summary of the formats of CR3 and the paging-structure entries with PAE paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how a paging-structure entry is used.

![Figure 4-7. Formats of CR3 and Paging-Structure Entries with PAE Paging](image)

NOTES:
1. M is an abbreviation for MAXPHYADDR.
2. CR3 has 64 bits only on processors supporting the Intel-64 architecture. These bits are ignored with PAE paging.
3. Reserved fields must be 0.
4. If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.
4.5 4-LEVEL PAGING AND 5-LEVEL PAGING

Because the operation of 4-level paging and 5-level paging is very similar, they are described together in this section. The following items highlight the distinctions between the two paging modes:

- A logical processor uses 4-level paging if CR0.PG = 1, CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 0. 4-level paging translates 48-bit linear addresses to 52-bit physical addresses.\(^1\) Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

- A logical processor uses 5-level paging if CR0.PG = 1, CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 1. 5-level paging translates 57-bit linear addresses to 52-bit physical addresses. Thus, 5-level paging supports a linear-address space sufficient to access the entire physical-address space.

4.5.1 Ordinary Paging and HLAT Paging

There are two forms of 4-level paging and 5-level paging that differ principally with regard to how linear-address translation identifies the first paging structure.

The normal form is called ordinary paging, and it uses CR3 to locate the first paging structure, similar to what is done for 32-bit paging. Section 4.5.2 provides details of this use of CR3.

An alternative form of paging may be used with the VMX feature called hypervisor-managed linear-address translation (HLAT). Called HLAT paging, this form is used only in VMX non-root operation and only if the “enable HLAT” VM-execution control is 1.\(^2\) HLAT paging locates the first paging structure using a VM-execution control field in the VMCS called the HLAT pointer (HLATP). Section 4.5.3 provides details.

Whether HLAT paging is used to translate a specific linear address depends on the address and on the value of a VM-execution control field in the VMCS called the HLAT prefix size:

- If the HLAT prefix size is zero, every linear address is translated using HLAT paging.
- If the HLAT prefix size is not zero, a linear address is translated using HLAT paging if bit 63 of the address is 1.\(^3\) The address is translated using ordinary paging if bit 63 of the address is 0.

In some cases, HLAT paging may specify that a translation of a linear address must be restarted. When this occurs, the linear address is then translated using ordinary paging (starting with a paging structure identified using CR3). The situations leading to this restart are detailed in Section 4.5.4, and additional details of the restart process are given in Section 4.5.5.

4.5.2 Use of CR3 with Ordinary 4-Level Paging and 5-Level Paging

Ordinary 4-level paging and 5-level paging each translate linear addresses using a hierarchy of in-memory paging structures located using the contents of CR3, which is used to locate the first paging structure. For 4-level paging, this is the PML4 table, and for 5-level paging it is the PML5 table. Use of CR3 with 4-level paging and 5-level paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

- Table 4-12 illustrates how CR3 is used with 4-level paging and 5-level paging if CR4.PCIDE = 0.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by 4-level paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.
2. HLAT paging is used only with 4-level paging and 5-level paging. It is never used with 32-bit paging or PAE paging, regardless of the value of the “enable HLAT” VM-execution control.
3. This behavior applies if the CPU enumerates a maximum HLAT prefix size of 1 in IA32_VMX_EPT_VPID_CAP[53:48] (see Appendix A.10). Behavior when a different value is enumerated is not currently defined.
Table 4-12. Use of CR3 with 4-Level Paging and 5-level Paging and CR4.PCIDE = 0 (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the PML4 table or PML5 table during linear-address translation (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the PML4 table or PML5 table during linear-address translation (see Section 4.9.2)</td>
</tr>
<tr>
<td>11:5</td>
<td>Ignored</td>
</tr>
<tr>
<td>M-1:12</td>
<td>Physical address of the 4-KByte aligned PML4 table or PML5 table used for linear-address translation¹</td>
</tr>
<tr>
<td>63:M</td>
<td>Reserved (must be 0)</td>
</tr>
</tbody>
</table>

NOTES:
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

- Table 4-13 illustrates how CR3 is used with 4-level paging and 5-level paging if CR4.PCIDE = 1.

Table 4-13. Use of CR3 with 4-Level Paging and 5-Level Paging and CR4.PCIDE = 1

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:0</td>
<td>PCID (see Section 4.10.1)¹</td>
</tr>
<tr>
<td>M-1:12</td>
<td>Physical address of the 4-KByte aligned PML4 table used for linear-address translation²</td>
</tr>
<tr>
<td>63:M</td>
<td>Reserved (must be 0)³</td>
</tr>
</tbody>
</table>

NOTES:
1. Section 4.9.2 explains how the processor determines the memory type used to access the PML4 table during linear-address translation with CR4.PCIDE = 1.
2. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.
3. See Section 4.10.4.1 for use of bit 63 of the source operand of the MOV to CR3 instruction.

After software modifies the value of CR4.PCIDE, the logical processor immediately begins using CR3 as specified for the new value. For example, if software changes CR4.PCIDE from 1 to 0, the current PCID immediately changes from CR3[11:0] to 000H (see also Section 4.10.4.1). In addition, the logical processor subsequently determines the memory type used to access the PML4 table using CR3.PWT and CR3.PCD, which had been bits 4:3 of the PCID.

4.5.3 Use of HLATP with HLAT 4-Level Paging and 5-Level Paging

With HLAT paging, 4-level paging and 5-level paging each translate linear addresses using a hierarchy of in-memory paging structures located using the value of HLATP (a VM-execution control field in the VMCS), which is used to locate the first paging structure. For 4-level paging, this is the PML4 table, and for 5-level paging it is the PML5 table.

HLATP has the same format as that given for CR3 in Table 4-12, with the exception that bits 2:0 and bits 11:5 are reserved and must be zero (these bits are checked by VM entry). HLATP does not contain a PCID value. HLATP paging with CR4.PCIDE = 1 uses the PCID value in CR3[11:0].
4.5.4 Linear-Address Translation with 4-Level Paging and 5-Level Paging

4-level paging and 5-level paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.\(^1\) Figure 4-8 illustrates the translation process for 4-level paging when it produces a 4-KByte page; Figure 4-9 covers the case of a 2-MByte page, and Figure 4-10 the case of a 1-GByte page. (The process for 5-level paging is similar.)

---

1. Not all processors support 1-GByte pages; see Section 4.1.4.
Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging
4-level paging and 5-level paging associate with each linear address a **protection key**. Section 4.6 explains how
the processor uses the protection key in its determination of the access rights of each linear address.

The remainder of this section describes the translation process used by 4-level paging and 5-level paging in more
detail, as well as how the page size and protection key are determined. Because the process used by the two
paging modes is similar, they are described together, with any differences identified, in the following items:

- With 5-level paging, a 4-KByte naturally aligned PML5 table is located at the physical address specified in
  bits 51:12 of CR3 (see Table 4-12). (4-level paging does not use a PML5 table and omits this step.) A PML5
table comprises 512 64-bit entries (PML5Es). A PML5E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3 or HLATP.
  - Bits 11:3 are bits 56:48 of the linear address.
  - Bits 2:0 are all 0.

Because a PML5E is identified using bits 56:48 of the linear address, it controls access to a 256-TByte region of
the linear-address space.

With HLAT paging, if bit 11 of the PML5E is 1, translation is restarted with ordinary paging with a maximum
page size of 256-TBytes (see Section 4.5.5). Otherwise, the translation process continues as described in the
next item.

- A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (for 4-
  level paging; see Table 4-12) or in bits 51:12 of the PML5E (for 5-level paging; see Table 4-14). A PML4 table
comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3 or the HLATP (for 4-level paging) or in bits 51:12 of the PML5E (for 5-level
    paging).
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.

Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of
the linear-address space.

With HLAT paging, if bit 11 of the PML4E is 1, translation is restarted with ordinary paging with a maximum
page size of 512-GBytes (see Section 4.5.5). Otherwise, the translation process continues as described in the
next item.

- A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in
  bits 51:12 of the PML4E (see Table 4-15). A page-directory-pointer table comprises 512 64-bit entries
(PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.
  - Bits 11:3 are bits 38:30 of the linear address.
  - Bits 2:0 are all 0.

Because a PDPTE is identified using bits 47:30 of the linear address, it controls access to a 1-GByte region of the
linear-address space.

With HLAT paging, if bit 11 of the PDPTE is 1, translation is restarted with ordinary paging with a maximum page
size of 1-GByte (see Section 4.5.5). Otherwise, the translation process continues as described below.

Use of the PDPTE depends on its PS flag (bit 7):\(^1\)

- If the PDPTE’s PS flag is 1, the PDPTE maps a 1-GByte page (see Table 4-16). The final physical address is
  computed as follows:
    - Bits 51:30 are from the PDPTE.
    - Bits 29:0 are from the original linear address.

  The linear address’s protection key is the value of bits 62:59 of the PDPTE (see Section 4.6.2).

---

\(^1\) The PS flag of a PDPTE is reserved and must be 0 (if the P flag is 1) if 1-GByte pages are not supported. See Section 4.1.4 for how
to determine whether 1-GByte pages are supported.
If the PDPTE’s PS flag is 0, a 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of the PDPTE (see Table 4-17). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
- Bits 51:12 are from the PDPTE.
- Bits 11:3 are bits 29:21 of the linear address.
- Bits 2:0 are all 0.

Because a PDE is identified using bits 47:21 of the linear address, it controls access to a 2-MByte region of the linear-address space.

With HLAT paging, if bit 11 of the PDE is 1, translation is restarted with ordinary paging with a maximum page size of 2-MBytes (see Section 4.5.5). Otherwise, the translation process continues as described below.

Use of the PDE depends on its PS flag:

- If the PDE’s PS flag is 1, the PDE maps a 2-MByte page (see Table 4-18). The final physical address is computed as follows:
  - Bits 51:21 are from the PDE.
  - Bits 20:0 are from the original linear address.
  
  The linear address’s protection key is the value of bits 62:59 of the PDE (see Section 4.6.2).

- If the PDE’s PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 4-19). A page table comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDE.
  - Bits 11:3 are bits 20:12 of the linear address.
  - Bits 2:0 are all 0.

- Because a PTE is identified using bits 47:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-20).

  With HLAT paging, if bit 11 of the PTE is 1, translation is restarted with ordinary paging with a maximum page size of 4-KBytes (see Section 4.5.5). Otherwise, the final physical address is computed as follows:
  - Bits 51:12 are from the PTE.
  - Bits 11:0 are from the original linear address.
  
  The linear address’s protection key is the value of bits 62:59 of the PTE (see Section 4.6.2).

If a paging-structure entry’s P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 4.7).

The following bits in a paging-structure entry are reserved with 4-level paging and 5-level paging (assuming that the entry’s P flag is 1):

- Bits 51:MAXPHYADDR are reserved in every paging-structure entry.
- The PS flag is reserved in a PMLSE or a PML4E.
- If 1-GByte pages are not supported, the PS flag is reserved in a PDPTE.¹
- If the PS flag in a PDPTE is 1, bits 29:13 of the entry are reserved.
- If the PS flag in a PDE is 1, bits 20:13 of the entry are reserved.
- If IA32_EFER.NXE = 0, the XD flag (bit 63) is reserved in every paging-structure entry.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

¹. See Section 4.1.4 for how to determine whether 1-GByte pages are supported.
Figure 4-11 gives a summary of the formats of CR3 and the 4-level and 5-level paging-structure entries. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do not because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how a paging-structure entry is used.

Table 4-14. Format of a PML5 Entry (PML5E) that References a PML4 Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a PML4 table</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 256-TByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 256-TByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the PML4 table referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the PML4 table referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
<tr>
<td>6</td>
<td>Ignored</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>10:8</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>M–1:12</td>
<td>Physical address of 4-KByte aligned PML4 table referenced by this entry</td>
</tr>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>62:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 256-TByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>

Table 4-15. Format of a PML4 Entry (PML4E) that References a Page-Directory-Pointer Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page-directory-pointer table</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 512-GByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 512-GByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
<tr>
<td>6</td>
<td>Ignored</td>
</tr>
</tbody>
</table>
### Table 4-15. Format of a PML4 Entry (PML4E) that References a Page-Directory-Pointer Table (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 (PS)</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>10:8</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>M-1:12</td>
<td>Physical address of 4-KByte aligned page-directory-pointer table referenced by this entry</td>
</tr>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>62:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 512-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>

### Table 4-16. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 1-GByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 1-GByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 1-GByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 1-GByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 1 (otherwise, this entry references a page directory; see Table 4-17)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>10:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>12 (PAT)</td>
<td>Indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>29:13</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>(M-1):30</td>
<td>Physical address of the 1-GByte page referenced by this entry</td>
</tr>
</tbody>
</table>
### Table 4-16. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>58:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>62:59</td>
<td>Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page’s access rights (see Section 4.6.2); otherwise, it is ignored and not used to control access rights.</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The PAT is supported on all processors that support 4-level paging.

### Table 4-17. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page directory</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 1-GByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
<tr>
<td>6</td>
<td>Ignored</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 0 (otherwise, this entry maps a 1-GByte page; see Table 4-16)</td>
</tr>
<tr>
<td>10:8</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>(M–1):12</td>
<td>Physical address of 4-KByte aligned page directory referenced by this entry</td>
</tr>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>62:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
<tr>
<td>Bit Position(s)</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 2-MByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 1 (otherwise, this entry references a page table; see Table 4-19)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>10:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>12 (PAT)</td>
<td>Indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>20:13</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>(M–1):21</td>
<td>Physical address of the 2-MByte page referenced by this entry</td>
</tr>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>58:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>62:59</td>
<td>Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 4.6.2); otherwise, it is ignored and not used to control access rights.</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>
Table 4-19. Format of a Page-Directory Entry that References a Page Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page table</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
<tr>
<td>6</td>
<td>Ignored</td>
</tr>
<tr>
<td>7 (PS)</td>
<td>Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 4-18)</td>
</tr>
<tr>
<td>10:8</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>(M-1):12</td>
<td>Physical address of 4-KByte aligned page table referenced by this entry</td>
</tr>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>62:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>
### Table 4-20. Format of a Page-Table Entry that Maps a 4-KByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-KByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PAT)</td>
<td>Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>10:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>11 (R)</td>
<td>For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)</td>
</tr>
<tr>
<td>(M–1):12</td>
<td>Physical address of the 4-KByte page referenced by this entry</td>
</tr>
<tr>
<td>51:M</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>58:52</td>
<td>Ignored</td>
</tr>
<tr>
<td>62:59</td>
<td>Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page’s access rights (see Section 4.6.2); otherwise, it is ignored and not used to control access rights.</td>
</tr>
<tr>
<td>63 (XD)</td>
<td>If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)</td>
</tr>
</tbody>
</table>
NOTES:
1. M is an abbreviation for MAXPHYADDR.
2. Reserved fields must be 0.
3. If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.
4. Bit 11 is R (restart) only for HLAT paging; it is ignored for ordinary paging.
5. The protection key is used only if software has enabled the appropriate feature; see Section 4.6.2. It is ignored otherwise.

**Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging**

<table>
<thead>
<tr>
<th>XD</th>
<th>Ignored</th>
<th>Rsvd.</th>
<th>Address of PML4 table</th>
<th>Ignored</th>
<th>Rsvd.</th>
<th>Address of PML5 table</th>
<th>Ignored</th>
<th>Rsvd.</th>
<th>Address of page-directory-pointer table</th>
<th>Ignored</th>
<th>Rsvd.</th>
<th>Address of page directory</th>
<th>Ignored</th>
<th>Rsvd.</th>
<th>Address of page frame</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **PML5E**: present
- **PML5E**: not present
- **PML4E**: present
- **PML4E**: not present
- **PDPTE**: 1GB page
- **PDPTE**: page directory
- **PDE**: 2MB page
- **PDE**: page table
- **PTE**: 4KB page
4.5.5  Restart of HLAT Paging

As noted in Section 4.5.1, HLAT paging may specify that a translation of a linear address must be restarted. Specifically, this occurs when HLAT paging encounters a paging-structure entry that sets bit 11 (see Section 4.5.4). When this occurs, translation of the linear address is restarted using ordinary paging (starting with a paging-structure identified using CR3). The restarted translation proceeds just as if the HLAT feature were not enabled. The entire linear address is translated again, including those portions that had been used by HLAT paging prior to the restart.

The process of restarting HLAT paging (using ordinary paging) always specifies a maximum page size to be used when a resulting translation is cached in the TLBs. This maximum page size depends on the level of the paging-structure entry that restarts the translation by setting bit 11; details are given in Section 4.5.4. The page size of the translation produced by the restarted process is never greater than this maximum page size. See Section 4.10.2.2 for more discussion.

4.6  ACCESS RIGHTS

There is a translation for a linear address if the processes described in Section 4.3, Section 4.4.2, and Section 4.5 (depending upon the paging mode) completes and produces a physical address. Whether an access is permitted by a translation is determined by the access rights specified by the paging-structure entries controlling the translation;¹ paging-mode modifiers in CR0, CR4, and the IA32_EFER MSR; EFLAGS.AC; and the mode of the access.

Section 4.6.1 describes how the processor determines the access rights for each linear address. Section 4.6.2 provides additional information about how protection keys contribute to access-rights determination. (They do so only with 4-level paging and 5-level paging, and only if CR4.PKE = 1 or CR4.PKS = 1.)

NOTE

If HLAT paging is restarted, permissions are determined only by the access rights specified by the paging-structure entries that the subsequent ordinary paging used to translate the linear address. The access rights specified by the entries used earlier by HLAT paging do not apply.

4.6.1  Determination of Access Rights

Every access to a linear address is either a supervisor-mode access or a user-mode access. For all instruction fetches and most data accesses, this distinction is determined by the current privilege level (CPL): accesses made while CPL < 3 are supervisor-mode accesses, while accesses made while CPL = 3 are user-mode accesses.

Some operations implicitly access system data structures with linear addresses; the resulting accesses to those data structures are supervisor-mode accesses regardless of CPL. Examples of such accesses include the following: accesses to the global descriptor table (GDT) or local descriptor table (LDT) to load a segment descriptor; accesses to the interrupt descriptor table (IDT) when delivering an interrupt or exception; and accesses to the task-state segment (TSS) as part of a task switch or change of CPL. All these accesses are called implicit supervisor-mode accesses regardless of CPL. Other accesses made while CPL < 3 are called explicit supervisor-mode accesses.

Access rights are also controlled by the mode of a linear address as specified by the paging-structure entries controlling the translation of the linear address. If the U/S flag (bit 2) is 0 in at least one of the paging-structure entries, the address is a supervisor-mode address. Otherwise, the address is a user-mode address.

When the shadow-stack feature of control-flow enforcement technology (CET) is enabled, certain accesses to linear addresses are considered shadow-stack accesses (see Section 17.2, “Shadow Stacks,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Like ordinary data accesses, each shadow-stack access is defined as being either a user access or a supervisor access. In general, a shadow-stack access is a user access if CPL = 3 and a supervisor access if CPL < 3. The WRUSS instruction is an exception; although it can be executed only if CPL = 0, the processor treats its shadow-stack accesses as user accesses.

¹. With PAE paging, the PDPTEs do not determine access rights.
Shadow-stack accesses are allowed only to **shadow-stack addresses**. A linear address is a shadow-stack address if the following are true of the translation of the linear address: (1) the R/W flag (bit 1) is 0 and the dirty flag (bit 6) is 1 in the paging-structure entry that maps the page containing the linear address; and (2) the R/W flag is 1 in every other paging-structure entry controlling the translation of the linear address.

The following items detail how paging determines access rights (only the items noted explicitly apply to shadow-stack accesses):

**NOTE**

Many of the items below refer to an address with a protection key for which read (or write) access is permitted. Section 4.6.2 provides details on when a protection key will permit (or not permit) a data access (read or write) to a linear address using that protection key.

- **For supervisor-mode accesses:**
  - Data may be read (implicitly or explicitly) from any supervisor-mode address with a protection key for which read access is permitted.
  - Data reads from user-mode pages.
    Access rights depend on the value of CR4.SMAP:
    - If CR4.SMAP = 0, data may be read from any user-mode address with a protection key for which read access is permitted.
    - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
      - If EFLAGS.AC = 1 and the access is explicit, data may be read from any user-mode address with a protection key for which read access is permitted.
      - If EFLAGS.AC = 0 or the access is implicit, data may not be read from any user-mode address.
  - Data writes to supervisor-mode addresses.
    Access rights depend on the value of CR0.WP:
    - If CR0.WP = 0, data may be written to any supervisor-mode address with a protection key for which write access is permitted.
    - If CR0.WP = 1, data may be written to any supervisor-mode address with a translation for which the R/W flag (bit 1) is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any supervisor-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
  - Data writes to user-mode addresses.
    Access rights depend on the value of CR0.WP:
    - If CR0.WP = 0, access rights depend on the value of CR4.SMAP:
      - If CR4.SMAP = 0, data may be written to any user-mode address with a protection key for which write access is permitted.
      - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
        - If EFLAGS.AC = 1 and the access is explicit, data may be written to any user-mode address with a protection key for which write access is permitted.
        - If EFLAGS.AC = 0 or the access is implicit, data may not be written to any user-mode address.
    - If CR0.WP = 1, access rights depend on the value of CR4.SMAP:
      - If CR4.SMAP = 0, data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any user-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
      - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
• If EFLAGS.AC = 1 and the access is explicit, data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any user-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.

• If EFLAGS.AC = 0 or the access is implicit, data may not be written to any user-mode address.

— Instruction fetches from supervisor-mode addresses.
  • For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any supervisor-mode address.
  • For other paging modes with IA32_EFER.NXE = 1, instructions may be fetched from any supervisor-mode address with a translation for which the XD flag (bit 63) is 0 in every paging-structure entry controlling the translation; instructions may not be fetched from any supervisor-mode address with a translation for which the XD flag is 1 in any paging-structure entry controlling the translation.

— Instruction fetches from user-mode addresses.
  Access rights depend on the values of CR4.SMEP:
  • If CR4.SMEP = 0, access rights depend on the paging mode and the value of IA32_EFER.NXE:
    — For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any user-mode address.
    — For other paging modes with IA32_EFER.NXE = 1, instructions may be fetched from any user-mode address with a translation for which the XD flag is 0 in every paging-structure entry controlling the translation; instructions may not be fetched from any user-mode address with a translation for which the XD flag is 1 in any paging-structure entry controlling the translation.
  • If CR4.SMEP = 1, instructions may not be fetched from any user-mode address.

— Supervisor-mode shadow-stack accesses are allowed only to supervisor-mode shadow-stack addresses (see above).

• For user-mode accesses:
  — Data reads.
    Access rights depend on the mode of the linear address:
    • Data may be read from any user-mode address with a protection key for which read access is permitted.
    • Data may not be read from any supervisor-mode address.
  — Data writes.
    Access rights depend on the mode of the linear address:
    • Data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted.
    • Data may not be written to any supervisor-mode address.
  — Instruction fetches.
    Access rights depend on the mode of the linear address, the paging mode, and the value of IA32_EFER.NXE:
    • For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any user-mode address.
    • For other paging modes with IA32_EFER.NXE = 1, instructions may be fetched from any user-mode address with a translation for which the XD flag is 0 in every paging-structure entry controlling the translation.
    • Instructions may not be fetched from any supervisor-mode address.
  — User-mode shadow-stack accesses made outside enclave mode are allowed only to user-mode shadow-stack addresses (see above). User-mode shadow-stack accesses made in enclave mode are treated like ordinary data accesses (see above).
A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). These structures may include information about access rights. The processor may enforce access rights based on the TLBs and paging-structure caches instead of on the paging structures in memory.

This fact implies that, if software modifies a paging-structure entry to change access rights, the processor might not use that change for a subsequent access to an affected linear address (see Section 4.10.4.3). See Section 4.10.4.2 for how software can ensure that the processor uses the modified access rights.

4.6.2 Protection Keys

4-level paging and 5-level paging associate a 4-bit protection key with each linear address (the protection key located in bits 62:59 of the paging-structure entry that mapped the page containing the linear address; see Section 4.5). Two protection key features control accesses to linear addresses based on their protection keys:

- If CR4.PKE = 1, the PKRU register determines, for each protection key, whether user-mode addresses with that protection key may be read or written.
- If CR4.PKS = 1, the IA32_PKRS MSR (MSR index 6E1H) determines, for each protection key, whether supervisor-mode addresses with that protection key may be read or written.

32-bit paging and PAE paging do not associate linear addresses with protection keys. For the purposes of Section 4.6.1, reads and writes are implicitly permitted for all protection keys with either of those paging modes.

The PKRU register (protection-key rights for user pages) is a 32-bit register with the following format: for each \( i \) (\( 0 \leq i \leq 15 \)), \( \text{PKRU}[2i] \) is the access-disable bit for protection key \( i \) (AD\(_i\)); \( \text{PKRU}[2i+1] \) is the write-disable bit for protection key \( i \) (WD\(_i\)). The IA32_PKRS MSR has the same format (bits 63:32 of the MSR are reserved and must be zero).

Software can use the RDPKRU and WRPKRU instructions with ECX = 0 to read and write PKRU. In addition, the PKRU register is XSAVE-managed state and can thus be read and written by instructions in the XSAVE feature set. See Chapter 13, “Managing State Using the XSAVE Feature Set,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the XSAVE feature set.

Software can use the RDMSR and WRMSR instructions to read and write the IA32_PKRS MSR. Writes to the IA32_PKRS MSR using WRMSR are not serializing. The IA32_PKRS MSR is not XSAVE-managed.

How a linear address’s protection key controls access to the address depends on the mode of the linear address:

- A linear address’s protection key controls only data accesses to the address. It does not in any way affect instructions fetches from the address.
- If CR4.PKE = 0, the protection key of a user-mode address does not control data accesses to the address (for the purposes of Section 4.6.1, reads and writes of user-mode addresses are implicitly permitted for all protection keys).

If CR4.PKE = 1, use of the protection key \( i \) of a user-mode address depends on the value of the PKRU register:

- If AD\(_i\) = 1, no data accesses are permitted.
- If WD\(_i\) = 1, permission may be denied to certain data write accesses:
  - User-mode write accesses are not permitted.
  - Supervisor-mode write accesses are not permitted if CR0.WP = 1. (If CR0.WP = 0, WD\(_i\) does not affect supervisor-mode write accesses to user-mode addresses with protection key \( i \).)

- If CR4.PKS = 0, the protection key of a supervisor-mode address does not control data accesses to the address (for the purposes of Section 4.6.1, reads and writes of supervisor-mode addresses are implicitly permitted for all protection keys).

If CR4.PKS = 1, use of the protection key \( i \) of a supervisor-mode address depends on the value of the IA32_PKRS MSR:

- If AD\(_i\) = 1, no data accesses are permitted.
- If WD\(_i\) = 1, write accesses are not permitted if CR0.WP = 1. (If CR0.WP = 0, IA32_PKRS.WD\(_i\) does not affect write accesses to supervisor-mode addresses with protection key \( i \).)

Protection keys apply to shadow-stack accesses just as they do to ordinary data accesses.
4.7 PAGE-FAULT EXCEPTIONS

Accesses using linear addresses may cause page-fault exceptions (#PF; exception 14). An access to a linear address may cause a page-fault exception for either of two reasons: (1) there is no translation for the linear address; or (2) there is a translation for the linear address, but its access rights do not permit the access.

As noted in Section 4.3, Section 4.4.2, and Section 4.5, there is no translation for a linear address if the translation process for that address would use a paging-structure entry in which the P flag (bit 0) is 0 or one that sets a reserved bit.\(^1\) If there is a translation for a linear address, its access rights are determined as specified in Section 4.6.

When Intel® Software Guard Extensions (Intel® SGX) are enabled, the processor may deliver exception 14 for reasons unrelated to paging. See Section 35.3, “Access-control Requirements,” and Section 35.20, “Enclave Page Cache Map (EPCM),” in Chapter 35, “Enclave Access Control and Data Structures.” Such an exception is called an SGX-induced page fault. The processor uses the error code to distinguish SGX-induced page faults from ordinary page faults.

Figure 4-12 illustrates the error code that the processor provides on delivery of a page-fault exception. The following items explain how the bits in the error code describe the nature of the page-fault exception:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>RSVD</td>
</tr>
<tr>
<td>9</td>
<td>RSVD</td>
</tr>
<tr>
<td>8</td>
<td>RSVD</td>
</tr>
<tr>
<td>7</td>
<td>RSVD</td>
</tr>
<tr>
<td>6</td>
<td>W/R</td>
</tr>
<tr>
<td>5</td>
<td>U/S</td>
</tr>
<tr>
<td>4</td>
<td>P</td>
</tr>
<tr>
<td>3</td>
<td>I/D</td>
</tr>
<tr>
<td>2</td>
<td>PK</td>
</tr>
<tr>
<td>1</td>
<td>SS</td>
</tr>
<tr>
<td>0</td>
<td>HLAT</td>
</tr>
<tr>
<td></td>
<td>SGX</td>
</tr>
</tbody>
</table>

- **P flag (bit 0).** This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.

1. If HLAT paging encounters a paging-structure entry that sets a reserved bit, there is no translation even if the bit 11 of the entry indicates a restart. In this case, there is a page fault and the translation is not restarted.
• **W/R** (bit 1).  
If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

• **U/S** (bit 2).  
If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging. User-mode and supervisor-mode accesses are defined in Section 4.6.

• **RSVD flag** (bit 3).  
This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address. (Because reserved bits are not checked in a paging-structure entry whose P flag is 0, bit 3 of the error code can be set only if bit 0 is also set.) Bits reserved in the paging-structure entries are reserved for future functionality. Software developers should be aware that such bits may be used in the future and that a paging-structure entry that causes a page-fault exception on one processor might not do so in the future.

• **I/D** flag (bit 4).  
This flag is 1 if (1) the access causing the page-fault exception was an instruction fetch; and (2) either (a) CR4.SMEP = 1; or (b) both (i) CR4.PAE = 1 (either PAE paging, 4-level paging, or 5-level paging is in use); and (ii) IA32_EFER.NXE = 1. Otherwise, the flag is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

• **PK flag** (bit 5).  
This flag is 1 only for data accesses and only with 4-level paging and 5-level paging. In these cases, the setting depends on the mode of the address being accessed:

  — For accesses to supervisor-mode addresses, the flag is set if (1) CR4.PKS = 1; (2) the linear address has protection key \(i\); and (3) the IA32_PKRS MSR (see Section 4.6.2) is such that either (a) \(\text{AD}i = 1\); or (b) the following all hold: (i) \(\text{WD}i = 1\); (ii) the access is a write access; and (iii) either CR0.WP = 1 or the access causing the page-fault exception was a user-mode access. (Note that this flag may be set on page faults due to user-mode accesses to supervisor-mode addresses.)

  — For accesses to user-mode addresses, the flag is set if (1) CR4.PKE = 1; (2) the linear address has protection key \(i\); and (3) the PKRU register (see Section 4.6.2) is such that either (a) \(\text{AD}i = 1\); or (b) the following all hold: (i) \(\text{WD}i = 1\); (ii) the access is a write access; and (iii) either CR0.WP = 1 or the access causing the page-fault exception was a user-mode access.

• **SS** (bit 6).  
If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

• **HLAT** (bit 7).  
This flag is 1 if there is no translation for the linear address using HLAT paging because, in one of the paging-structure entries used to translate that address, either the P flag was 0 or a reserved bit was set. An error code will set this flag only if it clears bit 0 or sets bit 3. This flag will not be set by a page fault resulting from a violation of access rights, nor for one encountered during ordinary paging, including the case in which there has been a restart of HLAT paging.

• **SGX flag** (bit 15).  
This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

Page-fault exceptions occur only due to an attempt to use a linear address. Failures to load the PDPT registers with PAE paging (see Section 4.4.1) cause general-protection exceptions (#GP(0)) and not page-fault exceptions.

---

1. Some past processors had errata for some page faults that occur when there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address. Due to these errata, some such page faults produced error codes that cleared bit 0 (P flag) and set bit 3 (RSVD flag).
4.8 ACCESSED AND DIRTY FLAGS

For any paging-structure entry that is used during linear-address translation, bit 5 is the **accessed** flag. For paging-structure entries that map a page (as opposed to referencing another paging structure), bit 6 is the **dirty** flag. These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Whenever the processor uses a paging-structure entry as part of linear-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a linear address, the processor sets the dirty flag (if it is not already set) in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

The previous two paragraphs apply also to HLAT paging. If HLAT paging encounters a paging-structure entry that sets bit 11, indicating a restart, the processor will set the accessed flag in that entry; it will not set the dirty flag because, if an entry indicates a restart, it does identify the final physical address for the linear address being translated.

NOTE

If software on one logical processor writes to a page while software on another logical processor concurrently clears the R/W flag in the paging-structure entry that maps the page, execution on some processors may result in the entry’s dirty flag being set (due to the write on the first logical processor) and the entry’s R/W flag being clear (due to the update to the entry on the second logical processor). This will never occur on a processor that supports control-flow enforcement technology (CET). Specifically, a processor that supports CET will never set the dirty flag in a paging-structure entry in which the R/W flag is clear.

Memory-management software may clear these flags when a page or a paging structure is initially loaded into physical memory. These flags are “sticky,” meaning that, once set, the processor does not clear them; only software can clear them.

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the processor might not set the corresponding bit in memory on a subsequent access using an affected linear address (see Section 4.10.4.3). See Section 4.10.4.2 for how software can ensure that these bits are updated as desired.

NOTE

The accesses used by the processor to set these flags may or may not be exposed to the processor’s self-modifying code detection logic. If the processor is executing code from the same memory area that is being used for the paging structures, the setting of these flags may or may not result in an immediate change to the executing code stream.

4.9 PAGING AND MEMORY TYPING

The **memory type** of a memory access refers to the type of caching used for that access. Chapter 12, “Memory Cache Control,” provides many details regarding memory typing in the Intel-64 and IA-32 architectures. This section describes how paging contributes to the determination of memory typing.

The way in which paging contributes to memory typing depends on whether the processor supports the **Page Attribute Table (PAT)**; see Section 12.12). Section 4.9.1 and Section 4.9.2 explain how paging contributes to memory typing depending on whether the PAT is supported.

---

1. With PAE paging, the PDPTEs are not used during linear-address translation but only to load the PDPTE registers for some executions of the MOV CR instruction (see Section 4.4.1). For this reason, the PDPTEs do not contain accessed flags with PAE paging.

2. The PAT is supported on Pentium III and more recent processor families. See Section 4.1.4 for how to determine whether the PAT is supported.
4.9.1 Paging and Memory Typing When the PAT is Not Supported (Pentium Pro and Pentium II Processors)

**NOTE**
The PAT is supported on all processors that support 4-level paging or 5-level paging. Thus, this section applies only to 32-bit paging and PAE paging.

If the PAT is not supported, paging contributes to memory typing in conjunction with the memory-type range registers (MTRRs) as specified in Table 12-6 in Section 12.5.2.1.

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a PCD value and a PWT value. The latter two values are determined as follows:

- For an access to a PDE with 32-bit paging, the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, the PCD and PWT values come from the relevant PDPTE register.
- For an access to a PTE, the PCD and PWT values come from the relevant PDE.
- For an access to the physical address that is the translation of a linear address, the PCD and PWT values come from the relevant PTE (if the translation uses a 4-KByte page) or the relevant PDE (otherwise).
- With PAE paging, the UC memory type is used when loading the PDPTEs (see Section 4.4.1).

4.9.2 Paging and Memory Typing When the PAT is Supported (Pentium III and More Recent Processor Families)

If the PAT is supported, paging contributes to memory typing in conjunction with the PAT and the memory-type range registers (MTRRs) as specified in Table 12-7 in Section 12.5.2.2.

The PAT is a 64-bit MSR (IA32_PAT; MSR index 277H) comprising eight (8) 8-bit entries (entry \(i\) comprises bits \(8i+7:8i\) of the MSR).

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Table 12-11 in Section 12.12.3 specifies how a memory type is selected from the PAT. Specifically, it comes from entry \(i\) of the PAT, where \(i\) is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with 4-level paging):
  - For 4-level paging or 5-level paging with CR4.PCIDE = 1, \(i = 0\).
  - Otherwise, \(i = 2 \times \text{PCD} + \text{PWT}\), where the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, \(i = 2 \times \text{PCD} + \text{PWT}\), where the PCD and PWT values come from the relevant PDPTE register.
- For an access to a paging-structure entry X whose address is in another paging-structure entry Y, \(i = 2 \times \text{PCD} + \text{PWT}\), where the PCD and PWT values come from Y.
- For an access to the physical address that is the translation of a linear address, \(i = 4 \times \text{PAT} + 2 \times \text{PCD} + \text{PWT}\), where the PAT, PCD, and PWT values come from the relevant PTE (if the translation uses a 4-KByte page), the relevant PDE (if the translation uses a 2-MByte page or a 4-MByte page), or the relevant PDPTE (if the translation uses a 1-GByte page).
- With PAE paging, the WB memory type is used when loading the PDPTEs (see Section 4.4.1).¹

---

¹ Some older IA-32 processors used the UC memory type when loading the PDPTEs. Some processors may use the UC memory type if CR0.CD = 1 or if the MTRRs are disabled. These behaviors are model-specific and not architectural.
4.9.3 Caching Paging-Related Information about Memory Typing

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). These structures may include information about memory typing. The processor may use memory-typing information from the TLBs and paging-structure caches instead of from the paging structures in memory. This fact implies that, if software modifies a paging-structure entry to change the memory-typing bits, the processor might not use that change for a subsequent translation using that entry or for access to an affected linear address. See Section 4.10.4.2 for how software can ensure that the processor uses the modified memory typing.

4.10 CACHING TRANSLATION INFORMATION

The Intel-64 and IA-32 architectures may accelerate the address-translation process by caching data from the paging structures on the processor. Because the processor does not ensure that the data that it caches are always consistent with the structures in memory, it is important for software developers to understand how and when the processor may cache such data. They should also understand what actions software can take to remove cached data that may be inconsistent and when it should do so. This section provides software developers information about the relevant processor operation.

Section 4.10.1 introduces process-context identifiers (PCIDs), which a logical processor may use to distinguish information cached for different linear-address spaces. Section 4.10.2 and Section 4.10.3 describe how the processor may cache information in translation lookaside buffers (TLBs) and paging-structure caches, respectively. Section 4.10.4 explains how software can remove inconsistent cached information by invalidating portions of the TLBs and paging-structure caches. Section 4.10.5 describes special considerations for multiprocessor systems.

4.10.1 Process-Context Identifiers (PCIDs)

Process-context identifiers (PCIDs) are a facility by which a logical processor may cache information for multiple linear-address spaces. The processor may retain cached information when software switches to a different linear-address space with a different PCID (e.g., by loading CR3; see Section 4.10.4.1 for details).

A PCID is a 12-bit identifier. Non-zero PCIDs are enabled by setting the PCIDE flag (bit 17) of CR4. If CR4.PCIDE = 0, the current PCID is always 000H; otherwise, the current PCID is the value of bits 11:0 of CR3.1 Not all processors allow CR4.PCIDE to be set to 1; see Section 4.1.4 for how to determine whether this is allowed.

The processor ensures that CR4.PCIDE can be 1 only in IA-32e mode (thus, 32-bit paging and PAE paging use only PCID 000H). In addition, software can change CR4.PCIDE from 0 to 1 only if CR3[11:0] = 000H. These requirements are enforced by the following limitations on the MOV CR instruction:

- MOV to CR4 causes a general-protection exception (#GP) if it would change CR4.PCIDE from 0 to 1 and either IA32_EFER.LMA = 0 or CR3[11:0] ≠ 000H.
- MOV to CR0 causes a general-protection exception if it would clear CR0.PG to 0 while CR4.PCIDE = 1.

When a logical processor creates entries in the TLBs (Section 4.10.2) and paging-structure caches (Section 4.10.3), it associates those entries with the current PCID. When using entries in the TLBs and paging-structure caches to translate a linear address, a logical processor uses only those entries associated with the current PCID (see Section 4.10.2.4 for an exception).

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. This is because (1) if CR4.PCIDE = 0, the logical processor will associate any newly cached information with the current PCID, 000H; and (2) if MOV to CR4 clears CR4.PCIDE, all cached information is invalidated (see Section 4.10.4.1).

1. Note that, while HLAT paging (Section 4.5.3) does not use CR3 to locate the first paging structure, it does use the PCID in CR3[11:0] when CR4.PCIDE = 1.
NOTE

In revisions of this manual that were produced when no processors allowed CR4.PCIDE to be set to 1, Section 4.10, "Caching Translation Information," discussed the caching of translation information without any reference to PCIDs. While the section now refers to PCIDs in its specification of this caching, this documentation change is not intended to imply any change to the behavior of processors that do not allow CR4.PCIDE to be set to 1.

4.10.2 Translation Lookaside Buffers (TLBs)

A processor may cache information about the translation of linear addresses in translation lookaside buffers (TLBs). In general, TLBs contain entries that map page numbers to page frames; these terms are defined in Section 4.10.2.1. Section 4.10.2.2 describes how information may be cached in TLBs, and Section 4.10.2.3 gives details of TLB usage. Section 4.10.2.4 explains the global-page feature, which allows software to indicate that certain translations should receive special treatment when cached in the TLBs.

4.10.2.1 Page Numbers, Page Frames, and Page Offsets

Section 4.3, Section 4.4.2, and Section 4.5 give details of how the different paging modes translate linear addresses to physical addresses. Specifically, the upper bits of a linear address (called the page number) determine the upper bits of the physical address (called the page frame); the lower bits of the linear address (called the page offset) determine the lower bits of the physical address. The boundary between the page number and the page offset is determined by the page size. Specifically:

- 32-bit paging:
  - If the translation does not use a PTE (because CR4.PSE = 1 and the PS flag is 1 in the PDE used), the page size is 4 MBytes and the page number comprises bits 31:22 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.

- PAE paging:
  - If the translation does not use a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 31:21 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.

- 4-level paging and 5-level paging:
  - If the translation does not use a PDE (because the PS flag is 1 in the PDPTE used), the page size is 1 GByte and the page number comprises bits 47:30 of the linear address.
  - If the translation does use a PDE but does not use a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 47:21 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 47:12 of the linear address.

  The page size identified by the preceding items may be reduced if there has been a restart of HLAT paging (see Section 4.5.5). Restart of HLAT paging always specifies a maximum page size; this page size is determined by the level of the paging-structure entry that caused the restart. The page size used by the translation is the minimum of the maximum page size specified by the restart and the page size determined by the restarted translation (as specified by the previous items).

  For example, suppose that HLAT paging encounters a PDE that sets bit 11, indicating a restart. As a result, the restart uses a maximum page size of 2 MBytes. Suppose that the restarted translation encounters a PDPTE that sets bit 7, indicating a 1-GByte page. In this case, the translation produced will have a page size of 2 MBytes (the smaller of the two sizes).
4.10.2.2 Caching Translations in TLBs

The processor may accelerate the paging process by caching individual translations in translation lookaside buffers (TLBs). Each entry in a TLB is an individual translation. Each translation is referenced by a page number. It contains the following information from the paging-structure entries used to translate linear addresses with the page number:

- The physical address corresponding to the page number (the page frame).
- The access rights from the paging-structure entries used to translate linear addresses with the page number (see Section 4.6):
  - The logical-AND of the R/W flags.
  - The logical-AND of the U/S flags.
  - The logical-OR of the XD flags (necessary only if IA32_EFER.NXE = 1).
  - The protection key (only with 4-level paging and 5-level paging).
- Attributes from a paging-structure entry that identifies the final page frame for the page number (either a PTE or a paging-structure entry in which the PS flag is 1):
  - The dirty flag (see Section 4.8).
  - The memory type (see Section 4.9).

(TLB entries may contain other information as well. A processor may implement multiple TLBs, and some of these may be for special purposes, e.g., only for instruction fetches. Such special-purpose TLBs may not contain some of this information if it is not necessary. For example, a TLB used only for instruction fetches need not contain information about the R/W and dirty flags.)

As noted in Section 4.10.1, any TLB entries created by a logical processor are associated with the current PCID. Processors need not implement any TLBs. Processors that do implement TLBs may invalidate any TLB entry at any time. Software should not rely on the existence of TLBs or on the retention of TLB entries.

4.10.2.3 Details of TLB Use

Because the TLBs cache entries only for linear addresses with translations, there can be a TLB entry for a page number only if the P flag is 1 and the reserved bits are 0 in each of the paging-structure entries used to translate that page number. In addition, the processor does not cache a translation for a page number unless the accessed flag is 1 in each of the paging-structure entries used during translation; before caching a translation, the processor sets any of these accessed flags that is not already 1.

Subject to the limitations given in the previous paragraph, the processor may cache a translation for any linear address, even if that address is not used to access memory. For example, the processor may cache translations required for prefetches and for accesses that result from speculative execution that would never actually occur in the executed code path.

If the page number of a linear address corresponds to a TLB entry associated with the current PCID, the processor may use that TLB entry to determine the page frame, access rights, and other attributes for accesses to that linear address. In this case, the processor may not actually consult the paging structures in memory. The processor may retain a TLB entry unmodified even if software subsequently modifies the relevant paging-structure entries in memory. See Section 4.10.4.2 for how software can ensure that the processor uses the modified paging-structure entries.

If the paging structures specify a translation using a page larger than 4 KBytes, some processors may cache multiple smaller-page TLB entries for that translation. Each such TLB entry would be associated with a page number corresponding to the smaller page size (e.g., bits 47:12 of a linear address with 4-level paging), even though part of that page number (e.g., bits 20:12) is part of the offset with respect to the page specified by the paging structures. The upper bits of the physical address in such a TLB entry are derived from the physical address in the PDE used to create the translation, while the lower bits come from the linear address of the access for which the translation is created. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. For example, an execution of INVLPG for a linear address on such a page invalidates any and all smaller-page TLB entries for the translation of any linear address on that page.
If software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes, the TLBs may subsequently contain multiple translations for the address range (one for each page size). A reference to a linear address in the address range may use any of these translations. Which translation is used may vary from one execution to another, and the choice may be implementation-specific.

4.10.2.4  Global Pages

The Intel-64 and IA-32 architectures also allow for **global pages** when the PGE flag (bit 7) is 1 in CR4. If the G flag (bit 8) is 1 in a paging-structure entry that maps a page (either a PTE or a paging-structure entry in which the PS flag is 1), any TLB entry cached for a linear address using that paging-structure entry is considered to be **global**. Because the G flag is used only in paging-structure entries that map a page, and because information from such entries is not cached in the paging-structure caches, the global-page feature does not affect the behavior of the paging-structure caches.

A logical processor may use a global TLB entry to translate a linear address, even if the TLB entry is associated with a PCID different from the current PCID.

4.10.3  Paging-Structure Caches

In addition to the TLBs, a processor may cache other information about the paging structures in memory.

4.10.3.1  Caches for Paging Structures

A processor may support any or all of the following paging-structure caches:

- **PML5E cache** (5-level paging only). Each PML5E-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 56:48 have that value. The entry contains information from the PML5E used to translate such linear addresses:
  - The physical address from the PML5E (the address of the PML4 table).
  - The value of the R/W flag of the PML5E.
  - The value of the U/S flag of the PML5E.
  - The value of the XD flag of the PML5E.
  - The values of the PCD and PWT flags of the PML5E.

The following items detail how a processor may use the PML5E cache:

- If the processor has a PML5E-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E in memory).
- The processor does not create a PML5E-cache entry unless the P flag is 1 and all reserved bits are 0 in the PML5E in memory.
- The processor does not create a PML5E-cache entry unless the accessed flag is 1 in the PML5E in memory; before caching a translation, the processor sets the accessed flag if it is not already 1.
- The processor may create a PML5E-cache entry even if there are no translations for any linear address that might use that entry (e.g., because the P flags are 0 in all entries in the referenced PML4 table).
- If the processor creates a PML5E-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E in memory.

- **PML4E cache** (4-level paging and 5-level paging only). The use of the PML4E cache depends on the paging mode:
  - For 4-level paging, each PML4E-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 47:39 have that value.
  - For 5-level paging, each PML4E-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 56:39 have that value.

A PML4E-cache entry contains information from the PML5E and PML4E used to translate the relevant linear addresses (for 4-level paging, the PML5E does not apply):
— The physical address from the PML4E (the address of the page-directory-pointer table).
— The logical-AND of the R/W flags in the PML5E and the PML4E.
— The logical-AND of the U/S flags in the PML5E and the PML4E.
— The logical-OR of the XD flags in the PML5E and the PML4E.
— The values of the PCD and PWT flags of the PML4E.

The following items detail how a processor may use the PML4E cache:
— If the processor has a PML4E-cache entry for a linear address, it may use that entry when translating the
  linear address (instead of the PML5E and PML4E in memory).
— The processor does not create a PML4E-cache entry unless the P flags are 1 and all reserved bits are 0 in
  the PML5E and the PML4E in memory.
— The processor does not create a PML4E-cache entry unless the accessed flags are 1 in the PML5E and the
  PML4E in memory; before caching a translation, the processor sets any accessed flags that are not already
  1.
— The processor may create a PML4E-cache entry even if there are no translations for any linear address that
  might use that entry (e.g., because the P flags are 0 in all entries in the referenced page-directory-pointer
  table).
— If the processor creates a PML4E-cache entry, the processor may retain it unmodified even if software
  subsequently modifies the corresponding PML4E in memory.

• PDPTE cache (4-level paging and 5-level paging only).1 The use of the PML4E cache depends on the paging
  mode:
— For 4-level paging, each PDPTE-cache entry is referenced by an 18-bit value and is used for linear
  addresses for which bits 47:30 have that value.
— For 5-level paging, each PDPTE-cache entry is referenced by a 27-bit value and is used for linear addresses
  for which bits 56:30 have that value.

A PDPTE-cache entry contains information from the PML5E, PML4E, PDPTE used to translate the relevant linear
addresses (for 4-level paging, the PML5E does not apply):
— The physical address from the PDPTE (the address of the page directory). (No PDPTE-cache entry is created
  for a PDPTE that maps a 1-GBYTE page.)
— The logical-AND of the R/W flags in the PML5E, PML4E, and PDPTE.
— The logical-AND of the U/S flags in the PML5E, PML4E, and PDPTE.
— The logical-OR of the XD flags in the PML5E, PML4E, and PDPTE.
— The values of the PCD and PWT flags of the PDPTE.

The following items detail how a processor may use the PDPTE cache:
— If the processor has a PDPTE-cache entry for a linear address, it may use that entry when translating the
  linear address (instead of the PML5E, PML4E, and PDPTE in memory).
— The processor does not create a PDPTE-cache entry unless the P flags are 1, the PS flags are 0, and the
  reserved bits are 0 in the PML5E, PML4E, and PDPTE in memory.
— The processor does not create a PDPTE-cache entry unless the accessed flags are 1 in the PML5E, PML4E,
  and PDPTE in memory; before caching a translation, the processor sets any accessed flags that are not
  already 1.
— The processor may create a PDPTE-cache entry even if there are no translations for any linear address that
  might use that entry.
— If the processor creates a PDPTE-cache entry, the processor may retain it unmodified even if software
  subsequently modifies the corresponding PML5E, PML4E, or PDPTE in memory.

1. With PAE paging, the PDPTEs are stored in internal, non-architectural registers. The operation of these registers is described in Sec-
   tion 4.4.1 and differs from that described here.
**PDE cache.** The use of the PDE cache depends on the paging mode:

- For 32-bit paging, each PDE-cache entry is referenced by a 10-bit value and is used for linear addresses for which bits 31:22 have that value.
- For PAE paging, each PDE-cache entry is referenced by an 11-bit value and is used for linear addresses for which bits 31:21 have that value.
- For 4-level paging, each PDE-cache entry is referenced by a 27-bit value and is used for linear addresses for which bits 47:21 have that value.
- For 5-level paging, each PDE-cache entry is referenced by a 36-bit value and is used for linear addresses for which bits 56:21 have that value.

A PDE-cache entry contains information from the PML5E, PML4E, PDPT, and PDE used to translate the relevant linear addresses (for 32-bit paging and PAE paging, only the PDE applies; for 4-level paging, the PML5E does not apply):

- The physical address from the PDE (the address of the page table). (No PDE-cache entry is created for a PDE that maps a page.)
- The logical-AND of the R/W flags in the PML5E, PML4E, PDPT, and PDE.
- The logical-AND of the U/S flags in the PML5E, PML4E, PDPT, and PDE.
- The logical-OR of the XD flags in the PML5E, PML4E, PDPT, and PDE.
- The values of the PCD and PWT flags of the PDE.

The following items detail how a processor may use the PDE cache (references below to PML5Es, PML4Es, and PDPTs apply only to 4-level paging and to 5-level paging, as appropriate):

- If the processor has a PDE-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E, PML4E, PDPT, and PDE in memory).
- The processor does not create a PDE-cache entry unless the P flags are 1, the PS flags are 0, and the reserved bits are 0 in the PML5E, PML4E, PDPT, and PDE in memory.
- The processor does not create a PDE-cache entry unless the accessed flag is 1 in the PML5E, PML4E, PDPT, and PDE in memory; before caching a translation, the processor sets any accessed flags that are not already 1.
- The processor may create a PDE-cache entry even if there are no translations for any linear address that might use that entry.
- If the processor creates a PDE-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E, PML4E, PDPT, or PDE in memory.

Information from a paging-structure entry can be included in entries in the paging-structure caches for other paging-structure entries referenced by the original entry. For example, if the R/W flag is 0 in a PML4E, then the R/W flag will be 0 in any PDPT-cache entry for a PDPT from the page-directory-pointer table referenced by that PML4E. This is because the R/W flag of each such PDPT-cache entry is the logical-AND of the R/W flags in the appropriate PML4E and PDPT.

On processors that support HLAT paging (see Section 4.5.1), each entry in a paging-structure cache indicates whether the entry was cached during ordinary paging or HLAT paging. When the processor commences linear-address translation using ordinary paging (respectively, HLAT paging), it will use only entries that indicate that they were cached during ordinary paging (respectively, HLAT paging).

Entries that were cached during HLAT paging also include the restart flag (bit 11) of the original paging-structure entry. When the processor commences HLAT paging using such an entry, it immediately restarts (using ordinary paging) if this cached restart flag is 1.

The paging-structure caches contain information only from paging-structure entries that reference other paging structures (and not those that map pages). Because the G flag is not used in such paging-structure entries, the global-page feature does not affect the behavior of the paging-structure caches.

The processor may create entries in paging-structure caches for translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path.
As noted in Section 4.10.1, any entries created in paging-structure caches by a logical processor are associated with the current PCID.

A processor may or may not implement any of the paging-structure caches. Software should rely on neither their presence nor their absence. The processor may invalidate entries in these caches at any time. Because the processor may create the cache entries at the time of translation and not update them following subsequent modifications to the paging structures in memory, software should take care to invalidate the cache entries appropriately when causing such modifications. The invalidation of TLBs and the paging-structure caches is described in Section 4.10.4.

4.10.3.2 Using the Paging-Structure Caches to Translate Linear Addresses

When a linear address is accessed, the processor uses a procedure such as the following to determine the physical address to which it translates and whether the access should be allowed:

- If the processor finds a TLB entry that is for the page number of the linear address and that is associated with the current PCID (or which is global), it may use the physical address, access rights, and other attributes from that entry.
- If the processor does not find a relevant TLB entry, it may use the upper bits of the linear address to select an entry from the PDE cache that is associated with the current PCID (Section 4.10.3.1 indicates which bits are used in each paging mode). It can then use that entry to complete the translation process (locating a PTE, etc.) as if it had traversed the PDE (and, for 4-level paging and 5-level paging, the PDPTE, PML4E, and PML5E, as appropriate) corresponding to the PDE-cache entry.
- The following items apply when 4-level paging or 5-level paging is used:
  - If the processor does not find a relevant TLB entry, PDE-cache entry, or PDPTE-cache entry, it may use the upper bits of the linear address (for 4-level paging, bits 47:30; for 5-level paging, bits 56:30) to select an entry from the PDPTE cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PDE, etc.) as if it had traversed the PDPTE, the PML4E, and (for 5-level paging) the PML5E corresponding to the PDPTE-cache entry.
  - If the processor does not find a relevant TLB entry, PDE-cache entry, or PDPTE-cache entry, it may use the upper bits of the linear address (for 4-level paging, bits 47:39; for 5-level paging, bits 56:39) to select an entry from the PML4E cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PDPTE, etc.) as if it had traversed the corresponding PML4E.
  - With 5-level paging, if the processor does not find a relevant TLB entry, PDE-cache entry, PDPTE-cache entry, or PML4E-cache entry, it may use bits 56:48 of the linear address to select an entry from the PML5E cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PML4E, etc.) as if it had traversed the corresponding PML5E.

(Any of the above steps would be skipped if the processor does not support the cache in question.)

If the processor does not find a TLB or paging-structure-cache entry for the linear address, it uses the linear address to traverse the entire paging-structure hierarchy, as described in Section 4.3, Section 4.4.2, and Section 4.5.

4.10.3.3 Multiple Cached Entries for a Single Paging-Structure Entry

The paging-structure caches and TLBs may contain multiple entries associated with a single PCID and with information derived from a single paging-structure entry. The following items give some examples for 4-level paging:

- Suppose that two PML4Es contain the same physical address and thus reference the same page-directory-pointer table. Any PDPTE in that table may result in two PDPTE-cache entries, each associated with a different set of linear addresses. Specifically, suppose that the n1th and n2th entries in the PML4 table contain the same physical address. This implies that the physical address in the mth PDPTE in the page-directory-pointer table would appear in the PDPTE-cache entries associated with both p1 and p2, where (p1 \(\gg\) 9) = n1, (p2 \(\gg\) 9) = n2, and (p1 & 1FFH) = (p2 & 1FFH) = m. This is because both PDPTE-cache entries use the same PDPTE, one resulting from a reference from the n1th PML4E and one from the n2th PML4E.
- Suppose that the first PML4E (i.e., the one in position 0) contains the physical address X in CR3 (the physical address of the PML4 table). This implies the following:
— Any PML4-cache entry associated with linear addresses with 0 in bits 47:39 contains address X.
— Any PDPTE-cache entry associated with linear addresses with 0 in bits 47:30 contains address X. This is because the translation for a linear address for which the value of bits 47:30 is 0 uses the value of bits 47:39 (0) to locate a page-directory-pointer table at address X (the address of the PML4 table). It then uses the value of bits 38:30 (also 0) to find address X again and to store that address in the PDPTE-cache entry.
— Any PDE-cache entry associated with linear addresses with 0 in bits 47:21 contains address X for similar reasons.
— Any TLB entry for page number 0 (associated with linear addresses with 0 in bits 47:12) translates to page frame X + 12 for similar reasons.

The same PML4E contributes its address X to all these cache entries because the self-referencing nature of the entry causes it to be used as a PML4E, a PDPTE, a PDE, and a PTE.

4.10.4 Invalidation of TLBs and Paging-Structure Caches

As noted in Section 4.10.2 and Section 4.10.3, the processor may create entries in the TLBs and the paging-structure caches when linear addresses are translated, and it may retain these entries even after the paging structures used to create them have been modified. To ensure that linear-address translation uses the modified paging structures, software should take action to invalidate any cached entries that may contain information that has since been modified.

4.10.4.1 Operations that Invalidate TLBs and Paging-Structure Caches

The following instructions invalidate entries in the TLBs and the paging-structure caches:

• INVLPg. This instruction takes a single operand, which is a linear address. The instruction invalidates any TLB entries that are for a page number corresponding to the linear address and that are associated with the current PCID. It also invalidates any global TLB entries with that page number, regardless of PCID (see Section 4.10.2.4). INVLPg also invalidates all entries in all paging-structure caches associated with the current PCID, regardless of the linear addresses to which they correspond.

• INVPCID. The operation of this instruction is based on instruction operands, called the INVPCID type and the INVPCID descriptor. Four INVPCID types are currently defined:
  — Individual-address. If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—associated with the PCID specified in the INVPCID descriptor and that would be used to translate the linear address specified in the INVPCID descriptor. (The instruction may also invalidate global translations, as well as mappings associated with other PCIDs and for other linear addresses.)
  — Single-context. If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. (The instruction may also invalidate global translations, as well as mappings associated with other PCIDs.)
  — All-context, including globals. If the INVPCID type is 2, the logical processor invalidates mappings—including global translations—associated with all PCIDs.
  — All-context. If the INVPCID type is 3, the logical processor invalidates mappings—except global translations—associated with all PCIDs. (The instruction may also invalidate global translations.)


• MOV to CR0. The instruction invalidates all TLB entries (including global entries) and all entries in all paging-structure caches (for all PCIDs) if it changes the value of CR0.PG from 1 to 0.

• MOV to CR3. The behavior of the instruction depends on the value of CR4.PCIDE:

  1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3), the instruction invalidates all of them.
  2. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3), the instruction invalidates all of them.
— If CR4.PCIDE = 0, the instruction invalidates all TLB entries associated with PCID 000H except those for global pages. It also invalidates all entries in all paging-structure caches associated with PCID 000H.

— If CR4.PCIDE = 1 and bit 63 of the instruction’s source operand is 0, the instruction invalidates all TLB entries associated with the PCID specified in bits 11:0 of the instruction’s source operand except those for global pages. It also invalidates all entries in all paging-structure caches associated with that PCID. It is not required to invalidate entries in the TLBs and paging-structure caches that are associated with other PCIDs.

— If CR4.PCIDE = 1 and bit 63 of the instruction’s source operand is 1, the instruction is not required to invalidate any TLB entries or entries in paging-structure caches.

• MOV to CR4. The behavior of the instruction depends on the bits being modified:
  — The instruction invalidates all TLB entries (including global entries) and all entries in all paging-structure caches (for all PCIDs) if (1) it changes the value of CR4.PGE;\(^1\) or (2) it changes the value of the CR4.PCIDE from 1 to 0.
  — The instruction invalidates all TLB entries and all entries in all paging-structure caches for the current PCID if (1) it changes the value of CR4.PAE; or (2) it changes the value of CR4.SMEP from 0 to 1.

• Task switch. If a task switch changes the value of CR3, it invalidates all TLB entries associated with PCID 000H except those for global pages. It also invalidates all entries in all paging-structure caches associated with PCID 000H.\(^2\)

• VMX transitions. See Section 4.11.1.

The processor is always free to invalidate additional entries in the TLBs and paging-structure caches. The following are some examples:

• INVLPAG may invalidate TLB entries for pages other than the one corresponding to its linear-address operand. It may invalidate TLB entries and paging-structure-cache entries associated with PCIDs other than the current PCID.

• INVPCID may invalidate TLB entries for pages other than the one corresponding to the specified linear address. It may invalidate TLB entries and paging-structure-cache entries associated with PCIDs other than the specified PCID.

• MOV to CR0 may invalidate TLB entries even if CR0.PG is not changing. For example, this may occur if either CR0.CD or CR0.NW is modified.

• MOV to CR3 may invalidate TLB entries for global pages. If CR4.PCIDE = 1 and bit 63 of the instruction’s source operand is 0, it may invalidate TLB entries and entries in the paging-structure caches associated with PCIDs other than the PCID it is establishing. It may invalidate entries if CR4.PCIDE = 1 and bit 63 of the instruction’s source operand is 1.

• MOV to CR4 may invalidate TLB entries when changing CR4.PSE or when changing CR4.SMEP from 1 to 0.

• On a processor supporting Hyper-Threading Technology, invalidations performed on one logical processor may invalidate entries in the TLBs and paging-structure caches used by other logical processors.

(Other instructions and operations may invalidate entries in the TLBs and the paging-structure caches, but the instructions identified above are recommended.)

In addition to the instructions identified above, page faults invalidate entries in the TLBs and paging-structure caches. In particular, a page-fault exception resulting from an attempt to use a linear address will invalidate any TLB entries that are for a page number corresponding to that linear address and that are associated with the current PCID. It also invalidates all entries in the paging-structure caches that would be used for that linear address and that are associated with the current PCID.\(^3\) These invalidations ensure that the page-fault exception will not recur (if the faulting instruction is re-executed) if it would not be caused by the contents of the paging structures.

---

1. If CR4.PGE is changing from 0 to 1, there were no global TLB entries before the execution; if CR4.PGE is changing from 1 to 0, there will be no global TLB entries after the execution.

2. Task switches do not occur in IA-32e mode and thus cannot occur with 4-level paging. Since CR4.PCIDE can be set only with 4-level paging, task switches occur only with CR4.PCIDE = 0.

3. Unlike INVLPAG, page faults need not invalidate all entries in the paging-structure caches, only those that would be used to translate the faulting linear address.
in memory (and if, therefore, it resulted from cached entries that were not invalidated after the paging structures were modified in memory).

As noted in Section 4.10.2, some processors may choose to cache multiple smaller-page TLB entries for a translation specified by the paging structures to use a page larger than 4 KBytes. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. The INVLP0 instruction and page faults provide the same assurances that they provide when a single TLB entry is used: they invalidate all TLB entries corresponding to the translation specified by the paging structures.

**4.10.4.2 Recommended Invalidation**

The following items provide some recommendations regarding when software should perform invalidations:

- If software modifies a paging-structure entry that maps a page (rather than referencing another paging structure), it should execute INVLP0 for any linear address with a page number whose translation uses that paging-structure entry.\(^1\)

  (If the paging-structure entry may be used in the translation of different page numbers — see Section 4.10.3.3 — software should execute INVLP0 for linear addresses with each of those page numbers; alternatively, it could use MOV to CR3 or MOV to CR4.)

- If software modifies a paging-structure entry that references another paging structure, it may use one of the following approaches depending upon the types and number of translations controlled by the modified entry:

  - Execute INVLP0 for linear addresses with each of the page numbers with translations that would use the entry. However, if no page numbers that would use the entry have translations (e.g., because the P flags are 0 in all entries in the paging structure referenced by the modified entry), it remains necessary to execute INVLP0 at least once.

  - Execute MOV to CR3 if the modified entry controls no global pages.

  - Execute MOV to CR4 to modify CR4.PGE.

- If CR4.PCIDE = 1 and software modifies a paging-structure entry that does not map a page or in which the G flag (bit 8) is 0, additional steps are required if the entry may be used for PCIDs other than the current one. Any one of the following suffices:

  - Execute MOV to CR4 to modify CR4.PGE, either immediately or before again using any of the affected PCIDs. For example, software could use different (previously unused) PCIDs for the processes that used the affected PCIDs.

  - For each affected PCID, execute MOV to CR3 to make that PCID current (and to load the address of the appropriate PML4 table). If the modified entry controls no global pages and bit 63 of the source operand to MOV to CR3 was 0, no further steps are required. Otherwise, execute INVLP0 for linear addresses with each of the page numbers with translations that would use the entry; if no page numbers that would use the entry have translations, execute INVLP0 at least once.

- If software using PAE paging modifies a PDPTE, it should reload CR3 with the register’s current value to ensure that the modified PDPTE is loaded into the corresponding PDPTE register (see Section 4.4.1).

- If the nature of the paging structures is such that a single entry may be used for multiple purposes (see Section 4.10.3.3), software should perform invalidations for all of these purposes. For example, if a single entry might serve as both a PDE and PTE, it may be necessary to execute INVLP0 with two (or more) linear addresses, one that uses the entry as a PDE and one that uses it as a PTE. (Alternatively, software could use MOV to CR3 or MOV to CR4.)

- As noted in Section 4.10.2, the TLBs may subsequently contain multiple translations for the address range if software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes. A reference to a linear address in the address range may use any of these translations.

  Software wishing to prevent this uncertainty should not write to a paging-structure entry in a way that would change, for any linear address, both the page size and either the page frame, access rights, or other attributes. It can instead use the following algorithm: first clear the P flag in the relevant paging-structure entry (e.g.,

---

1. One execution of INVLP0 is sufficient even for a page with size greater than 4 KBytes.
PDE); then invalidate any translations for the affected linear addresses (see above); and then modify the relevant paging-structure entry to set the P flag and establish modified translation(s) for the new page size.

- Software should clear bit 63 of the source operand to a MOV to CR3 instruction that establishes a PCID that had been used earlier for a different linear-address space (e.g., with a different value in bits 51:12 of CR3). This ensures invalidation of any information that may have been cached for the previous linear-address space. This assumes that both linear-address spaces use the same global pages and that it is thus not necessary to invalidate any global TLB entries. If that is not the case, software should invalidate those entries by executing MOV to CR4 to modify CR4.PGE.

### 4.10.4.3 Optional Invalidation

The following items describe cases in which software may choose not to invalidate and the potential consequences of that choice:

- If a paging-structure entry is modified to change the P flag from 0 to 1, no invalidation is necessary. This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the P flag is 0.1

- If a paging-structure entry is modified to change the accessed flag from 0 to 1, no invalidation is necessary (assuming that an invalidation was performed the last time the accessed flag was changed from 1 to 0). This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the accessed flag is 0.

- If a paging-structure entry is modified to change the R/W flag from 0 to 1, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted write access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.4.1).

- If CR4.SMEP = 0 and a paging-structure entry is modified to change the U/S flag from 0 to 1, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted user-mode access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.4.1).

- If a paging-structure entry is modified to change the XD flag from 1 to 0, failure to perform an invalidation may result in a “spurious” page-fault exception (e.g., in response to an attempted instruction fetch) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 4.10.4.1).

- If a paging-structure entry is modified to change the accessed flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent access to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such an access has not occurred.

- If software modifies a paging-structure entry that identifies the final physical address for a linear address (either a PTE or a paging-structure entry in which the PS flag is 1) to change the dirty flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent write to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such a write has not occurred.

- The read of a paging-structure entry in translating an address being used to fetch an instruction may appear to execute before an earlier write to that paging-structure entry if there is no serializing instruction between the write and the instruction fetch. Note that the invalidating instructions identified in Section 4.10.4.1 are all serializing instructions.

- Section 4.10.3.3 describes situations in which a single paging-structure entry may contain information cached in multiple entries in the paging-structure caches. Because all entries in these caches are invalidated by any execution of INVLPG, it is not necessary to follow the modification of such a paging-structure entry by executing INVLPG multiple times solely for the purpose of invalidating these multiple cached entries. (It may be necessary to do so to invalidate multiple TLB entries.)

---

1. If it is also the case that no invalidation was performed the last time the P flag was changed from 1 to 0, the processor may use a TLB entry or paging-structure cache entry that was created when the P flag had earlier been 1.
4.10.4.4 Delayed Invalidation

Required invalidations may be delayed under some circumstances. Software developers should understand that, between the modification of a paging-structure entry and execution of the invalidation instruction recommended in Section 4.10.4.2, the processor may use translations based on either the old value or the new value of the paging-structure entry. The following items describe some of the potential consequences of delayed invalidation:

• If a paging-structure entry is modified to change the P flag from 1 to 0, an access to a linear address whose translation is controlled by this entry may or may not cause a page-fault exception.
• If a paging-structure entry is modified to change the R/W flag from 0 to 1, write accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
• If a paging-structure entry is modified to change the U/S flag from 0 to 1, user-mode accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
• If a paging-structure entry is modified to change the XD flag from 1 to 0, instruction fetches from linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.

As noted in Section 9.1.1, an x87 instruction or an SSE instruction that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory and invalidation has been delayed, some of the accesses may complete (writing to memory) while another causes a page-fault exception. In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault.

In some cases, the consequences of delayed invalidation may not affect software adversely. For example, when freeing a portion of the linear-address space (by marking paging-structure entries “not present”), invalidation using INVLPG may be delayed if software does not re-allocate that portion of the linear-address space or the memory that had been associated with it. However, because of speculative execution (or errant software), there may be accesses to the freed portion of the linear-address space before the invalidations occur. In this case, the following can happen:

• Reads can occur to the freed portion of the linear-address space. Therefore, invalidation should not be delayed for an address range that has read side effects.
• The processor may retain entries in the TLBs and paging-structure caches for an extended period of time. Software should not assume that the processor will not use entries associated with a linear address simply because time has passed.
• As noted in Section 4.10.3.1, the processor may create an entry in a paging-structure cache even if there are no translations for any linear address that might use that entry. Thus, if software has marked “not present” all entries in a page table, the processor may subsequently create a PDE-cache entry for the PDE that references that page table (assuming that the PDE itself is marked “present”).
• If software attempts to write to the freed portion of the linear-address space, the processor might not generate a page fault. (Such an attempt would likely be the result of a software error.) For that reason, the page frames previously associated with the freed portion of the linear-address space should not be reallocated for another purpose until the appropriate invalidations have been performed.

4.10.5 Propagation of Paging-Structure Changes to Multiple Processors

As noted in Section 4.10.4, software that modifies a paging-structure entry may need to invalidate entries in the TLBs and paging-structure caches that were derived from the modified entry before it was modified. In a system containing more than one logical processor, software must account for the fact that there may be entries in the TLBs and paging-structure caches of logical processors other than the one used to modify the paging-structure entry. The process of propagating the changes to a paging-structure entry is commonly referred to as “TLB shootdown.”

TLB shootdown can be done using memory-based semaphores and/or interprocessor interrupts (IPI). The following items describe a simple but inefficient example of a TLB shootdown algorithm for processors supporting the Intel-64 and IA-32 architectures:

1. If the accesses are to different pages, this may occur even if invalidation has not been delayed.
1. Begin barrier: Stop all but one logical processor; that is, cause all but one to execute the HLT instruction or to enter a spin loop.

2. Allow the active logical processor to change the necessary paging-structure entries.

3. Allow all logical processors to perform invalidations appropriate to the modifications to the paging-structure entries.

4. Allow all logical processors to resume normal operation.

Alternative, performance-optimized, TLB shootdown algorithms may be developed; however, software developers must take care to ensure that the following conditions are met:

- All logical processors that are using the paging structures that are being modified must participate and perform appropriate invalidations after the modifications are made.

- If the modifications to the paging-structure entries are made before the barrier or if there is no barrier, the operating system must ensure one of the following: (1) that the affected linear-address range is not used between the time of modification and the time of invalidation; or (2) that it is prepared to deal with the consequences of the affected linear-address range being used during that period. For example, if the operating system does not allow pages being freed to be reallocated for another purpose until after the required invalidations, writes to those pages by errant software will not unexpectedly modify memory that is in use.

- Software must be prepared to deal with reads, instruction fetches, and prefetch requests to the affected linear-address range that are a result of speculative execution that would never actually occur in the executed code path.

When multiple logical processors are using the same linear-address space at the same time, they must coordinate before any request to modify the paging-structure entries that control that linear-address space. In these cases, the barrier in the TLB shootdown routine may not be required. For example, when freeing a range of linear addresses, some other mechanism can assure no logical processor is using that range before the request to free it is made. In this case, a logical processor freeing the range can clear the P flags in the PTEs associated with the range, free the physical page frames associated with the range, and then signal the other logical processors using that linear-address space to perform the necessary invalidations. All the affected logical processors must complete their invalidations before the linear-address range and the physical page frames previously associated with that range can be reallocated.

4.11 INTERACTIONS WITH VIRTUAL-MACHINE EXTENSIONS (VMX)

The architecture for virtual-machine extensions (VMX) includes features that interact with paging. Section 4.11.1 discusses ways in which VMX-specific control transfers, called VMX transitions specially affect paging. Section 4.11.2 gives an overview of VMX features specifically designed to support address translation.

4.11.1 VMX Transitions

The VMX architecture defines two control transfers called VM entries and VM exits; collectively, these are called VMX transitions. VM entries and VM exits are described in detail in Chapter 27 and Chapter 28, respectively, in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C. The following items identify paging-related details:

- VMX transitions modify the CR0 and CR4 registers and the IA32_EFER MSR concurrently. For this reason, they allow transitions between paging modes that would not otherwise be possible:
  - VM entries allow transitions from 4-level paging directly to either 32-bit paging or PAE paging.
  - VM exits allow transitions from either 32-bit paging or PAE paging directly to 4-level paging or 5-level paging.

- VMX transitions that result in PAE paging load the PDPT registers (see Section 4.4.1) as follows:
  - VM entries load the PDPT registers either from the physical address being loaded into CR3 or from the virtual-machine control structure (VMCS); see Section 27.3.2.4.
  - VM exits load the PDPT registers from the physical address being loaded into CR3; see Section 28.5.4.
VMX transitions invalidate the TLBs and paging-structure caches based on certain control settings. See Section 27.3.2.5 and Section 28.5.5 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

4.11.2 VMX Support for Address Translation

Chapter 29, "VMX Support for Address Translation," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, describes two features of the virtual-machine extensions (VMX) that interact directly with paging. These are virtual-processor identifiers (VPIDs) and the extended page table mechanism (EPT). VPIDs provide a way for software to identify to the processor the address spaces for different “virtual processors.” The processor may use this identification to maintain concurrently information for multiple address spaces in its TLBs and paging-structure caches, even when non-zero PCIDs are not being used. See Section 29.1 for details.

When EPT is in use, the addresses in the paging-structures are not used as physical addresses to access memory and memory-mapped I/O. Instead, they are treated as guest-physical addresses and are translated through a set of EPT paging structures to produce physical addresses. EPT can also specify its own access rights and memory typing; these are used on conjunction with those specified in this chapter. See Section 29.3 for more information.

Both VPIDs and EPT may change the way that a processor maintains information in TLBs and paging structure caches and the ways in which software can manage that information. Some of the behaviors documented in Section 4.10 may change. See Section 29.4 for details.

4.12 USING PAGING FOR VIRTUAL MEMORY

With paging, portions of the linear-address space need not be mapped to the physical-address space; data for the unmapped addresses can be stored externally (e.g., on disk). This method of mapping the linear-address space is referred to as virtual memory or demand-paged virtual memory.

Paging divides the linear address space into fixed-size pages that can be mapped into the physical-address space and/or external storage. When a program (or task) references a linear address, the processor uses paging to translate the linear address into a corresponding physical address if such an address is defined.

If the page containing the linear address is not currently mapped into the physical-address space, the processor generates a page-fault exception as described in Section 4.7. The handler for page-fault exceptions typically directs the operating system or executive to load data for the unmapped page from external storage into physical memory (perhaps writing a different page from physical memory out to external storage in the process) and to map it using paging (by updating the paging structures). When the page has been loaded into physical memory, a return from the exception handler causes the instruction that generated the exception to be restarted.

Paging differs from segmentation through its use of fixed-size pages. Unlike segments, which usually are the same size as the code or data structures they hold, pages have a fixed size. If segmentation is the only form of address translation used, a data structure present in physical memory will have all of its parts in memory. If paging is used, a data structure can be partly in memory and partly in disk storage.

4.13 MAPPING SEGMENTS TO PAGES

The segmentation and paging mechanisms provide support for a wide variety of approaches to memory management. When segmentation and paging are combined, segments can be mapped to pages in several ways. To implement a flat (unsegmented) addressing environment, for example, all the code, data, and stack modules can be mapped to one or more large segments (up to 4-GBytes) that share same range of linear addresses (see Figure 3-2 in Section 3.2.2). Here, segments are essentially invisible to applications and the operating-system or executive. If paging is used, the paging mechanism can map a single linear-address space (contained in a single segment) into virtual memory. Alternatively, each program (or task) can have its own large linear-address space (contained in its own segment), which is mapped into virtual memory through its own paging structures.

Segments can be smaller than the size of a page. If one of these segments is placed in a page which is not shared with another segment, the extra memory is wasted. For example, a small data structure, such as a 1-Byte semaphore, occupies 4 KBytes if it is placed in a page by itself. If many semaphores are used, it is more efficient to pack them into a single page.
The Intel-64 and IA-32 architectures do not enforce correspondence between the boundaries of pages and segments. A page can contain the end of one segment and the beginning of another. Similarly, a segment can contain the end of one page and the beginning of another.

Memory-management software may be simpler and more efficient if it enforces some alignment between page and segment boundaries. For example, if a segment which can fit in one page is placed in two pages, there may be twice as much paging overhead to support access to that segment.

One approach to combining paging and segmentation that simplifies memory-management software is to give each segment its own page table, as shown in Figure 4-13. This convention gives the segment a single entry in the page directory, and this entry provides the access control information for paging the entire segment.

![Figure 4-13. Memory Management Convention That Assigns a Page Table to Each Segment](image)
In protected mode, the Intel 64 and IA-32 architectures provide a protection mechanism that operates at both the segment level and the page level. This protection mechanism provides the ability to limit access to certain segments or pages based on privilege levels (four privilege levels for segments and two privilege levels for pages). For example, critical operating-system code and data can be protected by placing them in more privileged segments than those that contain applications code. The processor’s protection mechanism will then prevent application code from accessing the operating-system code and data in any but a controlled, defined manner.

Segment and page protection can be used at all stages of software development to assist in localizing and detecting design problems and bugs. It can also be incorporated into end-products to offer added robustness to operating systems, utilities software, and applications software.

When the protection mechanism is used, each memory reference is checked to verify that it satisfies various protection checks. All checks are made before the memory cycle is started; any violation results in an exception. Because checks are performed in parallel with address translation, there is no performance penalty. The protection checks that are performed fall into the following categories:

- Limit checks.
- Type checks.
- Privilege level checks.
- Restriction of addressable domain.
- Restriction of procedure entry-points.
- Restriction of instruction set.

All protection violation results in an exception being generated. See Chapter 6, “Interrupt and Exception Handling,” for an explanation of the exception mechanism. This chapter describes the protection mechanism and the violations which lead to exceptions.

The following sections describe the protection mechanism available in protected mode. See Chapter 21, “8086 Emulation,” for information on protection in real-address and virtual-8086 mode.

### 5.1 Enabling and Disabling Segment and Page Protection

Setting the PE flag in register CR0 causes the processor to switch to protected mode, which in turn enables the segment-protection mechanism. Once in protected mode, there is no control bit for turning the protection mechanism on or off. The part of the segment-protection mechanism that is based on privilege levels can essentially be disabled while still in protected mode by assigning a privilege level of 0 (most privileged) to all segment selectors and segment descriptors. This action disables the privilege level protection barriers between segments, but other protection checks such as limit checking and type checking are still carried out.

Page-level protection is automatically enabled when paging is enabled (by setting the PG flag in register CR0). Here again there is no mode bit for turning off page-level protection once paging is enabled. However, page-level protection can be disabled by performing the following operations:

- Clear the WP flag in control register CR0.
- Set the read/write (R/W) and user/supervisor (U/S) flags for each page-directory and page-table entry.

This action makes each page a writable, user page, which in effect disables page-level protection.
### 5.2 Fields and Flags Used for Segment-Level and Page-Level Protection

The processor's protection mechanism uses the following fields and flags in the system data structures to control access to segments and pages:

- **Descriptor type (S) flag** — (Bit 12 in the second doubleword of a segment descriptor.) Determines if the segment descriptor is for a system segment or a code or data segment.

- **Type field** — (Bits 8 through 11 in the second doubleword of a segment descriptor.) Determines the type of code, data, or system segment.

- **Limit field** — (Bits 0 through 15 of the first doubleword and bits 16 through 19 of the second doubleword of a segment descriptor.) Determines the size of the segment, along with the G flag and E flag (for data segments).

- **G flag** — (Bit 23 in the second doubleword of a segment descriptor.) Determines the size of the segment, along with the limit field and E flag (for data segments).

- **E flag** — (Bit 10 in the second doubleword of a data-segment descriptor.) Determines the size of the segment, along with the limit field and G flag.

- **Descriptor privilege level (DPL) field** — (Bits 13 and 14 in the second doubleword of a segment descriptor.) Determines the privilege level of the segment.

- **Requested privilege level (RPL) field** — (Bits 0 and 1 of any segment selector.) Specifies the requested privilege level of a segment selector.

- **Current privilege level (CPL) field** — (Bits 0 and 1 of the CS segment register.) Indicates the privilege level of the currently executing program or procedure. The term current privilege level (CPL) refers to the setting of this field.

- **User/supervisor (U/S) flag** — (Bit 2 of paging-structure entries.) Determines the type of page: user or supervisor.

- **Read/write (R/W) flag** — (Bit 1 of paging-structure entries.) Determines the type of access allowed to a page: read-only or read/write.

- **Execute-disable (XD) flag** — (Bit 63 of certain paging-structure entries.) Determines the type of access allowed to a page: executable or not-executable.

Figure 5-1 shows the location of the various fields and flags in the data-, code-, and system-segment descriptors; Figure 3-6 shows the location of the RPL (or CPL) field in a segment selector (or the CS register); and Chapter 4 identifies the locations of the U/S, R/W, and XD flags in the paging-structure entries.
Many different styles of protection schemes can be implemented with these fields and flags. When the operating system creates a descriptor, it places values in these fields and flags in keeping with the particular protection style chosen for an operating system or executive. Application programs do not generally access or modify these fields and flags.

The following sections describe how the processor uses these fields and flags to perform the various categories of checks described in the introduction to this chapter.

### 5.2.1 Code-Segment Descriptor in 64-bit Mode

Code segments continue to exist in 64-bit mode even though, for address calculations, the segment base is treated as zero. Some code-segment (CS) descriptor content (the base address and limit fields) is ignored; the remaining fields function normally (except for the readable bit in the type field).

Code segment descriptors and selectors are needed in IA-32e mode to establish the processor’s operating mode and execution privilege-level. The usage is as follows:
• IA-32e mode uses a previously unused bit in the CS descriptor. Bit 53 is defined as the 64-bit (L) flag and is used to select between 64-bit mode and compatibility mode when IA-32e mode is active (IA32_EFER.LMA = 1). See Figure 5-2.
  — If CS.L = 0 and IA-32e mode is active, the processor is running in compatibility mode. In this case, CS.D selects the default size for data and addresses. If CS.D = 0, the default data and address size is 16 bits. If CS.D = 1, the default data and address size is 32 bits.
  — If CS.L = 1 and IA-32e mode is active, the only valid setting is CS.D = 0. This setting indicates a default operand size of 32 bits and a default address size of 64 bits. The CS.L = 1 and CS.D = 1 bit combination is reserved for future use and a #GP fault will be generated on an attempt to use a code segment with these bits set in IA-32e mode.
• In IA-32e mode, the CS descriptor’s DPL is used for execution privilege checks (as in legacy 32-bit mode).

![Code-Segment Descriptor](image)

Figure 5-2. Descriptor Fields with Flags used in IA-32e Mode

5.3 LIMIT CHECKING

The limit field of a segment descriptor prevents programs or procedures from addressing memory locations outside the segment. The effective value of the limit depends on the setting of the G (granularity) flag (see Figure 5-1). For data segments, the limit also depends on the E (expansion direction) flag and the B (default stack pointer size and/or upper bound) flag. The E flag is one of the bits in the type field when the segment descriptor is for a data-segment type.

When the G flag is clear (byte granularity), the effective limit is the value of the 20-bit limit field in the segment descriptor. Here, the limit ranges from 0 to FFFFH (1 MByte). When the G flag is set (4-KByte page granularity), the processor scales the value in the limit field by a factor of 2^12 (4 KBytes). In this case, the effective limit ranges from FFH (4 KBytes) to FFFFFFFFH (4 GBytes). Note that when scaling is used (G flag is set), the lower 12 bits of a segment offset (address) are not checked against the limit; for example, note that if the segment limit is 0, offsets 0 through FFH are still valid.

For all types of segments except expand-down data segments, the effective limit is the last address that is allowed to be accessed in the segment, which is one less than the size, in bytes, of the segment. The processor causes a general-protection exception (or, if the segment is SS, a stack-fault exception) any time an attempt is made to access the following addresses in a segment:
  • A byte at an offset greater than the effective limit
  • A word at an offset greater than the (effective-limit – 1)
  • A doubleword at an offset greater than the (effective-limit – 3)
  • A quadword at an offset greater than the (effective-limit – 7)
• A double quadword at an offset greater than the \((\text{effective limit} - 15)\)

When the effective limit is FFFFFFFFH (4 GBytes), these accesses may or may not cause the indicated exceptions. Behavior is implementation-specific and may vary from one execution to another.

For expand-down data segments, the segment limit has the same function but is interpreted differently. Here, the effective limit specifies the last address that is not allowed to be accessed within the segment; the range of valid offsets is from \((\text{effective-limit} + 1)\) to FFFFFFFFH if the B flag is set and from \((\text{effective-limit} + 1)\) to FFFFH if the B flag is clear. An expand-down segment has maximum size when the segment limit is 0.

Limit checking catches programming errors such as runaway code, runaway subscripts, and invalid pointer calculations. These errors are detected when they occur, so identification of the cause is easier. Without limit checking, these errors could overwrite code or data in another segment.

In addition to checking segment limits, the processor also checks descriptor table limits. The GDTR and IDTR registers contain 16-bit limit values that the processor uses to prevent programs from selecting a segment descriptors outside the respective descriptor tables. The LDTR and task registers contain 32-bit segment limit value (read from the segment descriptors for the current LDT and TSS, respectively). The processor uses these segment limits to prevent accesses beyond the bounds of the current LDT and TSS. See Section 3.5.1, “Segment Descriptor Tables,” for more information on the GDT and LDT limit fields; see Section 6.10, “Interrupt Descriptor Table (IDT),” for more information on the IDT limit field; and see Section 8.2.4, “Task Register,” for more information on the TSS segment limit field.

5.3.1 Limit Checking in 64-bit Mode

In 64-bit mode, the processor does not perform runtime limit checking on code or data segments. However, the processor does check descriptor-table limits.

5.4 TYPE CHECKING

Segment descriptors contain type information in two places:

• The S (descriptor type) flag.
• The type field.

The processor uses this information to detect programming errors that result in an attempt to use a segment or gate in an incorrect or unintended manner.

The S flag indicates whether a descriptor is a system type or a code or data type. The type field provides 4 additional bits for use in defining various types of code, data, and system descriptors. Table 3-1 shows the encoding of the type field for code and data descriptors; Table 3-2 shows the encoding of the field for system descriptors.

The processor examines type information at various times while operating on segment selectors and segment descriptors. The following list gives examples of typical operations where type checking is performed (this list is not exhaustive):

• **When a segment selector is loaded into a segment register** — Certain segment registers can contain only certain descriptor types, for example:
  — The CS register only can be loaded with a selector for a code segment.
  — Segment selectors for code segments that are not readable or for system segments cannot be loaded into data-segment registers (DS, ES, FS, and GS).
  — Only segment selectors of writable data segments can be loaded into the SS register.

• **When a segment selector is loaded into the LDTR or task register** — For example:
  — The LDTR can only be loaded with a selector for an LDT.
  — The task register can only be loaded with a segment selector for a TSS.

• **When instructions access segments whose descriptors are already loaded into segment registers** — Certain segments can be used by instructions only in certain predefined ways, for example:
  — No instruction may write into an executable segment.
— No instruction may write into a data segment if it is not writable.
— No instruction may read an executable segment unless the readable flag is set.

• **When an instruction operand contains a segment selector** — Certain instructions can access segments or gates of only a particular type, for example:
  — A far CALL or far JMP instruction can only access a segment descriptor for a conforming code segment, nonconforming code segment, call gate, task gate, or TSS.
  — The LLDT instruction must reference a segment descriptor for an LDT.
  — The LTR instruction must reference a segment descriptor for a TSS.
  — The LAR instruction must reference a segment or gate descriptor for an LDT, TSS, call gate, task gate, code segment, or data segment.
  — The LSL instruction must reference a segment descriptor for a LDT, TSS, code segment, or data segment.
  — IDT entries must be interrupt, trap, or task gates.

• **During certain internal operations** — For example:
  — On a far call or far jump (executed with a far CALL or far JMP instruction), the processor determines the type of control transfer to be carried out (call or jump to another code segment, a call or jump through a gate, or a task switch) by checking the type field in the segment (or gate) descriptor pointed to by the segment (or gate) selector given as an operand in the CALL or JMP instruction. If the descriptor type is for a code segment or call gate, a call or jump to another code segment is indicated; if the descriptor type is for a TSS or task gate, a task switch is indicated.
  — On a call or jump through a call gate (or on an interrupt- or exception-handler call through a trap or interrupt gate), the processor automatically checks that the segment descriptor being pointed to by the gate is for a code segment.
  — On a call or jump to a new task through a task gate (or on an interrupt- or exception-handler call to a new task through a task gate), the processor automatically checks that the segment descriptor being pointed to by the task gate is for a TSS.
  — On a call or jump to a new task by a direct reference to a TSS, the processor automatically checks that the segment descriptor being pointed to by the CALL or JMP instruction is for a TSS.
  — On return from a nested task (initiated by an IRET instruction), the processor checks that the previous task link field in the current TSS points to a TSS.

### 5.4.1 Null Segment Selector Checking

Attempting to load a null segment selector (see Section 3.4.2, “Segment Selectors”) into the CS or SS segment register generates a general-protection exception (GP). A null segment selector can be loaded into the DS, ES, FS, or GS register, but any attempt to access a segment through one of these registers when it is loaded with a null segment selector results in a GP exception being generated. Loading unused data-segment registers with a null segment selector is a useful method of detecting accesses to unused segment registers and/or preventing unwanted accesses to data segments.

#### 5.4.1.1 NULL Segment Checking in 64-bit Mode

In 64-bit mode, the processor does not perform runtime checking on NULL segment selectors. The processor does not cause a GP fault when an attempt is made to access memory where the referenced segment register has a NULL segment selector.

### 5.5 PRIVILEGE LEVELS

The processor’s segment-protection mechanism recognizes 4 privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges. Figure 5-3 shows how these levels of privilege can be interpreted as rings of protection.
The center (reserved for the most privileged code, data, and stacks) is used for the segments containing the critical software, usually the kernel of an operating system. Outer rings are used for less critical software. (Systems that use only 2 of the 4 possible privilege levels should use levels 0 and 3.)

The processor uses privilege levels to prevent a program or task operating at a lesser privilege level from accessing a segment with a greater privilege, except under controlled situations. When the processor detects a privilege level violation, it generates a general-protection exception (#GP).

To carry out privilege-level checks between code segments and data segments, the processor recognizes the following three types of privilege levels:

- **Current privilege level (CPL)** — The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL.

- **Descriptor privilege level (DPL)** — The DPL is the privilege level of a segment or gate. It is stored in the DPL field of the segment or gate descriptor for the segment or gate. When the currently executing code segment attempts to access a segment or gate, the DPL of the segment or gate is compared to the CPL and RPL of the segment or gate selector (as described later in this section). The DPL is interpreted differently, depending on the type of segment or gate being accessed:
  - **Data segment** — The DPL indicates the numerically highest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.
  - **Nonconforming code segment (without using a call gate)** — The DPL indicates the privilege level that a program or task must be at to access the segment. For example, if the DPL of a nonconforming code segment is 0, only programs running at a CPL of 0 can access the segment.
  - **Call gate** — The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the call gate. (This is the same access rule as for a data segment.)
  - **Conforming code segment and nonconforming code segment accessed through a call gate** — The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.
— **TSS** — The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the TSS. (This is the same access rule as for a data segment.)

- **Requested privilege level (RPL)** — The RPL is an override privilege level that is assigned to segment selectors. It is stored in bits 0 and 1 of the segment selector. The processor checks the RPL along with the CPL to determine if access to a segment is allowed. Even if the program or task requesting access to a segment has sufficient privilege to access the segment, access is denied if the RPL is not of sufficient privilege level. That is, if the RPL of a segment selector is numerically greater than the CPL, the RPL overrides the CPL, and vice versa. The RPL can be used to ensure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment. See Section 5.10.4, "Checking Caller Access Privileges (ARPL Instruction),” for a detailed description of the purpose and typical use of the RPL.

Privilege levels are checked when the segment selector of a segment descriptor is loaded into a segment register. The checks used for data access differ from those used for transfers of program control among code segments; therefore, the two kinds of accesses are considered separately in the following sections.

### 5.6 PRIVILEGE LEVEL CHECKING WHEN ACCESSING DATA SEGMENTS

To access operands in a data segment, the segment selector for the data segment must be loaded into the data-segment registers (DS, ES, FS, or GS) or into the stack-segment register (SS). (Segment registers can be loaded with the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions.) Before the processor loads a segment selector into a segment register, it performs a privilege check (see Figure 5-4) by comparing the privilege levels of the currently running program or task (the CPL), the RPL of the segment selector, and the DPL of the segment’s segment descriptor. The processor loads the segment selector into the segment register if the DPL is numerically greater than or equal to both the CPL and the RPL. Otherwise, a general-protection fault is generated and the segment register is not loaded.

![Figure 5-4. Privilege Check for Data Access](image)

Figure 5-5 shows four procedures (located in codes segments A, B, C, and D), each running at different privilege levels and each attempting to access the same data segment.

1. The procedure in code segment A is able to access data segment E using segment selector E1, because the CPL of code segment A and the RPL of segment selector E1 are equal to the DPL of data segment E.

2. The procedure in code segment B is able to access data segment E using segment selector E2, because the CPL of code segment B and the RPL of segment selector E2 are both numerically lower than (more privileged) than the DPL of data segment E. A code segment B procedure can also access data segment E using segment selector E1.

3. The procedure in code segment C is not able to access data segment E using segment selector E3 (dotted line), because the CPL of code segment C and the RPL of segment selector E3 are both numerically greater than (less privileged) than the DPL of data segment E. Even if a code segment C procedure were to use segment selector...
E1 or E2, such that the RPL would be acceptable, it still could not access data segment E because its CPL is not privileged enough.

4. The procedure in code segment D should be able to access data segment E because code segment D’s CPL is numerically less than the DPL of data segment E. However, the RPL of segment selector E3 (which the code segment D procedure is using to access data segment E) is numerically greater than the DPL of data segment E, so access is not allowed. If the code segment D procedure were to use segment selector E1 or E2 to access the data segment, access would be allowed.

As demonstrated in the previous examples, the addressable domain of a program or task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

The RPL of a segment selector can always override the addressable domain of a program or task. When properly used, RPLs can prevent problems caused by accidental (or intensional) use of segment selectors for privileged data segments by less privileged programs or procedures.

It is important to note that the RPL of a segment selector for a data segment is under software control. For example, an application program running at a CPL of 3 can set the RPL for a data-segment selector to 0. With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment. To prevent these types of privilege-level-check violations, a program or procedure can check access privileges whenever it receives a data-segment selector from another procedure (see Section 5.10.4, “Checking Caller Access Privileges (ARPL Instruction)”).

5.6.1 Accessing Data in Code Segments

In some instances it may be desirable to access data structures that are contained in a code segment. The following methods of accessing data in code segments are possible:

- Load a data-segment register with a segment selector for a nonconforming, readable, code segment.
- Load a data-segment register with a segment selector for a conforming, readable, code segment.
- Use a code-segment override prefix (CS) to read a readable, code segment whose selector is already loaded in the CS register.

The same rules for accessing data segments apply to method 1. Method 2 is always valid because the privilege level of a conforming code segment is effectively the same as the CPL, regardless of its DPL. Method 3 is always valid because the DPL of the code segment selected by the CS register is the same as the CPL.
5.7 PRIVILEGE LEVEL CHECKING WHEN LOADING THE SS REGISTER

Privilege level checking also occurs when the SS register is loaded with the segment selector for a stack segment. Here all privilege levels related to the stack segment must match the CPL; that is, the CPL, the RPL of the stack-segment selector, and the DPL of the stack-segment descriptor must be the same. If the RPL and DPL are not equal to the CPL, a general-protection exception (#GP) is generated.

5.8 PRIVILEGE LEVEL CHECKING WHEN TRANSFERRING PROGRAM CONTROL BETWEEN CODE SEGMENTS

To transfer program control from one code segment to another, the segment selector for the destination code segment must be loaded into the code-segment register (CS). As part of this loading process, the processor examines the segment descriptor for the destination code segment and performs various limit, type, and privilege checks. If these checks are successful, the CS register is loaded, program control is transferred to the new code segment, and program execution begins at the instruction pointed to by the EIP register.

Program control transfers are carried out with the JMP, CALL, RET, SYSENTER, SYSEXIT, SYSCALL, SYSRET, INT n, and IRET instructions, as well as by the exception and interrupt mechanisms. Exceptions, interrupts, and the IRET instruction are special cases discussed in Chapter 6, "Interrupt and Exception Handling.” This chapter discusses only the JMP, CALL, RET, SYSENTER, SYSEXIT, SYSCALL, and SYSRET instructions.

A JMP or CALL instruction can reference another code segment in any of four ways:

- The target operand contains the segment selector for the target code segment.
- The target operand points to a call-gate descriptor, which contains the segment selector for the target code segment.
- The target operand points to a TSS, which contains the segment selector for the target code segment.
- The target operand points to a task gate, which points to a TSS, which in turn contains the segment selector for the target code segment.

The following sections describe first two types of references. See Section 8.3, “Task Switching,” for information on transferring program control through a task gate and/or TSS.

The SYSENTER and SYSEXIT instructions are special instructions for making fast calls to and returns from operating system or executive procedures. These instructions are discussed in Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”

The SYSCALL and SYSRET instructions are special instructions for making fast calls to and returns from operating system or executive procedures in 64-bit mode. These instructions are discussed in Section 5.8.8, “Fast System Calls in 64-Bit Mode.”

5.8.1 Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, so privilege-level checks are not performed. The far forms of the JMP, CALL, and RET instructions transfer control to other code segments, so the processor does perform privilege-level checks.

When transferring program control to another code segment without going through a call gate, the processor examines four kinds of privilege level and type information (see Figure 5-6):

- The CPL. (Here, the CPL is the privilege level of the calling code segment; that is, the code segment that contains the procedure that is making the call or jump.)
• The DPL of the segment descriptor for the destination code segment that contains the called procedure.
• The RPL of the segment selector of the destination code segment.
• The conforming (C) flag in the segment descriptor for the destination code segment, which determines whether the segment is a conforming (C flag is set) or nonconforming (C flag is clear) code segment. See Section 3.4.5.1, “Code- and Data-Segment Descriptor Types,” for more information about this flag.

The rules that the processor uses to check the CPL, RPL, and DPL depends on the setting of the C flag, as described in the following sections.

5.8.1.1 Accessing Nonconforming Code Segments

When accessing nonconforming code segments, the CPL of the calling procedure must be equal to the DPL of the destination code segment; otherwise, the processor generates a general-protection exception (#GP). For example in Figure 5-7:

• Code segment C is a nonconforming code segment. A procedure in code segment A can call a procedure in code segment C (using segment selector C1) because they are at the same privilege level (CPL of code segment A is equal to the DPL of code segment C).
• A procedure in code segment B cannot call a procedure in code segment C (using segment selector C2 or C1) because the two code segments are at different privilege levels.
The RPL of the segment selector that points to a nonconforming code segment has a limited effect on the privilege check. The RPL must be numerically less than or equal to the CPL of the calling procedure for a successful control transfer to occur. So, in the example in Figure 5-7, the RPLs of segment selectors C1 and C2 could legally be set to 0, 1, or 2, but not to 3.

When the segment selector of a nonconforming code segment is loaded into the CS register, the privilege level field is not changed; that is, it remains at the CPL (which is the privilege level of the calling procedure). This is true, even if the RPL of the segment selector is different from the CPL.

### 5.8.1.2 Accessing Conforming Code Segments

When accessing conforming code segments, the CPL of the calling procedure may be numerically equal to or greater than (less privileged) the DPL of the destination code segment; the processor generates a general-protection exception (#GP) only if the CPL is less than the DPL. (The segment selector RPL for the destination code segment is not checked if the segment is a conforming code segment.)

In the example in Figure 5-7, code segment D is a conforming code segment. Therefore, calling procedures in both code segment A and B can access code segment D (using either segment selector D1 or D2, respectively), because they both have CPLs that are greater than or equal to the DPL of the conforming code segment. For conforming code segments, the DPL represents the numerically lowest privilege level that a calling procedure may be at to successfully make a call to the code segment.

(Note that segments selectors D1 and D2 are identical except for their respective RPLs. But since RPLs are not checked when accessing conforming code segments, the two segment selectors are essentially interchangeable.)

When program control is transferred to a conforming code segment, the CPL does not change, even if the DPL of the destination code segment is less than the CPL. This situation is the only one where the CPL may be different from the DPL of the current code segment. Also, since the CPL does not change, no stack switch occurs.

Conforming segments are used for code modules such as math libraries and exception handlers, which support applications but do not require access to protected system facilities. These modules are part of the operating system or executive software, but they can be executed at numerically higher privilege levels (less privileged levels). Keeping the CPL at the level of a calling code segment when switching to a conforming code segment...
prevents an application program from accessing nonconforming code segments while at the privilege level (DPL) of a conforming code segment and thus prevents it from accessing more privileged data.

Most code segments are nonconforming. For these segments, program control can be transferred only to code segments at the same level of privilege, unless the transfer is carried out through a call gate, as described in the following sections.

5.8.2 Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides special set of descriptors called gate descriptors. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

Task gates are used for task switching and are discussed in Chapter 8, “Task Management.” Trap and interrupt gates are special kinds of call gates used for calling exception and interrupt handlers. They are described in Chapter 6, “Interrupt and Exception Handling.” This chapter is concerned only with call gates.

5.8.3 Call Gates

Call gates facilitate controlled transfers of program control between different privilege levels. They are typically used only in operating systems or executives that use the privilege-level protection mechanism. Call gates are also useful for transferring program control between 16-bit and 32-bit code segments, as described in Section 22.4, “Transferring Control Among Mixed-Size Code Segments.”

Figure 5-8 shows the format of a call-gate descriptor. A call-gate descriptor may reside in the GDT or in an LDT, but not in the interrupt descriptor table (IDT). It performs six functions:

- It specifies the code segment to be accessed.
- It defines an entry point for a procedure in the specified code segment.
- It specifies the privilege level required for a caller trying to access the procedure.
- If a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.
- It defines the size of values to be pushed onto the target stack: 16-bit gates force 16-bit pushes and 32-bit gates force 32-bit pushes.
- It specifies whether the call-gate descriptor is valid.

The segment selector field in a call gate specifies the code segment to be accessed. The offset field specifies the entry point in the code segment. This entry point is generally to the first instruction of a specific procedure. The DPL field indicates the privilege level of the call gate, which in turn is the privilege level required to access the selected
procedure through the gate. The P flag indicates whether the call-gate descriptor is valid. (The presence of the code segment to which the gate points is indicated by the P flag in the code segment’s descriptor.) The parameter count field indicates the number of parameters to copy from the calling procedures stack to the new stack if a stack switch occurs (see Section 5.8.5, “Stack Switching”). The parameter count specifies the number of words for 16-bit call gates and doublewords for 32-bit call gates.

Note that the P flag in a gate descriptor is normally always set to 1. If it is set to 0, a not present (#NP) exception is generated when a program attempts to access the descriptor. The operating system can use the P flag for special purposes. For example, it could be used to track the number of times the gate is used. Here, the P flag is initially set to 0 causing a trap to the not-present exception handler. The exception handler then increments a counter and sets the P flag to 1, so that on returning from the handler, the gate descriptor will be valid.

### 5.8.3.1 IA-32e Mode Call Gates

Call-gate descriptors in 32-bit mode provide a 32-bit offset for the instruction pointer (EIP); 64-bit extensions double the size of 32-bit mode call gates in order to store 64-bit instruction pointers (RIP). See Figure 5-9:

- The first eight bytes (bytes 7:0) of a 64-bit mode call gate are similar but not identical to legacy 32-bit mode call gates. The parameter-copy-count field has been removed.
- Bytes 11:8 hold the upper 32 bits of the target-segment offset in canonical form. A general-protection exception (#GP) is generated if software attempts to use a call gate with a target offset that is not in canonical form.
- 16-byte descriptors may reside in the same descriptor table with 16-bit and 32-bit descriptors. A type field, used for consistency checking, is defined in bits 12:8 of the 64-bit descriptor’s highest dword (cleared to zero). A general-protection exception (#GP) results if an attempt is made to access the upper half of a 64-bit mode descriptor as a 32-bit mode descriptor.

- Target code segments referenced by a 64-bit call gate must be 64-bit code segments (CS.L = 1, CS.D = 0). If not, the reference generates a general-protection exception, #GP (CS selector).
- Only 64-bit mode call gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). The legacy 32-bit mode call gate type (0CH) is redefined in IA-32e mode as a 64-bit call-gate type; no 32-bit call-gate type exists in IA-32e mode.
• If a far call references a 16-bit call gate type (04H) in IA-32e mode, a general-protection exception (#GP) is generated.

When a call references a 64-bit mode call gate, actions taken are identical to those taken in 32-bit mode, with the following exceptions:
• Stack pushes are made in eight-byte increments.
• A 64-bit RIP is pushed onto the stack.
• Parameter copying is not performed.

Use a matching far-return instruction size for correct operation (returns from 64-bit calls must be performed with a 64-bit operand-size return to process the stack correctly).

5.8.4 Accessing a Code Segment Through a Call Gate

To access a call gate, a far pointer to the gate is provided as a target operand in a CALL or JMP instruction. The segment selector from this pointer identifies the call gate (see Figure 5-10); the offset from the pointer is required, but not used or checked by the processor. (The offset can be set to any value.)

When the processor has accessed the call gate, it uses the segment selector from the call gate to locate the segment descriptor for the destination code segment. (This segment descriptor can be in the GDT or the LDT.) It then combines the base address from the code-segment descriptor with the offset from the call gate to form the linear address of the procedure entry point in the code segment.

As shown in Figure 5-11, four different privilege levels are used to check the validity of a program control transfer through a call gate:
• The CPL (current privilege level).
• The RPL (requestor privilege level) of the call gate’s selector.
• The DPL (descriptor privilege level) of the call gate descriptor.
• The DPL of the segment descriptor of the destination code segment.

The C flag (conforming) in the segment descriptor for the destination code segment is also checked.
The privilege checking rules are different depending on whether the control transfer was initiated with a CALL or a JMP instruction, as shown in Table 5-1.

### Table 5-1. Privilege Check Rules for Call Gates

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Privilege Check Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>CPL ≤ call gate DPL; RPL ≤ call gate DPL</td>
</tr>
<tr>
<td></td>
<td>Destination conforming code segment DPL ≤ CPL</td>
</tr>
<tr>
<td></td>
<td>Destination nonconforming code segment DPL ≤ CPL</td>
</tr>
<tr>
<td>JMP</td>
<td>CPL ≤ call gate DPL; RPL ≤ call gate DPL</td>
</tr>
<tr>
<td></td>
<td>Destination conforming code segment DPL ≤ CPL</td>
</tr>
<tr>
<td></td>
<td>Destination nonconforming code segment DPL = CPL</td>
</tr>
</tbody>
</table>

The DPL field of the call-gate descriptor specifies the numerically highest privilege level from which a calling procedure can access the call gate; that is, to access a call gate, the CPL of a calling procedure must be equal to or less than the DPL of the call gate. For example, in Figure 5-15, call gate A has a DPL of 3. So calling procedures at all CPLs (0 through 3) can access this call gate, which includes calling procedures in code segments A, B, and C. Call gate B has a DPL of 2, so only calling procedures at a CPL of 0, 1, or 2 can access call gate B, which includes calling procedures in code segments B and C. The dotted line shows that a calling procedure in code segment A cannot access call gate B.

The RPL of the segment selector to a call gate must satisfy the same test as the CPL of the calling procedure; that is, the RPL must be less than or equal to the DPL of the call gate. In the example in Figure 5-15, a calling procedure in code segment C can access call gate B using gate selector B2 or B1, but it could not use gate selector B3 to access call gate B.

If the privilege checks between the calling procedure and call gate are successful, the processor then checks the DPL of the code-segment descriptor against the CPL of the calling procedure. Here, the privilege check rules vary between CALL and JMP instructions. Only CALL instructions can use call gates to transfer program control to more privileged (numerically lower privilege level) nonconforming code segments; that is, to nonconforming code segments with a DPL less than the CPL. A JMP instruction can use a call gate only to transfer program control to a nonconforming code segment with a DPL equal to the CPL. CALL and JMP instruction can both transfer program control to a more privileged conforming code segment; that is, to a conforming code segment with a DPL less than or equal to the CPL.
If a call is made to a more privileged (numerically lower privilege level) nonconforming destination code segment, the CPL is lowered to the DPL of the destination code segment and a stack switch occurs (see Section 5.8.5, “Stack Switching”). If a call or jump is made to a more privileged conforming destination code segment, the CPL is not changed and no stack switch occurs.

Call gates allow a single code segment to have procedures that can be accessed at different privilege levels. For example, an operating system located in a code segment may have some services which are intended to be used by both the operating system and application software (such as procedures for handling character I/O). Call gates for these procedures can be set up that allow access at all privilege levels (0 through 3). More privileged call gates (with DPLs of 0 or 1) can then be set up for other operating system services that are intended to be used only by the operating system (such as procedures that initialize device drivers).

5.8.5 Stack Switching

Whenever a call gate is used to transfer program control to a more privileged nonconforming code segment (that is, when the DPL of the nonconforming destination code segment is less than the CPL), the processor automatically switches to the stack for the destination code segment’s privilege level. This stack switching is carried out to prevent more privileged procedures from crashing due to insufficient stack space. It also prevents less privileged procedures from interfering (by accident or intent) with more privileged procedures through a shared stack.

Each task must define up to 4 stacks: one for applications code (running at privilege level 3) and one for each of the privilege levels 2, 1, and 0 that are used. (If only two privilege levels are used [3 and 0], then only two stacks must be defined.) Each of these stacks is located in a separate segment and is identified with a segment selector and an offset into the stack segment (a stack pointer).

The segment selector and stack pointer for the privilege level 3 stack is located in the SS and ESP registers, respectively, when privilege-level-3 code is being executed and is automatically stored on the called procedure’s stack when a stack switch occurs.

Pointers to the privilege level 0, 1, and 2 stacks are stored in the TSS for the currently running task (see Figure 8-2). Each of these pointers consists of a segment selector and a stack pointer (loaded into the ESP register). These initial pointers are strictly read-only values. The processor does not change them while the task is running. They are used only to create new stacks when calls are made to more privileged levels (numerically lower
privilege levels). These stacks are disposed of when a return is made from the called procedure. The next time the procedure is called, a new stack is created using the initial stack pointer. (The TSS does not specify a stack for privilege level 3 because the processor does not allow a transfer of program control from a procedure running at a CPL of 0, 1, or 2 to a procedure running at a CPL of 3, except on a return.)

The operating system is responsible for creating stacks and stack-segment descriptors for all the privilege levels to be used and for loading initial pointers for these stacks into the TSS. Each stack must be read/write accessible (as specified in the type field of its segment descriptor) and must contain enough space (as specified in the limit field) to hold the following items:

- The contents of the SS, ESP, CS, and EIP registers for the calling procedure.
- The parameters and temporary variables required by the called procedure.
- The EFLAGS register and error code, when implicit calls are made to an exception or interrupt handler.

The stack will need to require enough space to contain many frames of these items, because procedures often call other procedures, and an operating system may support nesting of multiple interrupts. Each stack should be large enough to allow for the worst case nesting scenario at its privilege level.

(If the operating system does not use the processor’s multitasking mechanism, it still must create at least one TSS for this stack-related purpose.)

When a procedure call through a call gate results in a change in privilege level, the processor performs the following steps to switch stacks and begin execution of the called procedure at a new privilege level:

1. Uses the DPL of the destination code segment (the new CPL) to select a pointer to the new stack (segment selector and stack pointer) from the TSS.
2. Reads the segment selector and stack pointer for the stack to be switched to from the current TSS. Any limit violations detected while reading the stack-segment selector, stack pointer, or stack-segment descriptor cause an invalid TSS (#TS) exception to be generated.
3. Checks the stack-segment descriptor for the proper privileges and type and generates an invalid TSS (#TS) exception if violations are detected.
4. Temporarily saves the current values of the SS and ESP registers.
5. Loads the segment selector and stack pointer for the new stack in the SS and ESP registers.
6. Pushes the temporarily saved values for the SS and ESP registers (for the calling procedure) onto the new stack (see Figure 5-13).
7. Copies the number of parameter specified in the parameter count field of the call gate from the calling procedure's stack to the new stack. If the count is 0, no parameters are copied.
8. Pushes the return instruction pointer (the current contents of the CS and EIP registers) onto the new stack.
9. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively, and begins execution of the called procedure.

See the description of the CALL instruction in Chapter 3, "Instruction Set Reference, A-L", in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for a detailed description of the privilege level checks and other protection checks that the processor performs on a far call through a call gate.
The parameter count field in a call gate specifies the number of data items (up to 31) that the processor should copy from the calling procedure’s stack to the stack of the called procedure. If more than 31 data items need to be passed to the called procedure, one of the parameters can be a pointer to a data structure, or the saved contents of the SS and ESP registers may be used to access parameters in the old stack space. The size of the data items passed to the called procedure depends on the call gate size, as described in Section 5.8.3, "Call Gates."

### 5.8.5.1 Stack Switching in 64-bit Mode

Although protection-check rules for call gates are unchanged from 32-bit mode, stack-switch changes in 64-bit mode are different.

When stacks are switched as part of a 64-bit mode privilege-level change through a call gate, a new SS (stack segment) descriptor is not loaded; 64-bit mode only loads an inner-level RSP from the TSS. The new SS is forced to NULL and the SS selector’s RPL field is forced to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INTn, interrupts, and exceptions). The old SS and RSP are saved on the new stack. On a subsequent far RET, the old SS is popped from the stack and loaded into the SS register. See Table 5-2.

#### Table 5-2. 64-Bit-Mode Stack Layout After Far CALL with CPL Change

<table>
<thead>
<tr>
<th>32-bit Mode</th>
<th>IA-32e mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old SS Selector</td>
<td>+12</td>
</tr>
<tr>
<td>Old ESP</td>
<td>+8</td>
</tr>
<tr>
<td>CS Selector</td>
<td>+4</td>
</tr>
<tr>
<td>EIP</td>
<td>0</td>
</tr>
<tr>
<td>&lt; 4 Bytes &gt;</td>
<td>&lt; 8 Bytes &gt;</td>
</tr>
</tbody>
</table>

In 64-bit mode, stack operations resulting from a privilege-level-changing far call or far return are eight-bytes wide and change the RSP by eight. The mode does not support the automatic parameter-copy feature found in 32-bit mode. The call-gate count field is ignored. Software can access the old stack, if necessary, by referencing the old stack-segment selector and stack pointer saved on the new process stack.

In 64-bit mode, far RET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL ≠ 3, IRET allows SS to be loaded with a NULL selector. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent far RET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.
5.8.6 Returning from a Called Procedure

The RET instruction can be used to perform a near return, a far return at the same privilege level, and a far return to a different privilege level. This instruction is intended to execute returns from procedures that were called with a CALL instruction. It does not support returns from a JMP instruction, because the JMP instruction does not save a return instruction pointer on the stack.

A near return only transfers program control within the current code segment; therefore, the processor performs only a limit check. When the processor pops the return instruction pointer from the stack into the EIP register, it checks that the pointer does not exceed the limit of the current code segment.

On a far return at the same privilege level, the processor pops both a segment selector for the code segment being returned to and a return instruction pointer from the stack. Under normal conditions, these pointers should be valid, because they were pushed on the stack by the CALL instruction. However, the processor performs privilege checks to detect situations where the current procedure might have altered the pointer or failed to maintain the stack properly.

A far return that requires a privilege-level change is only allowed when returning to a less privileged level (that is, the DPL of the return code segment is numerically greater than the CPL). The processor uses the RPL field from the CS register value saved for the calling procedure (see Figure 5-13) to determine if a return to a numerically higher privilege level is required. If the RPL is numerically greater (less privileged) than the CPL, a return across privilege levels occurs.

The processor performs the following steps when performing a far return to a calling procedure (see Figures 6-2 and 6-4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for an illustration of the stack contents prior to and after a return):

1. Checks the RPL field of the saved CS register value to determine if a privilege level change is required on the return.

2. Loads the CS and EIP registers with the values on the called procedure’s stack. (Type and privilege level checks are performed on the code-segment descriptor and RPL of the code-segment selector.)

3. (If the RET instruction includes a parameter count operand and the return requires a privilege level change.) Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value (after popping the CS and EIP values), to step past the parameters on the called procedure’s stack. The resulting value in the ESP register points to the saved SS and ESP values for the calling procedure’s stack. (Note that the byte count in the RET instruction must be chosen to match the parameter count in the call gate that the calling procedure referenced when it made the original call multiplied by the size of the parameters.)

4. (If the return requires a privilege level change.) Loads the SS and ESP registers with the saved SS and ESP values and switches back to the calling procedure’s stack. The SS and ESP values for the called procedure’s stack are discarded. Any limit violations detected while loading the stack-segment selector or stack pointer cause a general-protection exception (#GP) to be generated. The new stack-segment descriptor is also checked for type and privilege violations.

5. (If the RET instruction includes a parameter count operand.) Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value, to step past the parameters on the calling procedure’s stack. The resulting ESP value is not checked against the limit of the stack segment. If the ESP value is beyond the limit, that fact is not recognized until the next stack operation.

6. (If the return requires a privilege level change.) Checks the contents of the DS, ES, FS, and GS segment registers. If any of these registers refer to segments whose DPL is less than the new CPL (excluding conforming code segments), the segment register is loaded with a null segment selector.

See the description of the RET instruction in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for a detailed description of the privilege level checks and other protection checks that the processor performs on a far return.

5.8.7 Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processors for the purpose of providing a fast (low overhead) mechanism for calling operating system or executive procedures.
SYSENTER is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSEXIT is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code. SYSENTER can be executed from privilege levels 3, 2, 1, or 0; SYSEXIT can only be executed from privilege level 0.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. This is because SYSENTER does not save any state information for use by SYSEXIT on a return.

The target instruction and stack pointer for these instructions are not specified through instruction operands. Instead, they are specified through parameters entered in MSRs and general-purpose registers.

For SYSENTER, target fields are generated using the following sources:

- **Target code segment** — Reads this from IA32_SYSENTER_CS.
- **Target instruction** — Reads this from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from the IA32_SYSENTER_ESP.

For SYSEXIT, target fields are generated using the following sources:

- **Target code segment** — Computed by adding 16 to the value in the IA32_SYSENTER_CS.
- **Target instruction** — Reads this from EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from ECX.

The SYSENTER and SYSEXIT instructions perform "fast" calls and returns because they force the processor into a predefined privilege level 0 state when SYSENTER is executed and into a predefined privilege level 3 state when SYSEXIT is executed. By forcing predefined and consistent processor states, the number of privilege checks ordinarily required to perform a far call to another privilege levels are greatly reduced. Also, by predefining the target context state in MSRs and general-purpose registers eliminates all memory accesses except when fetching the target code.

Any additional state that needs to be saved to allow a return to the calling procedure must be saved explicitly by the calling procedure or be predefined through programming conventions.

### 5.8.7.1 SYSENTER and SYSEXIT Instructions in IA-32e Mode

For Intel 64 processors, the SYSENTER and SYSEXIT instructions are enhanced to allow fast system calls from user code running at privilege level 3 (in compatibility mode or 64-bit mode) to 64-bit executive procedures running at privilege level 0. IA32_SYSENTER_EIP MSR and IA32_SYSENTER_ESP MSR are expanded to hold 64-bit addresses. If IA-32e mode is inactive, only the lower 32-bit addresses stored in these MSRs are used. The WRMSR instruction ensures that the addresses stored in these MSRs are canonical. Note that, in 64-bit mode, IA32_SYSENTER_CS must not contain a NULL selector.

When SYSENTER transfers control, the following fields are generated and bits set:

- **Target code segment** — Reads non-NULL selector from IA32_SYSENTER_CS.
- **New CS attributes** — CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Reads 64-bit canonical address from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value from IA32_SYSENTER_CS.
- **Stack pointer** — Reads 64-bit canonical address from IA32_SYSENTER_ESP.
- **New SS attributes** — SS base = 0, SS limit = FFFFFFFH.

When the SYSEXIT instruction transfers control to 64-bit mode user code using REX.W, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 32 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 1 (go to 64-bit mode).
- **Target instruction** — Reads 64-bit canonical address in RDX.
- **Stack segment** — Computed by adding 40 to the value of IA32_SYSENTER_CS.
5-8.8 Fast System Calls in 64-Bit Mode

The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used). The instructions, along with SYSENTER and SYSEXIT, are suited for IA-32e mode operation. SYSCALL and SYSRET, however, are not supported in compatibility mode (or in protected mode). Use CPUID to check if SYSCALL and SYSRET are available (CPUID.80000001H.EDX[bit 11] = 1).

SYSCALL is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSRET is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code.

Stack pointers for SYSCALL/SYSRET are not specified through model specific registers. The clearing of bits in RFLAGS is programmable rather than fixed. SYSCALL/SYSRET save and restore the RFLAGS register.

For SYSCALL, the processor saves RFLAGS into R11 and the RIP of the next instruction into RCX; it then gets the privilege-level 0 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[47:32].
- **Target instruction pointer** — Reads a 64-bit address from IA32_LSTAR. (The WRMSR instruction ensures that the value of the IA32_LSTAR MSR is canonical.)
- **Stack segment** — Computed by adding 8 to the value in IA32_STAR[47:32].
- **Flags** — The processor sets RFLAGS to the logical-AND of its current value with the complement of the value in the IA32_FMASK MSR.

When SYSRET transfers control to 64-bit mode user code using REX.W, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48] + 16.
- **Target instruction pointer** — Copies the value in RCX into RIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

When SYSRET transfers control to 32-bit mode user code using a 32-bit operand size, it then gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48].
- **Target instruction pointer** — Copies the value in ECX into EIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

It is the responsibility of the OS to ensure the descriptors in the GDT/LDT correspond to the selectors loaded by SYSCALL/SYSRET (consistent with the base, limit, and attribute values forced by the instructions).

See Figure 5-14 for the layout of IA32_STAR, IA32_LSTAR, and IA32_FMASK.
The SYSCALL instruction does not save the stack pointer, and the SYSRET instruction does not restore it. It is likely that the OS system-call handler will change the stack pointer from the user stack to the OS stack. If so, it is the responsibility of software first to save the user stack pointer. This might be done by user code, prior to executing SYSCALL, or by the OS system-call handler after SYSCALL.

Because the SYSRET instruction does not modify the stack pointer, it is necessary for software to switch back to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- **External interrupts.** The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- **Nonmaskable interrupts (NMIs).** The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, "Interrupt Stack Table").
- **General-protection exceptions (#GP).** The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

### 5.9 PRIVILEGED INSTRUCTIONS

Some of the system instructions (called "privileged instructions") are protected from use by application programs. The privileged instructions control system functions (such as the loading of system registers). They can be executed only when the CPL is 0 (most privileged). If one of these instructions is executed when the CPL is not 0, a general-protection exception (#GP) is generated. The following system instructions are privileged instructions:

- **LGDT** — Load GDT register.
- **LLDT** — Load LDT register.
• LTR — Load task register.
• LIDT — Load IDT register.
• MOV (control registers) — Load and store control registers.
• LMSW — Load machine status word.
• CLTS — Clear task-switched flag in register CR0.
• MOV (debug registers) — Load and store debug registers.
• INVD — Invalidate cache, without writeback.
• WBINVD — Invalidate cache, with writeback.
• INVLPG — Invalidate TLB entry.
• HLT — Halt processor.
• RDMSR — Read Model-Specific Registers.
• WRMSR — Write Model-Specific Registers.
• RDPMC — Read Performance-Monitoring Counter.
• RDTSC — Read Time-Stamp Counter.

Some of the privileged instructions are available only in the more recent families of Intel 64 and IA-32 processors (see Section 23.13, "New Instructions In the Pentium and Later IA-32 Processors").

The PCE and TSD flags in register CR4 (bits 4 and 2, respectively) enable the RDPMC and RDTSC instructions, respectively, to be executed at any CPL.

5.10 POINTER VALIDATION

When operating in protected mode, the processor validates all pointers to enforce protection between segments and maintain isolation between privilege levels. Pointer validation consists of the following checks:

1. Checking access rights to determine if the segment type is compatible with its use.
2. Checking read/write rights.
3. Checking if the pointer offset exceeds the segment limit.
4. Checking if the supplier of the pointer is allowed to access the segment.
5. Checking the offset alignment.

The processor automatically performs first, second, and third checks during instruction execution. Software must explicitly request the fourth check by issuing an ARPL instruction. The fifth check (offset alignment) is performed automatically at privilege level 3 if alignment checking is turned on. Offset alignment does not affect isolation of privilege levels.

5.10.1 Checking Access Rights (LAR Instruction)

When the processor accesses a segment using a far pointer, it performs an access rights check on the segment descriptor pointed to by the far pointer. This check is performed to determine if type and privilege level (DPL) of the segment descriptor are compatible with the operation to be performed. For example, when making a far call in protected mode, the segment-descriptor type must be for a conforming or nonconforming code segment, a call gate, a task gate, or a TSS. Then, if the call is to a nonconforming code segment, the DPL of the code segment must be equal to the CPL, and the RPL of the code segment’s segment selector must be less than or equal to the DPL. If type or privilege level are found to be incompatible, the appropriate exception is generated.

To prevent type incompatibility exceptions from being generated, software can check the access rights of a segment descriptor using the LAR (load access rights) instruction. The LAR instruction specifies the segment selector for the segment descriptor whose access rights are to be checked and a destination register. The instruction then performs the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code, data, LDT, call gate, task gate, or TSS segment-descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. If the privilege level and type checks pass, loads the second doubleword of the segment descriptor into the destination register (masked by the value 00FXFF00H, where X indicates that the corresponding 4 bits are undefined) and sets the ZF flag in the EFLAGS register. If the segment selector is not visible at the current privilege level or is an invalid type for the LAR instruction, the instruction does not modify the destination register and clears the ZF flag.

Once loaded in the destination register, software can perform additional checks on the access rights information.

5.10.2 Checking Read/Write Rights (VERR and VERW Instructions)

When the processor accesses any code or data segment it checks the read/write privileges assigned to the segment to verify that the intended read or write operation is allowed. Software can check read/write rights using the VERR (verify for reading) and VERW (verify for writing) instructions. Both these instructions specify the segment selector for the segment being checked. The instructions then perform the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code or data-segment descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. Checks that the segment is readable (for the VERR instruction) or writable (for the VERW instruction).

The VERR instruction sets the ZF flag in the EFLAGS register if the segment is visible at the CPL and readable; the VERW sets the ZF flag if the segment is visible and writable. (Code segments are never writable.) The ZF flag is cleared if any of these checks fail.

5.10.3 Checking That the Pointer Offset Is Within Limits (LSL Instruction)

When the processor accesses any segment it performs a limit check to ensure that the offset is within the limit of the segment. Software can perform this limit check using the LSL (load segment limit) instruction. Like the LAR instruction, the LSL instruction specifies the segment selector for the segment descriptor whose limit is to be checked and a destination register. The instruction then performs the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code, data, LDT, or TSS segment-descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. If the privilege level and type checks pass, loads the unscrambled limit (the limit scaled according to the setting of the G flag in the segment descriptor) into the destination register and sets the ZF flag in the EFLAGS register. If the segment selector is not visible at the current privilege level or is an invalid type for the LSL instruction, the instruction does not modify the destination register and clears the ZF flag.

Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.
5.10.4 Checking Caller Access Privileges (ARPL Instruction)

The requestor’s privilege level (RPL) field of a segment selector is intended to carry the privilege level of a calling procedure (the calling procedure’s CPL) to a called procedure. The called procedure then uses the RPL to determine if access to a segment is allowed. The RPL is said to “weaken” the privilege level of the called procedure to that of the RPL.

Operating-system procedures typically use the RPL to prevent less privileged application programs from accessing data located in more privileged segments. When an operating-system procedure (the called procedure) receives a segment selector from an application program (the calling procedure), it sets the segment selector’s RPL to the privilege level of the calling procedure. Then, when the operating system uses the segment selector to access its associated segment, the processor performs privilege checks using the calling procedure’s privilege level (stored in the RPL) rather than the numerically lower privilege level (the CPL) of the operating-system procedure. The RPL thus ensures that the operating system does not access a segment on behalf of an application program unless that program itself has access to the segment.

Figure 5-15 shows an example of how the processor uses the RPL field. In this example, an application program (located in code segment A) possesses a segment selector (segment selector D1) that points to a privileged data structure (that is, a data structure located in a data segment D at privilege level 0).

The application program cannot access data segment D, because it does not have sufficient privilege, but the operating system (located in code segment C) can. So, in an attempt to access data segment D, the application program executes a call to the operating system and passes segment selector D1 to the operating system as a parameter on the stack. Before passing the segment selector, the (well behaved) application program sets the RPL of the segment selector to its current privilege level (which in this example is 3). If the operating system attempts to access data segment D using segment selector D1, the processor compares the CPL (which is now 0 following the call), the RPL of segment selector D1, and the DPL of data segment D (which is 0). Since the RPL is greater than the DPL, access to data segment D is denied. The processor’s protection mechanism thus protects data segment D from access by the operating system, because application program’s privilege level (represented by the RPL of segment selector B) is greater than the DPL of data segment D.
Now assume that instead of setting the RPL of the segment selector to 3, the application program sets the RPL to 0 (segment selector D2). The operating system can now access data segment D, because its CPL and the RPL of segment selector D2 are both equal to the DPL of data segment D.

Because the application program is able to change the RPL of a segment selector to any value, it can potentially use a procedure operating at a numerically lower privilege level to access a protected data structure. This ability to lower the RPL of a segment selector breaches the processor’s protection mechanism.

Because a called procedure cannot rely on the calling procedure to set the RPL correctly, operating-system procedures (executing at numerically lower privilege-levels) that receive segment selectors from numerically higher privilege-level procedures need to test the RPL of the segment selector to determine if it is at the appropriate level. The ARPL (adjust requested privilege level) instruction is provided for this purpose. This instruction adjusts the RPL of one segment selector to match that of another segment selector.

The example in Figure 5-15 demonstrates how the ARPL instruction is intended to be used. When the operating-system receives segment selector D2 from the application program, it uses the ARPL instruction to compare the RPL of the segment selector with the privilege level of the application program (represented by the code-segment selector pushed onto the stack). If the RPL is less than application program’s privilege level, the ARPL instruction changes the RPL of the segment selector to match the privilege level of the application program (segment selector D1). Using this instruction thus prevents a procedure running at a numerically higher privilege level from accessing numerically lower privilege-level (more privileged) segments by lowering the RPL of a segment selector.

Note that the privilege level of the application program can be determined by reading the RPL field of the segment selector for the application-program’s code segment. This segment selector is stored on the stack as part of the call to the operating system. The operating system can copy the segment selector from the stack into a register for use as an operand for the ARPL instruction.

### 5.10.5 Checking Alignment

When the CPL is 3, alignment of memory references can be checked by setting the AM flag in the CR0 register and the AC flag in the EFLAGS register. Unaligned memory references generate alignment exceptions (#AC). The processor does not generate alignment exceptions when operating at privilege level 0, 1, or 2. See Table 6-7 for a description of the alignment requirements when alignment checking is enabled.

### 5.11 PAGE-LEVEL PROTECTION

Page-level protection can be used alone or applied to segments. When page-level protection is used with the flat memory model, it allows supervisor code and data (the operating system or executive) to be protected from user code and data (application programs). It also allows pages containing code to be write protected. When the segment- and page-level protection are combined, page-level read/write protection allows more protection granularity within segments.

With page-level protection (as with segment-level protection) each memory reference is checked to verify that protection checks are satisfied. All checks are made before the memory cycle is started, and any violation prevents the cycle from starting and results in a page-fault exception being generated. Because checks are performed in parallel with address translation, there is no performance penalty.

The processor performs two page-level protection checks:

- Restriction of addressable domain (supervisor and user modes).
- Page type (read only or read/write).

Violations of either of these checks results in a page-fault exception being generated. See Chapter 6, “Interrupt 14—Page-Fault Exception (#PF),” for an explanation of the page-fault exception mechanism. This chapter describes the protection violations which lead to page-fault exceptions.
5.11.1 Page-Protection Flags

Protection information for pages is contained in two flags in a paging-structure entry (see Chapter 4): the read/write flag (bit 1) and the user/supervisor flag (bit 2). The protection checks use the flags in all paging structures.

5.11.2 Restricting Addressable Domain

The page-level protection mechanism allows restricting access to pages based on two privilege levels:

- Supervisor mode (U/S flag is 0)—(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
- User mode (U/S flag is 1)—(Least privileged) For application code and data.

The segment privilege levels map to the page privilege levels as follows. If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode. When the processor is in supervisor mode, it can access all pages; when in user mode, it can access only user-level pages. (Note that the WP flag in control register CR0 modifies the supervisor permissions, as described in Section 5.11.3, “Page Type.”)

Note that to use the page-level protection mechanism, code and data segments must be set up for at least two segment-based privilege levels: level 0 for supervisor code and data segments and level 3 for user code and data segments. (In this model, the stacks are placed in the data segments.) To minimize the use of segments, a flat memory model can be used (see Section 3.2.1, “Basic Flat Model”).

Here, the user and supervisor code and data segments all begin at address zero in the linear address space and overlay each other. With this arrangement, operating-system code (running at the supervisor level) and application code (running at the user level) can execute as if there are no segments. Protection between operating-system and application code and data is provided by the processor’s page-level protection mechanism.

5.11.3 Page Type

The page-level protection mechanism recognizes two page types:

- Read-only access (R/W flag is 0).
- Read/write access (R/W flag is 1).

When the processor is in supervisor mode and the WP flag in register CR0 is clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored). When the processor is in user mode, it can write only to user-mode pages that are read/write accessible. User-mode pages which are read/write or read-only are readable; supervisor-mode pages are neither readable nor writable from user mode. A page-fault exception is generated on any attempt to violate the protection rules.

Starting with the P6 family, Intel processors allow user-mode pages to be write-protected against supervisor-mode access. Setting CR0.WP = 1 enables supervisor-mode sensitivity to write protected pages. If CR0.WP = 1, read-only pages are not writable from any privilege level. This supervisor write-protect feature is useful for implementing a "copy-on-write" strategy used by some operating systems, such as UNIX*, for task creation (also called forking or spawning). When a new task is created, it is possible to copy the entire address space of the parent task. This gives the child task a complete, duplicate set of the parent's segments and pages. An alternative copy-on-write strategy saves memory space and time by mapping the child's segments and pages to the same segments and pages used by the parent task. A private copy of a page gets created only when one of the tasks writes to the page. By using the WP flag and marking the shared pages as read-only, the supervisor can detect an attempt to write to a page, and can copy the page at that time.

5.11.4 Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page-directory entry (first-level page table) may differ from those of its page-table entry (second-level page table). The processor checks the protection for a page in both its page-directory and the page-table entries. Table 5-3 shows the protection provided by the possible combinations of protection attributes when the WP flag is clear.
5.11.5 Overrides to Page Protection

The following types of memory accesses are checked as if they are privilege-level 0 accesses, regardless of the CPL at which the processor is currently operating:

- Access to segment descriptors in the GDT, LDT, or IDT.
- Access to an inner-privilege-level stack during an inter-privilege-level call or a call to in exception or interrupt handler, when a change of privilege level occurs.

5.12 COMBINING PAGE AND SEGMENT PROTECTION

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection. If the processor detects a protection violation at either the segment level or the page level, the memory access is not carried out and an exception is generated. If an exception is generated by segmentation, no paging exception is generated.

Page-level protections cannot be used to override segment-level protection. For example, a code segment is by definition not writable. If a code segment is paged, setting the R/W flag for the pages to read-write does not make the pages writable. Attempts to write into the pages will be blocked by segment-level protection checks.

Page-level protection can be used to enhance segment-level protection. For example, if a large read-write data segment is paged, the page-protection mechanism can be used to write-protect individual pages.

Table 5-3. Combined Page-Directory and Page-Table Protection

<table>
<thead>
<tr>
<th>Page-Directory Entry</th>
<th>Page-Table Entry</th>
<th>Combined Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privilege</td>
<td>Access Type</td>
<td>Privilege</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>User</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>User</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>User</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>User</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>Supervisor</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>Supervisor</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>Supervisor</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>Supervisor</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>User</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>User</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Write</td>
<td>User</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
</tr>
</tbody>
</table>

NOTE:
* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. If CR0.WP = 0, supervisor privilege permits read-write access.
5.13 PAGE-LEVEL PROTECTION AND EXECUTE-DISABLE BIT

In addition to page-level protection offered by the U/S and R/W flags, paging structures used with PAE paging, 4-level paging, and 5-level paging provide the execute-disable bit (see Chapter 4, “Paging”). This bit offers additional protection for data pages.

An Intel 64 or IA-32 processor with the execute-disable bit capability can prevent data pages from being used by malicious software to execute code. This capability is provided in:

- 32-bit protected mode with PAE enabled.
- IA-32e mode.

While the execute-disable bit capability does not introduce new instructions, it does require operating systems to use a PAE-enabled environment and establish a page-granular protection policy for memory pages.

If the execute-disable bit of a memory page is set, that page can be used only as data. An attempt to execute code from a memory page with the execute-disable bit set causes a page-fault exception.

The execute-disable capability is not supported with 32-bit paging. Existing page-level protection mechanisms (see Section 5.11, “Page-Level Protection”) continue to apply to memory pages independent of the execute-disable setting.

5.13.1 Detecting and Enabling the Execute-Disable Capability

Software can detect the presence of the execute-disable capability using the CPUID instruction. CPUID.80000001H:EDX.NX [bit 20] = 1 indicates the capability is available.

If the capability is available, software can enable it by setting IA32_EFER.NXE[bit 11] to 1. IA32_EFER is available if CPUID.80000001H.EDX[bit 20 or 29] = 1.

If the execute-disable capability is not available, a write to set IA32_EFER.NXE produces a #GP exception. See Table 5-4.

Table 5-4. Extended Feature Enable MSR (IA32_EFER)

<table>
<thead>
<tr>
<th>63:12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7:1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Execute-disable bit enable (NXE)</td>
<td>IA-32e mode active (LMA)</td>
<td>Reserved</td>
<td>IA-32e mode enable (LME)</td>
<td>Reserved</td>
<td>SysCall enable (SCE)</td>
</tr>
</tbody>
</table>

5.13.2 Execute-Disable Page Protection

The execute-disable bit in the paging structures enhances page protection for data pages. Instructions cannot be fetched from a memory page if IA32_EFER.NXE = 1 and the execute-disable bit is set in any of the paging-structure entries used to map the page. Table 5-5 lists the valid usage of a page in relation to the value of execute-disable bit (bit 63) of the corresponding entry in each level of the paging structures. Execute-disable protection can be activated using the execute-disable bit at any level of the paging structure, irrespective of the corresponding entry in other levels. When execute-disable protection is not activated, the page can be used as code or data.

---

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.
### Table 5-5. Page Level Protection Matrix with Execute-Disable Bit Capability with 4-Level Paging

<table>
<thead>
<tr>
<th>Execute Disable Bit Value (Bit 63)</th>
<th>Valid Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PML4</td>
<td></td>
</tr>
<tr>
<td>Bit 63 = 1</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
* Value not checked.

In legacy PAE-enabled mode, Table 5-6 and Table 5-7 show the effect of setting the execute-disable bit for code and data pages.

### Table 5-6. 4-KByte Page Level Protection Matrix with Execute-Disable Bit Capability with PAE Paging

<table>
<thead>
<tr>
<th>Execute Disable Bit Value (Bit 63)</th>
<th>Valid Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE</td>
<td></td>
</tr>
<tr>
<td>Bit 63 = 1</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Bit 63 = 0</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**
* Value not checked.

### Table 5-7. 2-MByte Page Level Protection with Execute-Disable Bit Capability with PAE Paging

<table>
<thead>
<tr>
<th>Execute Disable Bit Value (Bit 63)</th>
<th>Valid Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE</td>
<td></td>
</tr>
<tr>
<td>Bit 63 = 1</td>
<td></td>
</tr>
<tr>
<td>Bit 63 = 0</td>
<td></td>
</tr>
</tbody>
</table>

**5.13.3 Reserved Bit Checking**

The processor enforces reserved bit checking in paging data structure entries. The bits being checked varies with paging mode and may vary with the size of physical address space.

Table 5-8 shows the reserved bits that are checked when the execute disable bit capability is enabled (CR4.PAE = 1 and IA32_EFER.NXE = 1). Table 5-8 and Table 5-9 show the following paging modes:

- Non-PAE 4-KByte paging: 4-KByte-page only paging (CR4.PAE = 0, CR4.PSE = 0).
- PSE36: 4-KByte and 4-MByte pages (CR4.PAE = 0, CR4.PSE = 1).
- PAE: 4-KByte and 2-MByte pages (CR4.PAE = 1, CR4.PSE = X).

The reserved bit checking depends on the physical address size supported by the implementation, which is reported in CPUID.80000008H. See the table note.
If execute disable bit capability is not enabled or not available, reserved bit checking in 64-bit mode includes bit 63 and additional bits. This and reserved bit checking for legacy 32-bit paging modes are shown in Table 5-10.

### Table 5-8. Page Level Protection Matrix with Execute-Disable Bit Capability Enabled

<table>
<thead>
<tr>
<th>Mode</th>
<th>Paging Mode</th>
<th>Check Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>4-KByte paging (non-PAE)</td>
<td>No reserved bits checked</td>
</tr>
<tr>
<td></td>
<td>PSE36 - PDE, 4-MByte page</td>
<td>Bit [21]</td>
</tr>
<tr>
<td></td>
<td>PSE36 - PDE, 4-KByte page</td>
<td>No reserved bits checked</td>
</tr>
<tr>
<td></td>
<td>PSE36 - PTE</td>
<td>No reserved bits checked</td>
</tr>
<tr>
<td></td>
<td>PAE - PDE, 4-KByte page</td>
<td>Bits [62:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PAE - PTE</td>
<td>Bits [62:MAXPHYADDR] *</td>
</tr>
<tr>
<td>64-bit</td>
<td>PML5E</td>
<td>Bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PML4E</td>
<td>Bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PDPTE</td>
<td>Bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PDE, 4-KByte page</td>
<td>Bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PTE</td>
<td>Bits [51:MAXPHYADDR] *</td>
</tr>
</tbody>
</table>

**NOTES:**
* MAXPHYADDR is the maximum physical address size and is indicated by CPUID.80000008H:EAX[bits 7-0].

Table 5-9. Reserved Bit Checking with Execute-Disable Bit Capability Not Enabled

<table>
<thead>
<tr>
<th>Mode</th>
<th>Paging Mode</th>
<th>Check Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>KByte paging (non-PAE)</td>
<td>No reserved bits checked</td>
</tr>
<tr>
<td></td>
<td>PSE36 - PDE, 4-MByte page</td>
<td>Bit [21]</td>
</tr>
<tr>
<td></td>
<td>PSE36 - PDE, 4-KByte page</td>
<td>No reserved bits checked</td>
</tr>
<tr>
<td></td>
<td>PSE36 - PTE</td>
<td>No reserved bits checked</td>
</tr>
<tr>
<td></td>
<td>PAE - PDE, 4-KByte page</td>
<td>Bits [63:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PAE - PTE</td>
<td>Bits [63:MAXPHYADDR] *</td>
</tr>
<tr>
<td>64-bit</td>
<td>PML5E</td>
<td>Bit [63], bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PML4E</td>
<td>Bit [63], bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PDPTE</td>
<td>Bit [63], bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PDE, 2-MByte page</td>
<td>Bit [63], bits [51:MAXPHYADDR] &amp; [20:13] *</td>
</tr>
<tr>
<td></td>
<td>PDE, 4-KByte page</td>
<td>Bit [63], bits [51:MAXPHYADDR] *</td>
</tr>
<tr>
<td></td>
<td>PTE</td>
<td>Bit [63], bits [51:MAXPHYADDR] *</td>
</tr>
</tbody>
</table>

**NOTES:**
* MAXPHYADDR is the maximum physical address size and is indicated by CPUID.80000008H:EAX[bits 7-0].

### 5.13.4 Exception Handling

When execute disable bit capability is enabled (IA32_EFER.NXE = 1), conditions for a page fault to occur include the same conditions that apply to an Intel 64 or IA-32 processor without execute disable bit capability plus the
following new condition: an instruction fetch to a linear address that translates to physical address in a memory page that has the execute-disable bit set.

An Execute Disable Bit page fault can occur at all privilege levels. It can occur on any instruction fetch, including (but not limited to): near branches, far branches, CALL/RET/INT/IRET execution, sequential instruction fetches, and task switches. The execute-disable bit in the page translation mechanism is checked only when:

- IA32_EFER.NXE = 1.
- The instruction translation look-aside buffer (ITLB) is loaded with a page that is not already present in the ITLB.
This chapter describes the interrupt and exception-handling mechanism when operating in protected mode on an Intel 64 or IA-32 processor. Most of the information provided here also applies to interrupt and exception mechanisms used in real-address, virtual-8086 mode, and 64-bit mode.

Chapter 21, “8086 Emulation,” describes information specific to interrupt and exception mechanisms in real-address and virtual-8086 mode. Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” describes information specific to interrupt and exception mechanisms in IA-32e mode and 64-bit sub-mode.

6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT n instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor’s interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

6.2 EXCEPTION AND INTERRUPT VECTORS

To aid in handling exceptions and interrupts, each architecturally defined exception and each interrupt condition requiring special handling by the processor is assigned a unique identification number, called a vector number. The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT). The table provides the entry point to an exception or interrupt handler (see Section 6.10, “Interrupt Descriptor Table (IDT)”).

The allowable range for vector numbers is 0 to 255. Vector numbers in the range 0 through 31 are reserved by the Intel 64 and IA-32 architectures for architecture-defined exceptions and interrupts. Not all of the vector numbers in this range have a currently defined function. The unassigned vector numbers in this range are reserved. Do not use the reserved vector numbers.

Vector numbers in the range 32 to 255 are designated as user-defined interrupts and are not reserved by the Intel 64 and IA-32 architecture. These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt mechanisms (see Section 6.3, “Sources of Interrupts”).
INTERRUPT AND EXCEPTION HANDLING

Table 6-1 shows vector number assignments for architecturally defined exceptions and for the NMI interrupt. This table gives the exception type (see Section 6.5, "Exception Classifications") and indicates whether an error code is saved on the stack for the exception. The source of each predefined exception and the NMI interrupt is also given.

6.3 SOURCES OF INTERRUPTS

The processor receives interrupts from two sources:

• External (hardware generated) interrupts.
• Software-generated interrupts.

6.3.1 External Interrupts

External interrupts are received through pins on the processor or through the local APIC. The primary interrupt pins on Pentium 4, Intel Xeon, P6 family, and Pentium processors are the LINT[1:0] pins, which are connected to the local APIC (see Chapter 11, "Advanced Programmable Interrupt Controller (APIC)"). When the local APIC is enabled, the LINT[1:0] pins can be programmed through the APIC’s local vector table (LVT) to be associated with any of the processor’s exception or interrupt vectors.

When the local APIC is global/hardware disabled, these pins are configured as INTR and NMI pins, respectively. Asserting the INTR pin signals the processor that an external interrupt has occurred. The processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A (see Section 6.2, "Exception and Interrupt Vectors"). Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt vector 2.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
<th>Error Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>Fault</td>
<td>No</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug Exception</td>
<td>Fault</td>
<td>No</td>
<td>Instruction, data, and I/O breakpoints; single-step; and others.</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>NMI Interrupt</td>
<td>Interrupt</td>
<td>No</td>
<td>Nonmaskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>Trap</td>
<td>No</td>
<td>INT3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
<td>No</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>Fault</td>
<td>No</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (Undefined Opcode)</td>
<td>Fault</td>
<td>No</td>
<td>UD instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Abort</td>
<td>Yes (zero)</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>Coprocessor Segment Overrun (reserved)</td>
<td>Fault</td>
<td>No</td>
<td>Floating-point instruction.¹</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Fault</td>
<td>Yes</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Fault</td>
<td>Yes</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack-Segment Fault</td>
<td>Fault</td>
<td>Yes</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Fault</td>
<td>Yes</td>
<td>Any memory reference.</td>
</tr>
</tbody>
</table>

¹ Floating-point instruction includes any instruction that can generate a floating-point exception.
The processor’s local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC’s pins can be directed to the local APIC through the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors). The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC. These processors have dedicated NMI and INTR pins. With these processors, external interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the INTR pin.

Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 32, “System Management Mode.”

### 6.3.2 Maskable Hardware Interrupts

Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a maskable hardware interrupt. Maskable hardware interrupts that can be delivered through the INTR pin include:

<table>
<thead>
<tr>
<th>Vector</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
<th>Error Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>—</td>
<td>(Intel reserved. Do not use.)</td>
<td>Fault</td>
<td>No</td>
<td>x87 FPU floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>x87 FPU Floating-Point Error (Math Fault)</td>
<td>Fault</td>
<td>No</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Fault</td>
<td>Yes (Zero)</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Abort</td>
<td>No</td>
<td>SSE/SSE2/SSE3 floating-point instructions</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>Fault</td>
<td>No</td>
<td>EPT violations</td>
</tr>
<tr>
<td>20</td>
<td>#VE</td>
<td>Virtualization Exception</td>
<td>Fault</td>
<td>No</td>
<td>RET, IRET, RSTOR, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.</td>
</tr>
<tr>
<td>21</td>
<td>#CP</td>
<td>Control Protection Exception</td>
<td>Fault</td>
<td>Yes</td>
<td>RET, IRET, RSTOR, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.</td>
</tr>
<tr>
<td>22-31</td>
<td>—</td>
<td>Intel reserved. Do not use.</td>
<td>Interrupt</td>
<td>External interrupt or INT n instruction.</td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td>—</td>
<td>User Defined (Non-reserved) Interrupts</td>
<td>Interrupt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

1. Processors after the Intel386 processor do not generate this exception.
2. This exception was introduced in the Intel486 processor.
3. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
4. This exception was introduced in the Pentium III processor.
5. This exception can occur only on processors that support the 1-setting of the “EPT-violation #VE’ VM-execution control.

The processor’s local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC’s pins can be directed to the local APIC through the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors). The I/O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LINT[1:0] pins are not available on the Intel486 processor and earlier Pentium processors that do not contain an on-chip local APIC. These processors have dedicated NMI and INTR pins. With these processors, external interrupts are typically generated by a system-based interrupt controller (8259A), with the interrupts being signaled through the INTR pin.

Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 32, “System Management Mode.”
all IA-32 architecture defined interrupt vectors from 0 through 255; those that can be delivered through the local APIC include interrupt vectors 16 through 255.

The IF flag in the EFLAGS register permits all maskable hardware interrupts to be masked as a group (see Section 6.8.1, "Masking Maskable Hardware Interrupts"). Note that when interrupts 0 through 15 are delivered through the local APIC, the APIC indicates the receipt of an illegal vector.

6.3.3 Software-Generated Interrupts

The INT \( n \) instruction permits interrupts to be generated from within software by supplying an interrupt vector number as an operand. For example, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.

Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction. If the processor’s predefined NMI vector is used, however, the response of the processor will not be the same as it would be from an NMI interrupt generated in the normal manner. If vector number 2 (the NMI vector) is used in this instruction, the NMI interrupt handler is called, but the processor’s NMI-handling hardware is not activated.

Interrupts generated in software with the INT \( n \) instruction cannot be masked by the IF flag in the EFLAGS register.

6.4 SOURCES OF EXCEPTIONS

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

6.4.1 Program-Error Exceptions

The processor generates one or more exceptions when it detects program errors during the execution in an application program or the operating system or executive. Intel 64 and IA-32 architectures define a vector number for each processor-detectable exception. Exceptions are classified as faults, traps, and aborts (see Section 6.5, "Exception Classifications").

6.4.2 Software-Generated Exceptions

The INTO, INT1, INT3, and BOUND instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, INT3 causes a breakpoint exception to be generated.

The INT \( n \) instruction can be used to emulate exceptions in software; but there is a limitation.\(^1\) If INT \( n \) provides a vector for one of the architecturally-defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware-generated exception normally produces an error code. The exception handler will still attempt to pop an error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

6.4.3 Machine-Check Exceptions

The P6 family and Pentium processors provide both internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions. These mechanisms are implementation depen-

---

1. The INT \( n \) instruction has opcode CD following by an immediate byte encoding the value of \( n \). In contrast, INT1 has opcode F1 and INT3 has opcode CC.
dent. When a machine-check error is detected, the processor signals a machine-check exception (vector 18) and returns an error code.

See Chapter 6, "Interrupt 18—Machine-Check Exception (#MC),” and Chapter 16, "Machine-Check Architecture,” for more information about the machine-check mechanism.

6.5 EXCEPTION CLASSIFICATIONS

Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

- **Faults** — A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.

- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.

- **Aborts** — An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Abort exceptions are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

**NOTE**

One exception subset normally reported as a fault is not restartable. Such exceptions result in loss of some processor state. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. In this situation, the exception handler sees that the instruction pointer (CS:EIP) has been restored as if the POPAD instruction had not been executed. However, internal processor state (the general-purpose registers) will have been modified. Such cases are considered programming errors. An application causing this class of exceptions should be terminated by the operating system.

6.6 PROGRAM OR TASK RESTART

To allow the restarting of program or task following the handling of an exception or an interrupt, all exceptions (except aborts) are guaranteed to report exceptions on an instruction boundary. All interrupts are guaranteed to be taken on an instruction boundary.

For fault-class exceptions, the return instruction pointer (saved when the processor generates an exception) points to the faulting instruction. So, when a program or task is restarted following the handling of a fault, the faulting instruction is restarted (re-executed). Restarting the faulting instruction is commonly used to handle exceptions that are generated when access to an operand is blocked. The most common example of this type of fault is a page-fault exception (#PF) that occurs when a program or task references an operand located on a page that is not in memory. When a page-fault exception occurs, the exception handler can load the page into memory and resume execution of the program or task by restarting the faulting instruction. To ensure that the restart is handled transparently to the currently executing program or task, the processor saves the necessary registers and stack pointers to allow a restart to the state prior to the execution of the faulting instruction.

For trap-class exceptions, the return instruction pointer points to the instruction following the trapping instruction. If a trap is detected during an instruction which transfers execution, the return instruction pointer reflects the transfer. For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction. All trap exceptions allow program or task restart with no loss of continuity. For example, the overflow exception is a trap exception. Here, the return instruction pointer points to the instruction following the INTO instruction that tested EFLAGS.OF (overflow) flag. The trap handler for this exception resolves the overflow condition. Upon return from the trap handler, program or task execution continues at the instruction following the INTO instruction.
The abort-class exceptions do not support reliable restarting of the program or task. Abort handlers are designed to collect diagnostic information about the state of the processor when the abort exception occurred and then shut down the application and system as gracefully as possible.

Interrupts rigorously support restarting of interrupted programs and tasks without loss of continuity. The return instruction pointer saved for an interrupt points to the next instruction to be executed at the instruction boundary where the processor took the interrupt. If the instruction just executed has a repeat prefix, the interrupt is taken at the end of the current iteration with the registers set to execute the next iteration.

The ability of a P6 family processor to speculatively execute instructions does not affect the taking of interrupts by the processor. Interrupts are taken at instruction boundaries located during the retirement phase of instruction execution; so they are always taken in the “in-order” instruction stream. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the P6 family processors’ microarchitecture and its support for out-of-order instruction execution.

Note that the Pentium processor and earlier IA-32 processors also perform varying amounts of prefetching and preliminary decoding. With these processors as well, exceptions and interrupts are not signaled until actual “in-order” execution of the instructions. For a given code sample, the signaling of exceptions occurs uniformly when the code is executed on any family of IA-32 processors (except where new exceptions or new opcodes have been defined).

6.7 NONMASKABLE INTERRUPT (NMI)

The nonmaskable interrupt (NMI) can be generated in either of two ways:

- External hardware asserts the NMI pin.
- The processor receives a message on the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors) with a delivery mode NMI.

When the processor receives a NMI from either of these sources, the processor handles it immediately by calling the NMI handler pointed to by interrupt vector number 2. The processor also invokes certain hardware conditions to ensure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 6.7.1, “Handling Multiple NMIs”).

Also, when an NMI is received from either of the above sources, it cannot be masked by the IF flag in the EFLAGS register.

It is possible to issue a maskable hardware interrupt (through the INTR pin) to vector 2 to invoke the NMI interrupt handler; however, this interrupt will not truly be an NMI interrupt. A true NMI interrupt that activates the processor’s NMI-handling hardware can only be delivered through one of the mechanisms listed above.

6.7.1 Handling Multiple NMIs

While an NMI interrupt handler is executing, the processor blocks delivery of subsequent NMIs until the next execution of the IRET instruction. This blocking of NMIs prevents nested execution of the NMI handler. It is recommended that the NMI interrupt handler be accessed through an interrupt gate to disable maskable hardware interrupts (see Section 6.8.1, “Masking Maskable Hardware Interrupts”).

An execution of the IRET instruction unblocks NMIs even if the instruction causes a fault. For example, if the IRET instruction executes with EFLAGS.VM = 1 and IOPL of less than 3, a general-protection exception is generated (see Section 21.2.7, “Sensitive Instructions”). In such a case, NMIs are unmasked before the exception handler is invoked.

6.8 ENABLING AND DISABLING INTERRUPTS

The processor inhibits the generation of some interrupts, depending on the state of the processor and of the IF and RF flags in the EFLAGS register, as described in the following sections.
6.8.1 Masking Maskable Hardware Interrupts

The IF flag can disable the servicing of maskable hardware interrupts received on the processor’s INTR pin or through the local APIC (see Section 6.3.2, “Maskable Hardware Interrupts”). When the IF flag is clear, the processor inhibits interrupts delivered to the INTR pin or through the local APIC from generating an internal interrupt request; when the IF flag is set, interrupts delivered to the INTR or through the local APIC pin are processed as normal external interrupts.

The IF flag does not affect non-maskable interrupts (NMIs) delivered to the NMI pin or delivery mode NMI messages delivered through the local APIC, nor does it affect processor generated exceptions. As with the other flags in the EFLAGS register, the processor clears the IF flag in response to a hardware reset.

The fact that the group of maskable hardware interrupts includes the reserved interrupt and exception vectors 0 through 32 can potentially cause confusion. Architecturally, when the IF flag is set, an interrupt for any of the vectors from 0 through 32 can be delivered to the processor through the INTR pin and any of the vectors from 16 through 32 can be delivered through the local APIC. The processor will then generate an interrupt and call the interrupt or exception handler pointed to by the vector number. So for example, it is possible to invoke the page-fault handler through the INTR pin (by means of vector 14); however, this is not a true page-fault exception. It is an interrupt. As with the INT n instruction (see Section 6.4.2, “Software-Generated Exceptions”), when an interrupt is generated through the INTR pin to an exception vector, the processor does not push an error code on the stack, so the exception handler may not operate correctly.

The IF flag can be set or cleared with the STI (set interrupt-enable flag) and CLI (clear interrupt-enable flag) instructions, respectively. These instructions may be executed only if the CPL is equal to or less than the IOPL. A general-protection exception (#GP) is generated if they are executed when the CPL is greater than the IOPL.2 If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI.3 The inhibition ends after delivery of another event (e.g., exception) or the execution of the next instruction.

The IF flag is also affected by the following operations:

- The PUSHF instruction stores all flags on the stack, where they can be examined and modified. The POPF instruction can be used to load the modified flags back into the EFLAGS register.
- Task switches and the POPF and IRET instructions load the EFLAGS register; therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is handled through an interrupt gate, the IF flag is automatically cleared, which disables maskable hardware interrupts. (If an interrupt is handled through a trap gate, the IF flag is not cleared.)


6.8.2 Masking Instruction Breakpoints

The RF (resume) flag in the EFLAGS register controls the response of the processor to instruction-breakpoint conditions (see the description of the RF flag in Section 2.3, “System Flags and Fields in the EFLAGS Register”).

When set, it prevents an instruction breakpoint from generating a debug exception (#DB); when clear, instruction breakpoints will generate debug exceptions. The primary function of the RF flag is to prevent the processor from going into a debug exception loop on an instruction-breakpoint. See Section 18.3.1.1, “Instruction-Breakpoint Exception Condition,” for more information on the use of this flag.

As noted in Section 6.8.3, execution of the MOV or POP instruction to load the SS register suppresses any instruction breakpoint on the next instruction (just as if EFLAGS.RF were 1).

---

2. The effect of the IOPL on these instructions is modified slightly when the virtual mode extension is enabled by setting the VME flag in control register CR4; see Section 21.3, “Interrupt and Exception Handling in Virtual-8086 Mode.” Behavior is also impacted by the PVI flag; see Section 21.4, “Protected-Mode Virtual Interrupts.”

3. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.
6.8.3 Masking Exceptions and Interrupts When Switching Stacks

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX  
MOV ESP, StackTop
```

(Software might also use the POP instruction to load SS and ESP.)

If an interrupt or exception occurs after the new SS segment descriptor has been loaded but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler (assuming that delivery of the interrupt or exception does not itself load a new stack pointer).

To account for this situation, the processor prevents certain events from being delivered after execution of a MOV to SS instruction or a POP to SS instruction. The following items provide details:

- Any instruction breakpoint on the next instruction is suppressed (as if EFLAGS.RF were 1).
- Any data breakpoint on the MOV to SS instruction or POP to SS instruction is inhibited until the instruction boundary following the next instruction.
- Any single-step trap that would be delivered following the MOV to SS instruction or POP to SS instruction (because EFLAGS.TF is 1) is suppressed.
- The suppression and inhibition ends after delivery of an exception or the execution of the next instruction.
- If a sequence of consecutive instructions each loads the SS register (using MOV or POP), only the first is guaranteed to inhibit or suppress events in this way.

Intel recommends that software use the LSS instruction to load the SS register and ESP together. The problem identified earlier does not apply to LSS, and the LSS instruction does not inhibit events as detailed above.

6.9 Prioritization of Concurrent Events

If more than one event is pending at an instruction boundary (between execution of instructions), the processor services them in a predictable order. Table 6-2 shows the priority among classes of event sources.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1 (Highest) | Hardware Reset and Machine Checks  
- RESET  
- Machine Check (#MC) |
| 2 | Trap on Task Switch  
- T flag in TSS is set (#DB) |
| 3 | External Hardware Interventions  
- FLUSH  
- STOPCLK  
- SMI  
- INIT |
| 4 | Traps on the Previous Instruction  
- Trap-class Debug Exceptions (#DB due to TF flag set or data/I-O breakpoint) |
| 5 | Nonmaskable Interrupts (NMI) ¹ |
| 6 | Maskable Hardware Interrupts ¹ |
| 7 | Fault-class Debug Exceptions (#DB due to instruction breakpoint) |

¹ These can be masked.
The processor first services a pending event from the class which has the highest priority, transferring execution to the first instruction of the handler. Lower priority exceptions are discarded; lower priority interrupts are held pending. Discarded exceptions may be re-generated when the event handler returns execution to the point in the program or task where the original event occurred. While the priority among the classes listed in Table 6-2 is consistent across processor implementations, the priority of events within a class is implementation-dependent and may vary from processor to processor.

Table 6-2 specifies the prioritization of events that may be pending at an instruction boundary. It does not specify the prioritization of faults that arise during instruction execution or event delivery (these include #BR, #TS, #NP, #SS, #GP, #PF, #AC, #MF, #XM, #VE, or #CP). It also does not apply to the events generated by the "Call to Interrupt Procedure" instructions (INT n, INTO, INT3, and INT1), as these events are integral to the execution of those instructions and do not occur between instructions.

### 6.10 INTERRUPT DESCRIPTOR TABLE (IDT)

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors (in protected mode). Unlike the GDT, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight (the number of bytes in a gate descriptor). Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 descriptors, because descriptors are required only for the interrupt and exception vectors that may occur. All empty descriptor slots in the IDT should have the present flag for the descriptor set to 0.

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is, \(8N - 1\)).

The IDT may reside anywhere in the linear address space. As shown in Figure 6-1, the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.

The LIDT (load IDT register) and SIDT (store IDT register) instructions load and store the contents of the IDTR register, respectively. The LIDT instruction loads the IDTR register with the base address and limit held in a memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another. The SIDT instruction copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

If a vector references a descriptor beyond the limit of the IDT, a general-protection exception (#GP) is generated.
NOTE
Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/or the blocking of interrupt delivery. IA-32 architecture rules need to be followed for setting up IDTR base/limit/access fields and each field in the gate descriptors. The same apply for the Intel 64 architecture. This includes implicit referencing of the destination code segment through the GDT or LDT and accessing the stack.

Figure 6-1. Relationship of the IDTR and IDT

6.11 IDT DESCRIPTORS

The IDT may contain any of three kinds of gate descriptors:
• Task-gate descriptor
• Interrupt-gate descriptor
• Trap-gate descriptor

Figure 6-2 shows the formats for the task-gate, interrupt-gate, and trap-gate descriptors. The format of a task gate used in an IDT is the same as that of a task gate used in the GDT or an LDT (see Section 8.2.5, “Task-Gate Descriptor”). The task gate contains the segment selector for a TSS for an exception and/or interrupt handler task.

Interrupt and trap gates are very similar to call gates (see Section 5.8.3, “Call Gates”). They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution to a handler procedure in an exception- or interrupt-handler code segment. These gates differ in the way the processor handles the IF flag in the EFLAGS register (see Section 6.12.1.3, “Flag Usage By Exception- or Interrupt-Handler Procedure”).
## 6.12 EXCEPTION AND INTERRUPT HANDLING

The processor handles calls to exception- and interrupt-handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate (see Section 5.8.2, "Gate Descriptors," through Section 5.8.6, "Returning from a Called Procedure"). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a task gate (see Section 8.3, "Task Switching").
6.12.1 Exception- or Interrupt-Handler Procedures

An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of the currently executing task (see Figure 6-3). The segment selector for the gate points to a segment descriptor for an executable code segment in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception- or interrupt-handling procedure.

![Figure 6-3. Interrupt Procedure Call](image-url)
When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figure 6-4).
  c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figure 6-4).
  b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL is less than or equal to the IOPL. See Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for a description of the complete operation performed by the IRET instruction.

If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure’s stack on the return.
6.12.1.1 Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be execute at a numerically lower privilege level, a shadow stack switch occurs. When the shadow stack switch occurs:
  - On a transfer from privilege level 3, if shadow stacks are enabled at privilege level 3 then the SSP is saved to the IA32_PL3_SSP MSR.
  - If shadow stacks are enabled at the privilege level where the handler will execute then the shadow stack for the handler is obtained from one of the following MSRs based on the privilege level at which the handler executes:
    - IA32_PL2_SSP if handler executes at privilege level 2.
    - IA32_PL1_SSP if handler executes at privilege level 1.
    - IA32_PL0_SSP if handler executes at privilege level 0.
  - The SSP obtained is then verified to ensure it points to a valid supervisory shadow stack that is not currently active by verifying a supervisor shadow stack token at the address pointed to by the SSP. The operations performed to verify and acquire the supervisor shadow stack token by making it busy are as described in Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
  - On this new shadow stack, the processor pushes the CS, LIP (CS.base + EIP), and SSP of the interrupted procedure if the interrupted procedure was executing at privilege level less than 3; see Figure 6-5.4
  - If the handler procedure is going to be executed at the same privilege level as the interrupted procedure and shadow stacks are enabled at current privilege level:
    - The processor saves the current state of the CS, LIP (CS.base + EIP), and SSP registers on the current shadow stack; see Figure 6-5.

---

4. If any of these pushes leads to an exception or a VM exit, the supervisor shadow-stack token remains busy.
To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions to enforce return address protection:

- Restores the CS and EIP registers to their values prior to the interrupt or exception.

Figure 6-5. Shadow Stack Usage on Transfers to Interrupt and Exception-Handling Routines
If shadow stack is enabled:

— Compares the values on shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.

— Pops the top-of-stack value (the SSP prior to the interrupt or exception) from shadow stack into SSP register.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs the actions below.

- If shadow stack is enabled at current privilege level:
  - If SSP is not aligned to 8 bytes then causes a control protection exception (#CP(FAR-RET/IRET)).
  - If privilege level of the procedure being returned to is less than 3 (returning to supervisor mode):
    - Compares the values on shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.
    - Temporarily saves the top-of-stack value (the SSP of the procedure being returned to) internally.
  - If a busy supervisor shadow stack token is present at address SSP+24, then marks the token free using operations described in section Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
  - If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode), restores the SSP register from the internally saved value.
  - If the privilege level of the procedure being returned to is 3 (returning to user mode) and shadow stack is enabled at privilege level 3, then restores the SSP register with value of IA32_PL3_SSP MSR.

6.12.1.2 Protection of Exception- and Interrupt-Handler Procedures

The privilege-level protection for exception- and interrupt-handler procedures is similar to that used for ordinary procedure calls when called through a call gate (see Section 5.8.4, “Accessing a Code Segment Through a Call Gate”). The processor does not permit transfer of execution to an exception- or interrupt-handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL.

An attempt to violate this rule results in a general-protection exception (#GP). The protection mechanism for exception- and interrupt-handler procedures is different in the following ways:

- Because interrupt and exception vectors have no RPL, the RPL is not checked on implicit calls to exception and interrupt handlers.

- The processor checks the DPL of the interrupt or trap gate only if an exception or interrupt is generated with an INT n, INT3, or INTO instruction. Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using a software interrupt to access critical exception handlers, such as the page-fault handler, providing that those handlers are placed in more privileged code segments (numerically lower privilege level). For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL of interrupt and trap gates.

Because exceptions and interrupts generally do not occur at predictable times, these privilege rules effectively impose restrictions on the privilege levels at which exception and interrupt- handling procedures can run. Either of the following techniques can be used to avoid privilege-level violations.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data from a data segment, the data segment needs to be accessible from privilege level 3, which would make it unprotected.

- The handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program or task is running at.

5. This check is not performed by execution of the INT1 instruction (opcode F1); it would be performed by execution of INT 1 (opcode CD 01).
6.12.1.3 Flag Usage By Exception- or Interrupt-Handler Procedure

When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the processor clears the TF flag in the EFLAGS register after it saves the contents of the EFLAGS register on the stack. (On calls to exception and interrupt handlers, the processor also clears the VM, RF, and NT flags in the EFLAGS register, after they are saved on the stack.) Clearing the TF flag prevents instruction tracing from affecting interrupt response and ensures that no single-step exception will be delivered after delivery to the handler. A subsequent IRET instruction restores the TF (and VM, RF, and NT) flags to the values in the saved contents of the EFLAGS register on the stack.

The only difference between an interrupt gate and a trap gate is the way the processor handles the IF flag in the EFLAGS register. When accessing an exception- or interrupt-handling procedure through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to its value in the saved contents of the EFLAGS register on the stack. Accessing a handler procedure through a trap gate does not affect the IF flag.

6.12.2 Interrupt Tasks

When an exception or interrupt handler is accessed through a task gate in the IDT, a task switch results. Handling an exception or interrupt with a separate task offers several advantages:

- The entire context of the interrupted program or task is saved automatically.
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

The disadvantage of handling an interrupt with a separate task is that the amount of machine state that must be saved on a task switch makes it slower than using an interrupt gate, resulting in increased interrupt latency.

A task gate in the IDT references a TSS descriptor in the GDT (see Figure 6-6). A switch to the handler task is handled in the same manner as an ordinary task switch (see Section 8.3, “Task Switching”). The link back to the interrupted task is stored in the previous task link field of the handler task’s TSS. If an exception caused an error code to be generated, this error code is copied to the stack of the new task.

When exception- or interrupt-handler tasks are used in an operating system, there are actually two mechanisms that can be used to dispatch tasks: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor’s interrupt mechanism). The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.
NOTE
Because IA-32 architecture tasks are not re-entrant, an interrupt-handler task must disable interrupts between the time it completes handling the interrupt and the time it executes the IRET instruction. This action prevents another interrupt from occurring while the interrupt task’s TSS is still marked busy, which would cause a general-protection (#GP) exception.

6.13 ERROR CODE
When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the format shown in Figure 6-7. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

**EXT**  **External event (bit 0)** — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception. The bit is cleared if the exception occurred during delivery of a software interrupt (INT n, INT3, or INTO).

**IDT**  **Descriptor location (bit 1)** — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.

**TI**  **GDT/LDT (bit 2)** — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

---

6. The bit is also set if the exception occurred during delivery of INT1.
The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment selector was referenced in an operation.

The format of the error code is different for page-fault exceptions (#PF). See the “Interrupt 14—Page-Fault Exception (#PF)” section in this chapter.

The format of the error code is different for control protection exceptions (#CP). See the “Interrupt 21—Control Protection Exception (#CP)” section in this chapter.

The error code is pushed on the stack as a doubleword or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for doubleword pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT[1:0] pins) or the INTn instruction, even if an error code is normally produced for those exceptions.

### 6.14 Exception and Interrupt Handling in 64-Bit Mode

In 64-bit mode, interrupt and exception handling is similar to what has been described for non-64-bit modes. The following are the exceptions:

- All interrupt handlers pointed by the IDT are in 64-bit code (this does not apply to the SMI handler).
- The size of interrupt-stack pushes is fixed at 64 bits; and the processor uses 8-byte, zero extended stores.
- The stack pointer (SS:RSP) is pushed unconditionally on interrupts. In legacy modes, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack-switch mechanism and a new interrupt shadow stack-switch mechanism.
- The alignment of interrupt stack frame is different.

#### 6.14.1 64-Bit Mode IDT

Interrupt and trap gates are 16 bytes in length to provide a 64-bit offset for the instruction pointer (RIP). The 64-bit RIP referenced by interrupt-gate descriptors allows an interrupt service routine to be located anywhere in the linear-address space. See Figure 6-8.
In 64-bit mode, the IDT index is formed by scaling the interrupt vector by 16. The first eight bytes (bytes 7:0) of a 64-bit mode interrupt gate are similar but not identical to legacy 32-bit interrupt gates. The type field (bits 11:8 in bytes 7:4) is described in Table 3-2. The Interrupt Stack Table (IST) field (bits 4:0 in bytes 7:4) is used by the stack switching mechanisms described in Section 6.14.5, "Interrupt Stack Table." Bytes 11:8 hold the upper 32 bits of the target RIP (interrupt segment offset) in canonical form. A general-protection exception (#GP) is generated if software attempts to reference an interrupt gate with a target RIP that is not in canonical form.

The target code segment referenced by the interrupt gate must be a 64-bit code segment (CS.L = 1, CS.D = 0). If the target is not a 64-bit code segment, a general-protection exception (#GP) is generated with the IDT vector number reported as the error code.

Only 64-bit interrupt and trap gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). Legacy 32-bit interrupt or trap gate types (0EH or 0FH) are redefined in IA-32e mode as 64-bit interrupt and trap gate types. No 32-bit interrupt or trap gate type exists in IA-32e mode. If a reference is made to a 16-bit interrupt or trap gate (06H or 07H), a general-protection exception (#GP(0)) is generated.

### 6.14.2 64-Bit Mode Stack Frame

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes. SS:ESP is pushed only on a CPL change. In 64-bit mode, the size of interrupt stack-frame pushes is fixed at eight bytes. This is because only 64-bit mode gates can be referenced. 64-bit mode also pushes SS:RSP unconditionally, rather than only on a CPL change.

When shadow stacks are enabled at the interrupt handler’s privilege level and the interrupted procedure was not executing at a privilege level 3, then the processor pushes the CS:LIP:SSP of the interrupted procedure on the shadow stack of the interrupt handler (where LIP is the linear address of the return address).

Aside from error codes, pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stackframe size across all interrupts. Interrupt service-routine entry points that handle interrupts generated by the INTn instruction or external INTR# signal can push an additional error code place-holder to maintain consistency.

In legacy mode, the stack pointer may be at any alignment when an interrupt or exception causes a stack frame to be pushed. This causes the stack frame and succeeding pushes done by an interrupt handler to be at arbitrary alignments. In IA-32e mode, the RSP is aligned to a 16-byte boundary before pushing the stack frame. The stack frame itself is aligned on a 16-byte boundary when the interrupt handler is called. The processor can arbitrarily realign the new RSP on interrupts because the previous (possibly unaligned) RSP is unconditionally saved on the newly aligned stack. The previous RSP will be automatically restored by a subsequent IRET.
Aligning the stack permits exception and interrupt frames to be aligned on a 16-byte boundary before interrupts are re-enabled. This allows the stack to be formatted for optimal storage of 16-byte XMM registers, which enables the interrupt handler to use faster 16-byte aligned loads and stores (MOVAPS rather than MOVUPS) to save and restore XMM registers.

Although the RSP alignment is always performed when LMA = 1, it is only of consequence for the kernel-mode case where there is no stack switch or IST used. For a stack switch or IST, the OS would have presumably put suitably aligned RSP values in the TSS.

6.14.3 IRET in IA-32e Mode

In IA-32e mode, IRET executes with an 8-byte operand size. There is nothing that forces this requirement. The stack is formatted in such a way that for actions where IRET is required, the 8-byte IRET operand size works correctly.

Because interrupt stack-frame pushes are always eight bytes in IA-32e mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64-bit operand-size prefix. The size of the pop is determined by the address size of the instruction. The SS/ESP/RSP size adjustment is determined by the stack size.

IRET pops SS:RSP unconditionally off the interrupt stack frame only when it is executed in 64-bit mode. In compatibility mode, IRET pops SS:RSP off the stack only if there is a CPL change. This allows legacy applications to execute properly in compatibility mode when using the IRET instruction. 64-bit interrupt service routines that exit with an IRET unconditionally pop SS:RSP off of the interrupt stack frame, even if the target code segment is running in 64-bit mode or at CPL = 0. This is because the original interrupt always pushes SS:RSP.

When shadow stacks are enabled and the target privilege level is not 3, the CS:LIP from the shadow stack frame is compared to the return linear address formed by CS:EIP from the stack. If they do not match then the processor caused a control protection exception (#CP(FAR-RET/IRET)), else the processor pops the SSP of the interrupted procedure from the shadow stack. If the target privilege level is 3 and shadow stacks are enabled at privilege level 3, then the SSP for the interrupted procedure is restored from the IA32_PL3_SSP MSR.

In IA-32e mode, IRET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL ≠ 3, IRET allows SS to be loaded with a NULL selector. As part of the stack switch mechanism, an interrupt or exception sets the new SS to NULL, instead of fetching a new SS selector from the TSS and loading the corresponding descriptor from the GDT or LDT. The new SS selector is set to NULL in order to properly handle returns from subsequent nested far transfers. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent IRET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

6.14.4 Stack Switching in IA-32e Mode

The IA-32 architecture provides a mechanism to automatically switch stack frames in response to an interrupt. The 64-bit extensions of Intel 64 architecture implement a modified version of the legacy stack-switching mechanism and an alternative stack-switching mechanism called the interrupt stack table (IST).

In IA-32 modes, the legacy IA-32 stack-switch mechanism is unchanged. In IA-32e mode, the legacy stack-switch mechanism is modified. When stacks are switched as part of a 64-bit mode privilege-level change (resulting from an interrupt), a new SS descriptor is not loaded. IA-32e mode loads only an inner-level RSP from the TSS. The new SS selector is forced to NULL and the SS selector’s RPL field is set to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INT, interrupts, and exceptions). The old SS and RSP are saved on the new stack (Figure 6-9). On the subsequent IRET, the old SS is popped from the stack and loaded into the SS register.

In summary, a stack switch in IA-32e mode works like the legacy stack switch, except that a new SS selector is not loaded from the TSS. Instead, the new SS is forced to NULL.
In IA-32e mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack-switching mechanism described above. This mechanism unconditionally switches stacks when it is enabled. It can be enabled on an individual interrupt-vector basis using a field in the IDT entry. This means that some interrupt vectors can use the modified legacy mechanism and others can use the IST mechanism.

The IST mechanism is only available in IA-32e mode. It is part of the 64-bit mode TSS. The motivation for the IST mechanism is to provide a method for specific interrupts (such as NMI, double-fault, and machine-check) to always execute on a known good stack. In legacy mode, interrupts can use the task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the legacy task-switch mechanism is not supported in IA-32e mode.

The IST mechanism provides up to seven IST pointers in the TSS. The pointers are referenced by an interrupt-gate descriptor in the interrupt-descriptor table (IDT); see Figure 6-8. The gate descriptor contains a 3-bit IST index field that provides an offset into the IST section of the TSS. Using the IST mechanism, the processor loads the value pointed by an IST pointer into the RSP.

When an interrupt occurs, the new SS selector is forced to NULL and the SS selector’s RPL field is set to the new CPL. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal. If the IST index is zero, the modified legacy stack-switching mechanism described above is used.

To support this stack-switching mechanism with shadow stacks enabled, the processor provides an MSR, IA32_INTERRUPT_SSP_TABLE, to program the linear address of a table of seven shadow stack pointers that are selected using the IST index from the gate descriptor. To switch to a shadow stack selected from the interrupt shadow stack table pointed to by the IA32_INTERRUPT_SSP_TABLE, the processor requires that the shadow stack addresses programmed into this table point to a supervisor shadow stack token; see Figure 6-10.

![Figure 6-9. IA-32e Mode Stack Usage After Privilege Level Change](image-url)
6.15 EXCEPTION AND INTERRUPT REFERENCE

The following sections describe conditions which generate exceptions and interrupts. They are arranged in the order of vector numbers. The information contained in these sections are as follows:

- **Exception Class** — Indicates whether the exception class is a fault, trap, or abort type. Some exceptions can be either a fault or trap type, depending on when the error condition is detected. (This section is not applicable to interrupts.)

- **Description** — Gives a general description of the purpose of the exception or interrupt type. It also describes how the processor handles the exception or interrupt.

- **Exception Error Code** — Indicates whether an error code is saved for the exception. If one is saved, the contents of the error code are described. (This section is not applicable to interrupts.)

- **Saved Instruction Pointer** — Describes which instruction the saved (or return) instruction pointer points to. It also indicates whether the pointer can be used to restart a faulting instruction.

- **Program State Change** — Describes the effects of the exception or interrupt on the state of the currently running program or task and the possibilities of restarting the program or task without loss of continuity.
Interrupt 0—Divide Error Exception (#DE)

Exception Class    Fault.

Description
Indicates the divisor operand for a DIV or IDIV instruction is 0 or that the result cannot be represented in the number of bits specified for the destination operand.

Exception Error Code
None.

Saved Instruction Pointer
Saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change
A program-state change does not accompany the divide error, because the exception occurs before the faulting instruction is executed.
Interrupt 1—Debug Exception (#DB)

**Exception Class**  Trap or Fault. The exception handler can distinguish between traps or faults by examining the contents of DR6 and the other debug registers.

**Description**
Indicates that one or more of several debug-exception conditions has been detected. Whether the exception is a fault or a trap depends on the condition (see Table 6-3). See Chapter 18, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” for detailed information about the debug exceptions.

<table>
<thead>
<tr>
<th>Exception Condition</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch breakpoint</td>
<td>Fault</td>
</tr>
<tr>
<td>Data read or write breakpoint</td>
<td>Trap</td>
</tr>
<tr>
<td>I/O read or write breakpoint</td>
<td>Trap</td>
</tr>
<tr>
<td>General detect condition (in conjunction with in-circuit emulation)</td>
<td>Fault</td>
</tr>
<tr>
<td>Single-step</td>
<td>Trap</td>
</tr>
<tr>
<td>Task-switch</td>
<td>Trap</td>
</tr>
<tr>
<td>Execution of INT11</td>
<td>Trap</td>
</tr>
</tbody>
</table>

**Table 6-3. Debug Exception Conditions and Corresponding Exception Classes**

**NOTES:**
1. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

**Exception Error Code**
None. An exception handler can examine the debug registers to determine which condition caused the exception.

**Saved Instruction Pointer**
Fault — Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap — Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

**Program State Change**
Fault — A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap — A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

The following items detail the treatment of debug exceptions on the instruction boundary following execution of the MOV or the POP instruction that loads the SS register:

- If EFLAGS.TF is 1, no single-step trap is generated.
- If the instruction encounters a data breakpoint, the resulting debug exception is delivered after completion of the instruction after the MOV or POP. This occurs even if the next instruction is INT n, INT3, or INTO.
- Any instruction breakpoint on the instruction after the MOV or POP is suppressed (as if EFLAGS.RF were 1).

Any debug exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a debug exception instead causes execution to roll back to just before the XBEGIN instruction.
Interrupt 2—NMI Interrupt

Exception Class  Not applicable.

Description
The nonmaskable interrupt (NMI) is generated externally by asserting the processor’s NMI pin or through an NMI request set by the I/O APIC to the local APIC. This interrupt causes the NMI interrupt handler to be called.

Exception Error Code
Not applicable.

Saved Instruction Pointer
The processor always takes an NMI interrupt on an instruction boundary. The saved contents of CS and EIP registers point to the next instruction to be executed at the point the interrupt is taken. See Section 6.5, “Exception Classifications,” for more information about when the processor takes NMI interrupts.

Program State Change
The instruction executing when an NMI interrupt is received is completed before the NMI is generated. A program or task can thus be restarted upon returning from an interrupt handler without loss of continuity, provided the interrupt handler saves the state of the processor before handling the interrupt and restores the processor’s state prior to a return.
Interrupt 3—Breakpoint Exception (#BP)

Exception Class  
Trap.

Description
Indicates that a breakpoint instruction (INT3, opcode CC) was executed, causing a breakpoint trap to be generated. Typically, a debugger sets a breakpoint by replacing the first opcode byte of an instruction with the opcode for the INT3 instruction. (The INT3 instruction is one byte long, which makes it easy to replace an opcode in a code segment in RAM with the breakpoint opcode.) The operating system or a debugging tool can use a data segment mapped to the same physical address space as the code segment to place an INT3 instruction in places where it is desired to call the debugger.

With the P6 family, Pentium, Intel486, and Intel386 processors, it is more convenient to set breakpoints with the debug registers. (See Section 18.3.2, “Breakpoint Exception (#BP)—Interrupt Vector 3,” for information about the breakpoint exception.) If more breakpoints are needed beyond what the debug registers allow, the INT3 instruction can be used.

Any breakpoint exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a break exception instead causes execution to roll back to just before the XBEGIN instruction and then delivers a debug exception (#DB) — not a breakpoint exception. See Section 16.3.7, “RTM-Enabled Debugger Support,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

A breakpoint exception can also be generated by executing the INT n instruction with an operand of 3. The action of this instruction (INT 3) is slightly different than that of the INT3 instruction (see “INT n/INTO/INT3/INT1—Call to Interrupt Procedure” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

Exception Error Code
None.

Saved Instruction Pointer
Saved contents of CS and EIP registers point to the instruction following the INT3 instruction.

Program State Change
Even though the EIP points to the instruction following the breakpoint instruction, the state of the program is essentially unchanged because the INT3 instruction does not affect any register or memory locations. The debugger can thus resume the suspended program by replacing the INT3 instruction that caused the breakpoint with the original opcode and decrementing the saved contents of the EIP register. Upon returning from the debugger, program execution resumes with the replaced instruction.
Interrupt 4—Overflow Exception (#OF)

Exception Class Trap.

Description
Indicates that an overflow trap occurred when an INTO instruction was executed. The INTO instruction checks the state of the OF flag in the EFLAGS register. If the OF flag is set, an overflow trap is generated.

Some arithmetic instructions (such as the ADD and SUB) perform both signed and unsigned arithmetic. These instructions set the OF and CF flags in the EFLAGS register to indicate signed overflow and unsigned overflow, respectively. When performing arithmetic on signed operands, the OF flag can be tested directly or the INTO instruction can be used. The benefit of using the INTO instruction is that if the overflow exception is detected, an exception handler can be called automatically to handle the overflow condition.

Exception Error Code
None.

Saved Instruction Pointer
The saved contents of CS and EIP registers point to the instruction following the INTO instruction.

Program State Change
Even though the EIP points to the instruction following the INTO instruction, the state of the program is essentially unchanged because the INTO instruction does not affect any register or memory locations. The program can thus resume normal execution upon returning from the overflow exception handler.
Interrupt 5—BOUND Range Exceeded Exception (#BR)

**Exception Class**      Fault.

**Description**
Indicates that a BOUND-range-exceeded fault occurred when a BOUND instruction was executed. The BOUND instruction checks that a signed array index is within the upper and lower bounds of an array located in memory. If the array index is not within the bounds of the array, a BOUND-range-exceeded fault is generated.

**Exception Error Code**
None.

**Saved Instruction Pointer**
The saved contents of CS and EIP registers point to the BOUND instruction that generated the exception.

**Program State Change**
A program-state change does not accompany the bounds-check fault, because the operands for the BOUND instruction are not modified. Returning from the BOUND-range-exceeded exception handler causes the BOUND instruction to be restarted.
Interrupt 6—Invalid Opcode Exception (#UD)

Exception Class: Fault.

Description
Indicates that the processor did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode; for example, the source operand for a LES instruction is not a memory location.
- Attempted to execute an MMX or SSE/SSE2/SSE3 instruction on an Intel 64 or IA-32 processor that does not support the MMX technology or SSE/SSE2/SSE3/SSSE3 extensions, respectively. CPUID feature flags MMX (bit 23), SSE (bit 25), SSE2 (bit 26), SSE3 (ECX, bit 0), SSSE3 (ECX, bit 9) indicate support for these extensions.
- Attempted to execute an MMX instruction or SSE/SSE2/SSE3/SSSE3 SIMD instruction (with the exception of the MOVNTI, PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, CLFLUSH, MONITOR, and MWAIT instructions) when the EM flag in control register CR0 is set (1).
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction when the OSFXSR bit in control register CR4 is clear (0). Note this does not include the following SSE/SSE2/SSE3 instructions: MASKMOVQ, MOVNTQ, MOVNTI, PREFETCHh, SFENCE, LFENCE, MFENCE, and CLFLUSH; or the 64-bit versions of the PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSUFWB, PADDQ, PSUBQ, PALIGNR, PABSB, PABSD, PABS, PHADD, PHADDSW, PHADDQ, PHSUB, PHSUBW, PHSUBD, PHSUBSW, PHSUBQ, PMADDUBSM, PMULHRSW, PSHUFB, PALIGNR, PSIGNB, PSIGND, and PSIGNW.
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction on an Intel 64 or IA-32 processor that caused a SIMD floating-point exception when the OSXMMEXCPT bit in control register CR4 is clear (0).
- Executed a UD0, UD1 or UD2 instruction. Note that even though it is the execution of the UD0, UD1 or UD2 instruction that causes the invalid opcode exception, the saved instruction pointer will still point at the UD0, UD1 or UD2 instruction.
- Detected a LOCK prefix that precedes an instruction that may not be locked or one that may be locked but the destination operand is not a memory location.
- Attempted to execute an LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW, or ARPL instruction while in real-address or virtual-8086 mode.
- Attempted to execute the RSM instruction when not in SMM mode.

In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 6.5, “Exception Classifications,” for general rules for taking of interrupts and exceptions.)

The opcodes D6 and F1 are undefined opcodes reserved by the Intel 64 and IA-32 architectures. These opcodes, even though undefined, do not generate an invalid opcode exception.

Exception Error Code
None.

Saved Instruction Pointer
The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change
A program-state change does not accompany an invalid-opcode fault, because the invalid instruction is not executed.
Interrupt 7—Device Not Available Exception (#NM)

Exception Class Fault.

Description
Indicates one of the following things:
The device-not-available exception is generated by either of three conditions:
• The processor executed an x87 FPU floating-point instruction while the EM flag in control register CR0 was set (1). See the paragraph below for the special case of the WAIT/FWAIT instruction.
• The processor executed a WAIT/FWAIT instruction while the MP and TS flags of register CR0 were set, regardless of the setting of the EM flag.
• The processor executed an x87 FPU, MMX, or SSE/SSE2/SSE3 instruction (with the exception of MOVNTI, PAUSE, PREFETCH, SFENCE, LFENCE, MFENCE, and CLFLUSH) while the TS flag in control register CR0 was set and the EM flag is clear.

The EM flag is set when the processor does not have an internal x87 FPU floating-point unit. A device-not-available exception is then generated each time an x87 FPU floating-point instruction is encountered, allowing an exception handler to call floating-point instruction emulation routines.

The TS flag indicates that a context switch (task switch) has occurred since the last time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction was executed; but that the context of the x87 FPU, XMM, and MXCSR registers were not saved. When the TS flag is set and the EM flag is clear, the processor generates a device-not-available exception each time an x87 floating-point, MMX, or SSE/SSE2/SSE3 instruction is encountered (with the exception of the instructions listed above). The exception handler can then save the context of the x87 FPU, XMM, and MXCSR registers before it executes the instruction. See Section 2.5, “Control Registers,” for more information about the TS flag.

The MP flag in control register CR0 is used along with the TS flag to determine if WAIT or FWAIT instructions should generate a device-not-available exception. It extends the function of the TS flag to the WAIT and FWAIT instructions, giving the exception handler an opportunity to save the context of the x87 FPU before the WAIT or FWAIT instruction is executed. The MP flag is provided primarily for use with the Intel 286 and Intel386 DX processors. For programs running on the Pentium 4, Intel Xeon, P6 family, Pentium, or Intel486 DX processors, or the Intel 487 SX coprocessors, the MP flag should always be set; for programs running on the Intel486 SX processor, the MP flag should be clear.

Exception Error Code
None.

Saved Instruction Pointer
The saved contents of CS and EIP registers point to the floating-point instruction or the WAIT/FWAIT instruction that generated the exception.

Program State Change
A program-state change does not accompany a device-not-available fault, because the instruction that generated the exception is not executed.
If the EM flag is set, the exception handler can then read the floating-point instruction pointed to by the EIP and call the appropriate emulation routine.
If the MP and TS flags are set or the TS flag alone is set, the exception handler can save the context of the x87 FPU, clear the TS flag, and continue execution at the interrupted floating-point or WAIT/FWAIT instruction.
Interrupt 8—Double Fault Exception (#DF)

**Exception Class** Abort.

**Description**
Indicates that the processor detected a second exception while calling an exception handler for a prior exception. Normally, when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception. To determine when two faults need to be signalled as a double fault, the processor divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults (see Table 6-4).

<table>
<thead>
<tr>
<th>Class</th>
<th>Vector Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign Exceptions and Interrupts</td>
<td>1, 2, 3, 4, 5, 6, 7, 9, 16, 17, 18, 19, All</td>
<td>Debug, NMI Interrupt, Breakpoint, Overflow, BOUND Range Exceeded, Invalid Opcode, Device Not Available, Coprocessor Segment Overrun, Floating-Point Error, Alignment Check, Machine Check, SIMD floating-point, INT n, INTR</td>
</tr>
<tr>
<td>Contributory Exceptions</td>
<td>0, 10, 11, 12, 13, 21</td>
<td>Divide Error, Invalid TSS, Segment Not Present, Stack Fault, General Protection, Control Protection</td>
</tr>
<tr>
<td>Page Faults</td>
<td>14, 20</td>
<td>Page Fault, Virtualization Exception</td>
</tr>
</tbody>
</table>

Table 6-5 shows the various combinations of exception classes that cause a double fault to be generated. A double-fault exception falls in the abort class of exceptions. The program or task cannot be restarted or resumed. The double-fault handler can be used to collect diagnostic information about the state of the machine and/or, when possible, to shut the application and/or system down gracefully or restart the system.
A segment or page fault may be encountered while prefetching instructions; however, this behavior is outside the domain of Table 6-5. Any further faults generated while the processor is attempting to transfer control to the appropriate fault handler could still lead to a double-fault sequence.

<table>
<thead>
<tr>
<th>First Exception</th>
<th>Second Exception</th>
<th>Contributory</th>
<th>Page Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>Handle Exceptions Serially</td>
<td>Handle Exceptions Serially</td>
<td>Handle Exceptions Serially</td>
</tr>
<tr>
<td>Contributory</td>
<td>Handle Exceptions Serially</td>
<td>Generate a Double Fault</td>
<td>Handle Exceptions Serially</td>
</tr>
<tr>
<td>Page Fault</td>
<td>Handle Exceptions Serially</td>
<td>Generate a Double Fault</td>
<td>Generate a Double Fault</td>
</tr>
<tr>
<td>Double Fault</td>
<td>Handle Exceptions Serially</td>
<td>Enter Shutdown Mode</td>
<td>Enter Shutdown Mode</td>
</tr>
</tbody>
</table>

If another contributory or page fault exception occurs while attempting to call the double-fault handler, the processor enters shutdown mode. This mode is similar to the state following execution of an HLT instruction. In this mode, the processor stops executing instructions until an NMI interrupt, SMI interrupt, hardware reset, or INIT# is received. The processor generates a special bus cycle to indicate that it has entered shutdown mode. Software designers may need to be aware of the response of hardware when it goes into shutdown mode. For example, hardware may turn on an indicator light on the front panel, generate an NMI interrupt to record diagnostic information, invoke reset initialization, generate an INIT initialization, or generate an SMI. If any events are pending during shutdown, they will be handled after an wake event from shutdown is processed (for example, A20M# interrupts).

Exception Error Code

Zero. The processor always pushes an error code of 0 onto the stack of the double-fault handler.

Saved Instruction Pointer

The saved contents of CS and EIP registers are undefined.

Program State Change

A program-state following a double-fault exception is undefined. The program or task cannot be resumed or restarted. The only available action of the double-fault exception handler is to collect all possible context information for use in diagnostics and then close the application and/or shut down or reset the processor.

If the double fault occurs when any portion of the exception handling machine state is corrupted, the handler cannot be invoked and the processor must be reset.
Interrupt 9—Coprocessor Segment Overrun

Exception Class  Abort. (Intel reserved; do not use. Recent IA-32 processors do not generate this exception.)

Description
Indicates that an Intel386 CPU-based systems with an Intel 387 math coprocessor detected a page or segment violation while transferring the middle portion of an Intel 387 math coprocessor operand. The P6 family, Pentium, and Intel486 processors do not generate this exception; instead, this condition is detected with a general protection exception (#GP), interrupt 13.

Exception Error Code
None.

Saved Instruction Pointer
The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change
A program-state following a coprocessor segment-overrun exception is undefined. The program or task cannot be resumed or restarted. The only available action of the exception handler is to save the instruction pointer and reinitialize the x87 FPU using the FNINIT instruction.
Interrupt 10—Invalid TSS Exception (#TS)

Exception Class       Fault.

Description
Indicates that there was an error related to a TSS. Such an error might be detected during a task switch or during the execution of instructions that use information from a TSS. Table 6-6 shows the conditions that cause an invalid TSS exception to be generated.

<table>
<thead>
<tr>
<th>Error Code Index</th>
<th>Invalid Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSS segment selector index</td>
<td>The TSS segment limit is less than 67H for 32-bit TSS or less than 2CH for 16-bit TSS.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>During an IRET task switch, the TI flag in the TSS segment selector indicates the LDT.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>During an IRET task switch, the TSS segment selector exceeds descriptor table limit.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>During an IRET task switch, the busy flag in the TSS descriptor indicates an inactive task.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>During a task switch, an attempt to access data in a TSS results in a limit violation or canonical fault.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>During an IRET task switch, the backlink is a NULL selector.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>During an IRET task switch, the backlink points to a descriptor which is not a busy TSS.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The new TSS descriptor is beyond the GDT limit.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The new TSS selector is null on an attempt to lock the new TSS.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The new TSS selector has the TI bit set on an attempt to lock the new TSS.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The new TSS descriptor is not an available TSS descriptor on an attempt to lock the new TSS.</td>
</tr>
<tr>
<td>LDT segment selector index</td>
<td>LDT not valid or not present.</td>
</tr>
<tr>
<td>Stack segment selector index</td>
<td>The stack segment selector exceeds descriptor table limit.</td>
</tr>
<tr>
<td>Stack segment selector index</td>
<td>The stack segment selector is NULL.</td>
</tr>
<tr>
<td>Stack segment selector index</td>
<td>The stack segment descriptor is a non-data segment.</td>
</tr>
<tr>
<td>Stack segment selector index</td>
<td>The stack segment is not writable.</td>
</tr>
<tr>
<td>Stack segment selector index</td>
<td>The stack segment DPL ≠ CPL.</td>
</tr>
<tr>
<td>Stack segment selector index</td>
<td>The stack segment selector RPL ≠ CPL.</td>
</tr>
<tr>
<td>Code segment selector index</td>
<td>The code segment selector exceeds descriptor table limit.</td>
</tr>
<tr>
<td>Code segment selector index</td>
<td>The code segment selector is NULL.</td>
</tr>
<tr>
<td>Code segment selector index</td>
<td>The code segment descriptor is not a code segment type.</td>
</tr>
<tr>
<td>Code segment selector index</td>
<td>The nonconforming code segment DPL ≠ CPL.</td>
</tr>
<tr>
<td>Code segment selector index</td>
<td>The conforming code segment DPL is greater than CPL.</td>
</tr>
<tr>
<td>Data segment selector index</td>
<td>The data segment selector exceeds the descriptor table limit.</td>
</tr>
<tr>
<td>Data segment selector index</td>
<td>The data segment descriptor is not a readable code or data type.</td>
</tr>
<tr>
<td>Data segment selector index</td>
<td>The data segment descriptor is a nonconforming code type and RPL &gt; DPL.</td>
</tr>
<tr>
<td>Data segment selector index</td>
<td>The data segment descriptor is a nonconforming code type and CPL &gt; DPL.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The TSS segment descriptor/upper descriptor is beyond the GDT segment limit.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The TSS segment descriptor is not an available TSS type.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The TSS segment descriptor is an available 286 TSS type in IA-32e mode.</td>
</tr>
</tbody>
</table>
INTERRUPT AND EXCEPTION HANDLING

This exception can be generated either in the context of the original task or in the context of the new task (see Section 8.3, “Task Switching”). Until the processor has completely verified the presence of the new TSS, the exception is generated in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete. Any invalid-TSS conditions detected after this point are handled in the context of the new task. (A task switch is considered complete when the task register is loaded with the segment selector for the new TSS and, if the switch is due to a procedure call or interrupt, the previous task link field of the new TSS references the old TSS.)

The invalid-TSS handler must be a task called using a task gate. Handling this exception inside the faulting TSS context is not recommended because the processor state may not be consistent.

Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception was caused by an event external to the currently running program (for example, if an external interrupt handler using a task gate attempted a task switch to an invalid TSS).

Saved Instruction Pointer

If the exception condition was detected before the task switch was carried out, the saved contents of CS and EIP registers point to the instruction that invoked the task switch. If the exception condition was detected after the task switch was carried out, the saved contents of CS and EIP registers point to the first instruction of the new task.

Program State Change

The ability of the invalid-TSS handler to recover from the fault depends on the error condition that causes the fault. See Section 8.3, “Task Switching,” for more information on the task switch process and the possible recovery actions that can be taken.

If an invalid TSS exception occurs during a task switch, it can occur before or after the commit-to-new-task point. If it occurs before the commit point, no program state change occurs. If it occurs after the commit point (when the segment descriptor information for the new segment selectors have been loaded in the segment registers), the processor will load all the state information from the new TSS before it generates the exception. During a task switch, the processor first loads all the segment registers with segment selectors from the TSS, then checks their contents for validity. If an invalid TSS exception is discovered, the remaining segment registers are loaded but not checked for validity and therefore may not be usable for referencing memory. The invalid TSS handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should load all segment registers before trying to resume the new task; otherwise, general-protection exceptions (#GP) may result later under conditions that make diagnosis more difficult. The Intel recommended way of dealing with this situation is to use a task for the invalid TSS exception handler. The task switch back to the interrupted task from the invalid-TSS exception-handler task will then cause the processor to check the registers as it loads them from the TSS.

<table>
<thead>
<tr>
<th>Error Code Index</th>
<th>Invalid Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSS segment selector index</td>
<td>The TSS segment upper descriptor is not the correct type.</td>
</tr>
<tr>
<td>TSS segment selector index</td>
<td>The TSS segment descriptor contains a non-canonical base.</td>
</tr>
</tbody>
</table>
Interrupt 11—Segment Not Present (#NP)

Exception Class  Fault.

Description
Indicates that the present flag of a segment or gate descriptor is clear. The processor can generate this exception during any of the following operations:

• While attempting to load CS, DS, ES, FS, or GS registers. [Detection of a not-present segment while loading the SS register causes a stack fault exception (#SS) to be generated.] This situation can occur while performing a task switch.
• While attempting to load the LDTR using an LLDT instruction. Detection of a not-present LDT while loading the LDTR during a task switch operation causes an invalid-TSS exception (#TS) to be generated.
• When executing the LTR instruction and the TSS is marked not present.
• While attempting to use a gate descriptor or TSS that is marked segment-not-present, but is otherwise valid.

An operating system typically uses the segment-not-present exception to implement virtual memory at the segment level. If the exception handler loads the segment and returns, the interrupted program or task resumes execution.

A not-present indication in a gate descriptor, however, does not indicate that a segment is not present (because gates do not correspond to segments). The operating system may use the present flag for gate descriptors to trigger exceptions of special significance to the operating system.

A contributory exception or page fault that subsequently referenced a not-present segment would cause a double fault (#DF) to be generated instead of #NP.

Exception Error Code
An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception resulted from either:

• an external event (NMI or INTR) that caused an interrupt, which subsequently referenced a not-present segment
• a benign exception that subsequently referenced a not-present segment

The IDT flag is set if the error code refers to an IDT entry. This occurs when the IDT entry for an interrupt being serviced references a not-present gate. Such an event could be generated by an INT instruction or a hardware interrupt.

Saved Instruction Pointer
The saved contents of CS and EIP registers normally point to the instruction that generated the exception. If the exception occurred while loading segment descriptors for the segment selectors in a new TSS, the CS and EIP registers point to the first instruction in the new task. If the exception occurred while accessing a gate descriptor, the CS and EIP registers point to the instruction that invoked the access (for example a CALL instruction that references a call gate).

Program State Change
If the segment-not-present exception occurs as the result of loading a register (CS, DS, SS, ES, FS, GS, or LDTR), a program-state change does accompany the exception because the register is not loaded. Recovery from this exception is possible by simply loading the missing segment into memory and setting the present flag in the segment descriptor.

If the segment-not-present exception occurs while accessing a gate descriptor, a program-state change does not accompany the exception. Recovery from this exception is possible merely by setting the present flag in the gate descriptor.

If a segment-not-present exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 8.3, “Task Switching”). If it occurs before the commit point, no program state change
INTERRUPT AND EXCEPTION HANDLING

occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The segment-not-present exception handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)
Interrupt 12—Stack Fault Exception (#SS)

Exception Class Fault.

Description
Indicates that one of the following stack related conditions was detected:

- A limit violation is detected during an operation that refers to the SS register. Operations that can cause a limit violation include stack-oriented instructions such as POP, PUSH, CALL, RET, IRET, ENTER, and LEAVE, as well as other memory references which implicitly or explicitly use the SS register (for example, MOV AX, [BP+6] or MOV AX, SS:[EAX+6]). The ENTER instruction generates this exception when there is not enough stack space for allocating local variables.
- A not-present stack segment is detected when attempting to load the SS register. This violation can occur during the execution of a task switch, a CALL instruction to a different privilege level, a return to a different privilege level, an LSS instruction, or a MOV or POP instruction to the SS register.
- A canonical violation is detected in 64-bit mode during an operation that reference memory using the stack pointer register containing a non-canonical memory address.

Recovery from this fault is possible by either extending the limit of the stack segment (in the case of a limit violation) or loading the missing stack segment into memory (in the case of a not-present violation).

In the case of a canonical violation that was caused intentionally by software, recovery is possible by loading the correct canonical value into RSP. Otherwise, a canonical violation of the address in RSP likely reflects some register corruption in the software.

Exception Error Code
If the exception is caused by a not-present stack segment or by overflow of the new stack during an inter-privilege-level call, the error code contains a segment selector for the segment that caused the exception. Here, the exception handler can test the present flag in the segment descriptor pointed to by the segment selector to determine the cause of the exception. For a normal limit violation (on a stack segment already in use) the error code is set to 0.

Saved Instruction Pointer
The saved contents of CS and EIP registers generally point to the instruction that generated the exception. However, when the exception results from attempting to load a not-present stack segment during a task switch, the CS and EIP registers point to the first instruction of the new task.

Program State Change
A program-state change does not generally accompany a stack-fault exception, because the instruction that generated the fault is not executed. Here, the instruction can be restarted after the exception handler has corrected the stack fault condition.

If a stack fault occurs during a task switch, it occurs after the commit-to-new-task point (see Section 8.3, "Task Switching"). Here, the processor loads all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The stack fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that are more difficult to diagnose. (See the Program State Change description for "Interrupt 10—Invalid TSS Exception (#TS)" in this chapter for additional information on how to handle this situation.)
Interrupt 13—General Protection Exception (#GP)

Exception Class  Fault.

Description
Indicates that the processor detected one of a class of protection violations called “general-protection violations.” The conditions that cause this exception to be generated comprise all the protection violations that do not cause other exceptions to be generated (such as, invalid-TSS, segment-not-present, stack-fault, or page-fault exceptions). The following conditions cause general-protection exceptions to be generated:

- Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments.
- Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
- Transferring execution to a segment that is not executable.
- Writing to a code segment or a read-only data segment.
- Reading from an execute-only code segment.
- Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs).
- Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment.
- Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment.
- Loading the SS register with the segment selector of an executable segment or a null segment selector.
- Loading the CS register with a segment selector for a data segment or a null segment selector.
- Switching to a busy task during a call or jump to a TSS.
- Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
- Violating any of the privilege rules described in Chapter 5, “Protection.”
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).
- Loading the CR0 register with a set PG flag (paging enabled) and a clear PE flag (protection disabled).
- Loading the CR0 register with a set NW flag and a clear CD flag.
- Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
- Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handler’s code segment DPL is greater than 0.
- Attempting to write a 1 into a reserved bit of CR4.
- Attempting to execute a privileged instruction when the CPL is not equal to 0 (see Section 5.9, “Privileged Instructions,” for a list of privileged instructions).
- Attempting to execute SGDT, SIDT, SLDT, SMSW, or STR when CR4.UMIP = 1 and the CPL is not equal to 0.
- Writing to a reserved bit in an MSR.
- Accessing a gate that contains a null segment selector.
- Executing the INT n instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.
- The segment selector in a call, interrupt, or trap gate does not point to a code segment.
- The segment selector operand in the LLDT instruction is a local type (TI flag is set) or does not point to a segment descriptor of the LDT type.
- The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
- The target code-segment selector for a call, jump, or return is null.
INTERRUPT AND EXCEPTION HANDLING

- If the PAE and/or PSE flag in control register CR4 is set and the processor detects any reserved bits in a page-directory-pointer-table entry set to 1. These bits are checked during a write to control registers CR0, CR3, or CR4 that causes a reloading of the page-directory-pointer-table entry.
- Attempting to write a non-zero value into the reserved bits of the MXCSR register.
- Executing an SSE/SSE2/SSE3 instruction that attempts to access a 128-bit memory location that is not aligned on a 16-byte boundary when the instruction requires 16-byte alignment. This condition also applies to the stack segment.

A program or task can be restarted following any general-protection exception. If the exception occurs while attempting to call an interrupt handler, the interrupted program can be restartable, but the interrupt may be lost.

Exception Error Code
The processor pushes an error code onto the exception handler’s stack. If the fault condition was detected while loading a segment descriptor, the error code contains a segment selector to or IDT vector number for the descriptor; otherwise, the error code is 0. The source of the selector in an error code may be any of the following:
- An operand of the instruction.
- A selector from a gate which is the operand of the instruction.
- A selector from a TSS involved in a task switch.
- IDT vector number.

Saved Instruction Pointer
The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change
In general, a program-state change does not accompany a general-protection exception, because the invalid instruction or operation is not executed. An exception handler can be designed to correct all of the conditions that cause general-protection exceptions and restart the program or task without any loss of program continuity.

If a general-protection exception occurs during a task switch, it can occur before or after the commit-to-new-task point (see Section 8.3, “Task Switching”). If it occurs before the commit point, no program state change occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The general-protection exception handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

General Protection Exception in 64-bit Mode
The following conditions cause general-protection exceptions in 64-bit mode:
- If the memory address is in a non-canonical form.
- If a segment descriptor memory address is in non-canonical form.
- If the target offset in a destination operand of a call or jmp is in a non-canonical form.
- If a code segment or 64-bit call gate overlaps non-canonical space.
- If the code segment descriptor pointed to by the selector in the 64-bit gate doesn’t have the L-bit set and the D-bit clear.
- If the EFLAGS.NT bit is set in IRET.
- If the stack segment selector of IRET is null when going back to compatibility mode.
- If the stack segment selector of IRET is null going back to CPL3 and 64-bit mode.
- If a null stack segment selector RPL of IRET is not equal to CPL going back to non-CPL3 and 64-bit mode.
- If the proposed new code segment descriptor of IRET has both the D-bit and the L-bit set.
• If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and it has both the D-bit and the L-bit set.
• If the segment descriptor from a 64-bit call gate is in non-canonical space.
• If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
• If the type field of the upper 64 bits of a 64-bit call gate is not 0.
• If an attempt is made to load a null selector in the SS register in compatibility mode.
• If an attempt is made to load null selector in the SS register in CPL3 and 64-bit mode.
• If an attempt is made to load a null selector in the SS register in non-CPL3 and 64-bit mode where RPL is not equal to CPL.
• If an attempt is made to clear CR0.PG while IA-32e mode is enabled.
• If an attempt is made to set a reserved bit in CR3, CR4 or CR8.
Interrupt 14—Page-Fault Exception (#PF)

Exception Class  Fault.

Description
Indicates that, with paging enabled (the PG flag in the CR0 register is set), the processor detected one of the following conditions while using the page-translation mechanism to translate a linear address to a physical address:

- The P (present) flag in a page-directory or page-table entry needed for the address translation is clear, indicating that a page table or the page containing the operand is not present in physical memory.
- The procedure does not have sufficient privilege to access the indicated page (that is, a procedure running in user mode attempts to access a supervisor-mode page). If the SMAP flag is set in CR4, a page fault may also be triggered by code running in supervisor mode that tries to access data at a user-mode address. If either the PKE flag or the PKS flag is set in CR4, the protection-key rights registers may cause page faults on data accesses to linear addresses with certain protection keys.
- Code running in user mode attempts to write to a read-only page. If the WP flag is set in CR0, the page fault will also be triggered by code running in supervisor mode that tries to write to a read-only page.
- An instruction fetch to a linear address that translates to a physical address in a memory page with the execute-disable bit set (for information about the execute-disable bit, see Chapter 4, “Paging”). If the SMEP flag is set in CR4, a page fault will also be triggered by code running in supervisor mode that tries to fetch an instruction from a user-mode address.
- One or more reserved bits in paging-structure entry are set to 1. See description below of RSVD error code flag.
- A shadow-stack access is made to a page that is not a shadow-stack page. See Section 17.2, “Shadow Stacks,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and Section 4.6, “Access Rights.”
- An enclave access violates one of the specified access-control requirements. See Section 35.3, “Access-control Requirements,” and Section 35.20, “Enclave Page Cache Map (EPCM),” in Chapter 35, “Enclave Access Control and Data Structures.” In this case, the exception is called an **SGX-induced page fault**. The processor uses the error code (below) to distinguish SGX-induced page faults from ordinary page faults.

The exception handler can recover from page-not-present conditions and restart the program or task without any loss of program continuity. It can also restart the program or task after a privilege violation, but the problem that caused the privilege violation may be uncorrectable.

See also: Section 4.7, “Page-Fault Exceptions.”

Exception Error Code
Yes (special format). The processor provides the page-fault handler with two items of information to aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 6-11). The processor establishes the bits in the error code as follows:
  - P flag (bit 0).
    This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
  - W/R (bit 1).
    If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
  - U/S (bit 2).
    If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
— RSVD flag (bit 3).
  This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the
  paging-structure entries used to translate that address.

— I/D flag (bit 4).
  This flag is 1 if the access causing the page-fault exception was an instruction fetch. This flag describes the
  access causing the page-fault exception, not the access rights specified by paging.

— PK flag (bit 5).
  This flag is 1 if the access causing the page-fault exception was a data access to a linear address with a
  protection key for which the protection-key rights registers disallow access.

— SS (bit 6).
  If the access causing the page-fault exception was a shadow-stack access (including shadow-stack
  accesses in enclave mode), this flag is 1; otherwise, it is 0. This flag describes the access causing the page-
  fault exception, not the access rights specified by paging.

— HLAT (bit 7).
  This flag is 1 if there is no translation for the linear address using HLAT paging because, in one of the
  paging-structure entries used to translate that address, either the P flag was 0 or a reserved bit was set. An
  error code will set this flag only if it clears bit 0 or sets bit 3. This flag will not be set by a page fault resulting
  from a violation of access rights, nor for one encountered during ordinary paging, including the case in
  which there has been a restart of HLAT paging.

— SGX flag (bit 15).
  This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-
  control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is
  set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

See Section 4.6, "Access Rights,” and Section 4.7, “Page-Fault Exceptions,” for more information about page-
fault exceptions and the error codes that they produce.
The contents of the CR2 register. The processor loads the CR2 register with the 32-bit linear address that generated the exception. The page-fault handler can use this address to locate the corresponding page directory and page-table entries. Another page fault can potentially occur during execution of the page-fault handler; the handler should save the contents of the CR2 register before a second page fault can occur. If a page fault is caused by a page-level protection violation, the access flag in the page-directory entry is set when the fault occurs. The behavior of IA-32 processors regarding the access flag in the corresponding page-table entry is model specific and not architecturally defined.

Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. If the page-fault exception occurred during a task switch, the CS and EIP registers may point to the first instruction of the new task (as described in the following “Program State Change” section).

Program State Change

A program-state change does not normally accompany a page-fault exception, because the instruction that causes the exception to be generated is not executed. After the page-fault exception handler has corrected the violation (for example, loaded the missing page into memory), execution of the program or task can be resumed.

---

7. Processors update CR2 whenever a page fault is detected. If a second page fault occurs while an earlier page fault is being delivered, the faulting linear address of the second fault will overwrite the contents of CR2 (replacing the previous address). These updates to CR2 occur even if the page fault results in a double fault or occurs during the delivery of a double fault.
When a page-fault exception is generated during a task switch, the program-state may change, as follows. During a task switch, a page-fault exception can occur during any of following operations:

- While writing the state of the original task into the TSS of that task.
- While reading the GDT to locate the TSS descriptor of the new task.
- While reading the TSS of the new task.
- While reading segment descriptors associated with segment selectors from the new task.
- While reading the LDT of the new task to verify the segment registers stored in the new TSS.

In the last two cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be called through a task gate.

If a page fault occurs during a task switch, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The page-fault handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

Additional Exception-Handling Information

Special care should be taken to ensure that an exception that occurs during an explicit stack switch does not cause the processor to use an invalid stack pointer (SS:ESP). Software written for 16-bit IA-32 processors often use a pair of instructions to change to a new stack, for example:

```
  MOV SS, AX
  MOV SP, StackTop
```

When executing this code on one of the 32-bit IA-32 processors, it is possible to get a page fault, general-protection fault (#GP), or alignment check fault (#AC) after the segment selector has been loaded into the SS register but before the ESP register has been loaded. At this point, the two parts of the stack pointer (SS and ESP) are inconsistent. The new stack segment is being used with the old stack pointer.

The processor does not use the inconsistent stack pointer if the exception handler switches to a well defined stack (that is, the handler is a task or a more privileged procedure). However, if the exception handler is called at the same privilege level and from the same task, the processor will attempt to use the inconsistent stack pointer.

In systems that handle page-fault, general-protection, or alignment check exceptions within the faulting task (with trap or interrupt gates), software executing at the same privilege level as the exception handler should initialize a new stack by using the LSS instruction rather than a pair of MOV instructions, as described earlier in this note. When the exception handler is running at privilege level 0 (the normal case), the problem is limited to procedures or tasks that run at privilege level 0, typically the kernel of the operating system.
Interrupt 16—x87 FPU Floating-Point Error (#MF)

**Exception Class**  
Fault.

**Description**  
Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, "Control Registers," for a detailed description of the NE flag.)

**NOTE**  
SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

- Invalid operation (#I)
  - Stack overflow or underflow (#IS)
  - Invalid arithmetic operation (#IA)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

Each of these error conditions represents an x87 FPU exception type, and for each of exception type, the x87 FPU provides a flag in the x87 FPU status register and a mask bit in the x87 FPU control register. If the x87 FPU detects a floating-point error and the mask bit for the exception type is set, the x87 FPU handles the exception automatically by generating a predefined (default) response and continuing program execution. The default responses have been designed to provide a reasonable result for most floating-point applications.

If the mask for the exception is clear and the NE flag in register CR0 is set, the x87 FPU does the following:

1. Sets the necessary flag in the FPU status register.
2. Waits until the next “waiting” x87 FPU instruction or WAIT/FWAIT instruction is encountered in the program’s instruction stream.
3. Generates an internal error signal that cause the processor to generate a floating-point exception (#MF).

Prior to executing a waiting x87 FPU instruction or the WAIT/FWAIT instruction, the x87 FPU checks for pending x87 FPU floating-point exceptions (as described in step 2 above). Pending x87 FPU floating-point exceptions are ignored for “non-waiting” x87 FPU instructions, which include the FNINIT, FNCLEX, FNSTSW, FNSTSW AX, FNSTCW, FNSTENV, and FNSAVE instructions. Pending x87 FPU exceptions are also ignored when executing the state management instructions FXSAVE and FXRSTOR.

All of the x87 FPU floating-point error conditions can be recovered from. The x87 FPU floating-point-error exception handler can determine the error condition that caused the exception from the settings of the flags in the x87 FPU status word. See "Software Exception Handling" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information on handling x87 FPU floating-point exceptions.

**Exception Error Code**  
None. The x87 FPU provides its own error information.

**Saved Instruction Pointer**  
The saved contents of CS and EIP registers point to the floating-point or WAIT/FWAIT instruction that was about to be executed when the floating-point-error exception was generated. This is not the faulting instruction in which the error condition was detected. The address of the faulting instruction is contained in the x87 FPU instruction pointer.
interrupt and exception handling

register. See Section 8.1.8, "x87 FPU Instruction and Data (Operand) Pointers," in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about information the FPU saves for use in handling floating-point-error exceptions.

Program State Change

A program-state change generally accompanies an x87 FPU floating-point exception because the handling of the exception is delayed until the next waiting x87 FPU floating-point or WAIT/FWAIT instruction following the faulting instruction. The x87 FPU, however, saves sufficient information about the error condition to allow recovery from the error and re-execution of the faulting instruction if needed.

In situations where non-x87 FPU floating-point instructions depend on the results of an x87 FPU floating-point instruction, a WAIT or FWAIT instruction can be inserted in front of a dependent instruction to force a pending x87 FPU floating-point exception to be handled before the dependent instruction is executed. See "x87 FPU Exception Synchronization" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about synchronization of x87 floating-point-error exceptions.
Interrupt 17—Alignment Check Exception (#AC)

Exception Class  Fault.

Description
Indicates that the processor detected an unaligned memory operand when alignment checking was enabled. Alignment checks are only carried out in data (or stack) accesses (not in code fetches or system segment accesses). An example of an alignment-check violation is a word stored at an odd byte address, or a doubleword stored at an address that is not an integer multiple of 4. Table 6-7 lists the alignment requirements various data types recognized by the processor.

### Table 6-7. Alignment Requirements by Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Address Must Be Divisible By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>2</td>
</tr>
<tr>
<td>Doubleword</td>
<td>4</td>
</tr>
<tr>
<td>Single precision floating-point (32-bits)</td>
<td>4</td>
</tr>
<tr>
<td>Double precision floating-point (64-bits)</td>
<td>8</td>
</tr>
<tr>
<td>Double extended precision floating-point (80-bits)</td>
<td>8</td>
</tr>
<tr>
<td>Quadword</td>
<td>8</td>
</tr>
<tr>
<td>Double quadword</td>
<td>16</td>
</tr>
<tr>
<td>Segment Selector</td>
<td>2</td>
</tr>
<tr>
<td>32-bit Far Pointer</td>
<td>2</td>
</tr>
<tr>
<td>48-bit Far Pointer</td>
<td>4</td>
</tr>
<tr>
<td>32-bit Pointer</td>
<td>4</td>
</tr>
<tr>
<td>GDTR, IDTR, LDTR, or Task Register Contents</td>
<td>4</td>
</tr>
<tr>
<td>FSTENV/FLDENV Save Area</td>
<td>4 or 2, depending on operand size</td>
</tr>
<tr>
<td>FSAVE/FRSTOR Save Area</td>
<td>4 or 2, depending on operand size</td>
</tr>
<tr>
<td>Bit String</td>
<td>2 or 4 depending on the operand-size attribute</td>
</tr>
</tbody>
</table>

Note that the alignment check exception (#AC) is generated only for data types that must be aligned on word, doubleword, and quadword boundaries. A general-protection exception (#GP) is generated 128-bit data types that are not aligned on a 16-byte boundary.

To enable alignment checking, the following conditions must be true:
- AM flag in CR0 register is set.
- AC flag in the EFLAGS register is set.
- The CPL is 3 (including virtual-8086 mode).

Alignment-check exceptions (#AC) are generated only when operating at privilege level 3 (user mode). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate alignment-check exceptions, even when caused by a memory reference made from privilege level 3.

Storing the contents of the GDTR, IDTR, LDTR, or task register in memory while at privilege level 3 can generate an alignment-check exception. Although application programs do not normally store these registers, the fault can be avoided by aligning the information stored on an even word-address.

The FXSAVE/XSAVE and FXRSTOR/XRSTOR instructions save and restore a 512-byte data structure, the first byte of which must be aligned on a 16-byte boundary. If the alignment-check exception (#AC) is enabled when executing these instructions (and CPL is 3), a misaligned memory operand can cause either an alignment-check exception or a general-protection exception (#GP) depending on the processor implementation (see “FXSAVE-Save x87 FPU, MMX, SSE, and SSE2 State” and “FXRSTOR-Restore x87 FPU, MMX, SSE, and SSE2 State” in Chapter 3.
The MOVDQU, MOVUPS, and MOVUPD instructions perform 128-bit unaligned loads or stores. The LDDQU instructions loads 128-bit unaligned data. They do not generate general-protection exceptions (#GP) when operands are not aligned on a 16-byte boundary. If alignment checking is enabled, alignment-check exceptions (#AC) may or may not be generated depending on processor implementation when data addresses are not aligned on an 8-byte boundary.

FSAVE and FRSTOR instructions can generate unaligned references, which can cause alignment-check faults. These instructions are rarely needed by application programs.

**Exception Error Code**

Yes. The error code is null; all bits are clear except possibly bit 0 — EXT; see Section 6.13. EXT is set if the #AC is recognized during delivery of an event other than a software interrupt (see “INT n/INTO/INT3/INT1—Call to Interrupt Procedure” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

**Saved Instruction Pointer**

The saved contents of CS and EIP registers point to the instruction that generated the exception.

**Program State Change**

A program-state change does not accompany an alignment-check fault, because the instruction is not executed.
Interrupt 18—Machine-Check Exception (#MC)

Exception Class  Abort.

Description
Indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The machine-check exception is model-specific, available on the Pentium and later generations of processors. The implementation of the machine-check exception is different between different processor families, and these implementations may not be compatible with future Intel 64 or IA-32 processors. (Use the CPUID instruction to determine whether this feature is present.)

Bus errors detected by external agents are signaled to the processor on dedicated pins: the BINIT# and MCERR# pins on the Pentium 4, Intel Xeon, and P6 family processors and the BUSCHK# pin on the Pentium processor. When one of these pins is enabled, asserting the pin causes error information to be loaded into machine-check registers and a machine-check exception is generated.

The machine-check exception and machine-check architecture are discussed in detail in Chapter 16, “Machine-Check Architecture.” Also, see the data books for the individual processors for processor-specific hardware information.

Exception Error Code
None. Error information is provided by machine-check MSRs.

Saved Instruction Pointer
For the Pentium 4 and Intel Xeon processors, the saved contents of extended machine-check state registers are directly associated with the error that caused the machine-check exception to be generated (see Section 16.3.1.2, “IA32_MCG_STATUS MSR,” and Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs”).

For the P6 family processors, if the EIPV flag in the MCG_STATUS MSR is set, the saved contents of CS and EIP registers are directly associated with the error that caused the machine-check exception to be generated; if the flag is clear, the saved instruction pointer may not be associated with the error (see Section 16.3.1.2, “IA32_MCG_STATUS MSR”).

For the Pentium processor, contents of the CS and EIP registers may not be associated with the error.

Program State Change
The machine-check mechanism is enabled by setting the MCE flag in control register CR4.

For the Pentium 4, Intel Xeon, P6 family, and Pentium processors, a program-state change always accompanies a machine-check exception, and an abort class exception is generated. For abort exceptions, information about the exception can be collected from the machine-check MSRs, but the program cannot generally be restarted.

If the machine-check mechanism is not enabled (the MCE flag in control register CR4 is clear), a machine-check exception causes the processor to enter the shutdown state.
Interrupt 19—SIMD Floating-Point Exception (#XM)

Exception Class Fault.

Description
Indicates the processor has detected an SSE/SSE2/SSE3 SIMD floating-point exception. The appropriate status flag in the MXCSR register must be set and the particular exception unmasked for this interrupt to be generated. There are six classes of numeric exception conditions that can occur while executing an SSE/ SSE2/SSE3 SIMD floating-point instruction:
- Invalid operation (#I)
- Divide-by-zero (#Z)
- Denormal operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (Precision) (#P)

The invalid operation, divide-by-zero, and denormal-operand exceptions are pre-computation exceptions; that is, they are detected before any arithmetic operation occurs. The numeric underflow, numeric overflow, and inexact result exceptions are post-computational exceptions.

See “SIMD Floating-Point Exceptions” in Chapter 11 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional information about the SIMD floating-point exception classes.

When a SIMD floating-point exception occurs, the processor does either of the following things:
- It handles the exception automatically by producing the most reasonable result and allowing program execution to continue undisturbed. This is the response to masked exceptions.
- It generates a SIMD floating-point exception, which in turn invokes a software exception handler. This is the response to unmasked exceptions.

Each of the six SIMD floating-point exception conditions has a corresponding flag bit and mask bit in the MXCSR register. If an exception is masked (the corresponding mask bit in the MXCSR register is set), the processor takes an appropriate automatic default action and continues with the computation. If the exception is unmasked (the corresponding mask bit is clear) and the operating system supports SIMD floating-point exceptions (the OSXMMEXCPT flag in control register CR4 is set), a software exception handler is invoked through a SIMD floating-point exception. If the exception is unmasked and the OSXMMEXCPT bit is clear (indicating that the operating system does not support unmasked SIMD floating-point exceptions), an invalid opcode exception (#UD) is signaled instead of a SIMD floating-point exception.

Note that because SIMD floating-point exceptions are precise and occur immediately, the situation does not arise where an x87 FPU instruction, a WAIT/FWAIT instruction, or another SSE/SSE2/SSE3 instruction will catch a pending unmasked SIMD floating-point exception.

In situations where a SIMD floating-point exception occurred while the SIMD floating-point exceptions were masked (causing the corresponding exception flag to be set) and the SIMD floating-point exception was subsequently unmasked, then no exception is generated when the exception is unmasked.

When SSE/SSE2/SSE3 SIMD floating-point instructions operate on packed operands (made up of two or four sub-operands), multiple SIMD floating-point exception conditions may be detected. If no more than one exception condition is detected for one or more sets of sub-operands, the exception flags are set for each exception condition detected. For example, an invalid exception detected for one sub-operand will not prevent the reporting of a divide-by-zero exception for another sub-operand. However, when two or more exceptions conditions are generated for one sub-operand, only one exception condition is reported, according to the precedences shown in Table 6-8. This exception precedence sometimes results in the higher priority exception condition being reported and the lower priority exception conditions being ignored.
### Table 6-8. SIMD Floating-Point Exceptions Priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Highest)</td>
<td>Invalid operation exception due to SNaN operand (or any NaN operand for maximum, minimum, or certain compare and convert operations).</td>
</tr>
<tr>
<td>2</td>
<td>QNaN operand(^1).</td>
</tr>
<tr>
<td>3</td>
<td>Any other invalid operation exception not mentioned above or a divide-by-zero exception(^2).</td>
</tr>
<tr>
<td>4</td>
<td>Denormal operand exception(^2).</td>
</tr>
<tr>
<td>5</td>
<td>Numeric overflow and underflow exceptions possibly in conjunction with the inexact result exception(^2).</td>
</tr>
<tr>
<td>6 (Lowest)</td>
<td>Inexact result exception.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Though a QNaN this is not an exception, the handling of a QNaN operand has precedence over lower priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a divide-by-zero- exception.
2. If masked, then instruction execution continues, and a lower priority exception can occur as well.

### Exception Error Code

None.

### Saved Instruction Pointer

The saved contents of CS and EIP registers point to the SSE/SSE2/SSE3 instruction that was executed when the SIMD floating-point exception was generated. This is the faulting instruction in which the error condition was detected.

### Program State Change

A program-state change does not accompany a SIMD floating-point exception because the handling of the exception is immediate unless the particular exception is masked. The available state information is often sufficient to allow recovery from the error and re-execution of the faulting instruction if needed.
Interrupt 20—Virtualization Exception (#VE)

Exception Class Fault.

Description
Indicates that the processor detected an EPT violation in VMX non-root operation. Not all EPT violations cause virtualization exceptions. See Section 26.5.7.2 for details.
The exception handler can recover from EPT violations and restart the program or task without any loss of program continuity. In some cases, however, the problem that caused the EPT violation may be uncorrectable.

Exception Error Code
None.

Saved Instruction Pointer
The saved contents of CS and EIP registers generally point to the instruction that generated the exception.

Program State Change
A program-state change does not normally accompany a virtualization exception, because the instruction that causes the exception to be generated is not executed. After the virtualization exception handler has corrected the violation (for example, by executing the EPTP-switching VM function), execution of the program or task can be resumed.

Additional Exception-Handling Information
The processor saves information about virtualization exceptions in the virtualization-exception information area. See Section 26.5.7.2 for details.
Interrupt 21—Control Protection Exception (#CP)

Exception Class Fault.

Description
Indicates a control flow transfer attempt violated the control flow enforcement technology constraints.

Exception Error Code
Yes (special format). The processor provides the control protection exception handler with following information through the error code on the stack.

- Bit 14:0 - CPEC
  - 1 - NEAR-RET: Indicates the #CP was caused by a near RET instruction.
  - 2 - FAR-RET/IRET: Indicates the #CP was caused by a FAR RET or IRET instruction.
  - 3 - ENDBRANCH: indicates the #CP was due to missing ENDBRANCH at target of an indirect call or jump instruction.
  - 4 - RSTORSSP: Indicates the #CP was caused by a shadow-stack-restore token check failure in the RSTORSSP instruction.
  - 5 - SETSSBSY: Indicates #CP was caused by a supervisor shadow stack token check failure in the SETSSBSY instruction.

- Bit 15 (ENCL) of the error code, if set to 1, indicates the #CP occurred during enclave execution.

Saved Instruction Pointer
The saved contents of the CS and EIP registers generally point to the instruction that generated the exception.

Program State Change
A program-state change does not normally accompany a control protection exception, because the instruction that causes the exception to be generated is not executed.

When a control protection exception is generated during a task switch, the program-state may change as follows. During a task switch, a control protection exception can occur during any of following operations:

- If task switch is initiated by IRET, CS and LIP stored on old task shadow stack do not match CS and LIP of new task (where LIP is the linear address of the return address).
- If task switch is initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is < 3), OR the SSP from IA32_PL3_SSP (if new task CPL = 3) is not aligned to 4 bytes or is a value beyond 4GB.

Figure 6-12. Exception Error Code Information
In these cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits control protection faults to occur during task-switches, the control protection fault handler should be called through a task gate.
Interrupts 32 to 255—User Defined Interrupts

Exception Class  Not applicable.

Description
Indicates that the processor did one of the following things:
• Executed an INT \( n \) instruction where the instruction operand is one of the vector numbers from 32 through 255.
• Responded to an interrupt request at the INTR pin or from the local APIC when the interrupt vector number associated with the request is from 32 through 255.

Exception Error Code
Not applicable.

Saved Instruction Pointer
The saved contents of CS and EIP registers point to the instruction that follows the INT \( n \) instruction or instruction following the instruction on which the INTR signal occurred.

Program State Change
A program-state change does not accompany interrupts generated by the INT \( n \) instruction or the INTR signal. The INT \( n \) instruction generates the interrupt within the instruction stream. When the processor receives an INTR signal, it commits all state changes for all previous instructions before it responds to the interrupt; so, program execution can resume upon returning from the interrupt handler.
7.1 INTRODUCTION

This chapter provides details of an architectural feature called user interrupts. This feature defines user interrupts as new events in the architecture. User interrupts are delivered to software operating in 64-bit mode with CPL = 3 without any change to segmentation state. An individual user interrupt is identified by a 6-bit user-interrupt vector, which is pushed on the stack as part of user-interrupt delivery. The UIRET (user-interrupt return) instruction reverses user-interrupt delivery.

System software configures the user-interrupt architecture with MSRs. An operating system (OS) may update the content of some of these MSRs when switching between OS-managed threads. One of these MSRs references a data structure called the user posted-interrupt descriptor (UPID). User interrupts for an OS-managed thread can be posted in the UPID associated with that thread. Such user interrupts will be delivered after receipt of an ordinary interrupt (identified in the UPID) called a user-interrupt notification.1 System software can define operations to post user interrupts and to send user-interrupt notifications. In addition, the user-interrupt feature defines the SENDUIPI instruction, by which application software can send interprocessor user interrupts (user IPIs). An execution of SENDUIPI posts a user interrupt in a UPID and may send a user-interrupt notification.

(Platforms may include mechanisms to process external interrupts as either ordinary interrupts or user interrupts. Those processed as user interrupts would be posted in UPIDs and may result in user-interrupt notifications. Specifics of such mechanisms are outside of the scope of this manual.)

Section 7.2 explains how a processor enumerates support for user interrupts and how they are enabled by system software. Section 7.3 identifies the new processor state defined for user interrupts. Section 7.4 explains how a processor identifies and delivers user interrupts. Section 7.5 describes how a processor identifies and processes user-interrupt notifications. Section 7.6 enumerates new instructions that support management of user interrupts. Section 7.7 defines new support for user inter-processor interrupts (user IPIs).

7.2 ENUMERATION AND ENABLING

Software enables user interrupts by setting bit 25 (UINTR) in control register CR4. Setting CR4.UINTR enables user-interrupt delivery (Section 7.4.2), user-interrupt notification identification (Section 7.5.1), and the user-interrupt instructions (Section 7.6). It does not affect the accessibility of the user-interrupt MSRs (Section 7.3) by RDMSR, WRMSR or the XSAVE feature set.

Processor support for user interrupts is enumerated by CPUID.(EAX=7,ECX=0):EDX[5]. If this bit is set, software can set CR4.UINTR to 1 and can access the user-interrupt MSRs using RDMSR and WRMSR (see Section 7.3).

The user-interrupt feature is XSAVE-managed (see Section 13.5). This implies that aspects of the feature are enumerated as part of enumeration of the XSAVE feature set. See Section 13.5.11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for details.

7.3 USER-INTERRUPT STATE AND USER-INTERRUPT MSRS

The user-interrupt architecture defines the following new state. Some of this state can be accessed via the RDMSR and WRMSR instructions (through new user-interrupt MSRs detailed in Section 7.3.2) and some can be accessed using instructions described in Section 7.6.

1. For clarity, this chapter uses the term ordinary interrupts to refer to those events in the existing interrupt architecture, which are typically delivered to system software operating with CPL = 0.
### 7.3.1 User-Interrupt State

The following are the elements of the user-interrupt state (listed here independent of how they are accessed):

- **UIRR: user-interrupt request register.**
  This value includes one bit for each of the 64 user-interrupt vectors. If UIRR[i] = 1, a user interrupt with vector i is requesting service. The notation **UIRRV** is used to refer to the position of the most significant bit set in UIRR; if UIRR = 0, UIRRV = 0.

- **UIF: user-interrupt flag.**
  If UIF = 0, user-interrupt delivery is blocked; if UIF = 1, user interrupts may be delivered. User-interrupt delivery clears UIF, and the new UIRET instruction sets it. Section 7.6 defines other instructions for accessing UIF.

- **UIHANDLER: user-interrupt handler.**
  This is the linear address of the user-interrupt handler. User-interrupt delivery loads this address into RIP.

- **UISTACKADJUST: user-interrupt stack adjustment.**
  This value controls adjustment to the stack pointer (RSP) prior to user-interrupt delivery. It can be configured to load RSP with an alternate stack pointer or configured to prevent user-interrupt delivery from overwriting data above the current stack top.

  The value UISTACKADJUST must be canonical. If bit 0 is 1, user-interrupt delivery loads RSP with UISTACKADJUST; otherwise, it subtracts UISTACKADJUST from RSP. Either way, user-interrupt delivery then aligns RSP to a 16-byte boundary. See Section 7.4.2 for details.

- **UINV: user-interrupt notification vector.**
  This is the vector of the ordinary interrupts that are treated as user-interrupt notifications (Section 7.5.1). When the logical processor receives user-interrupt notification, it processes the user interrupts in the user posted-interrupt descriptor (UPID) referenced by UPIDADDR (see below and Section 7.5.2).

- **UPIDADDR: user posted-interrupt descriptor address.**
  This is the linear address of the UPID that the logical processor consults upon receiving an ordinary interrupt with vector UINV.

- **UITTADDR: user-interrupt target table address.**
  This is the linear address of user-interrupt target table (UITT), which the logical processor consults when software executes the SENDUIPI instruction (see Section 7.7).

- **UITTSZ: user-interrupt target table size.**
  This value is the highest index of a valid entry in the UITT (see Section 7.7).

### 7.3.2 User-Interrupt MSRs

Some of the state elements identified in Section 7.3.1 can be accessed as user-interrupt MSRs using the RDMSR and WRMSR instructions:

- **IA32_UINTR_RR MSR (MSR address 985H).** This MSR is an interface to UIRR (64 bits).
  Following a WRMSR to this MSR, the logical processor recognizes a pending user interrupt if and only if some bit is set in the MSR.

- **IA32_UINTR_HANDLER MSR (MSR address 986H).** This MSR is an interface to the UIHANDLER address. This is a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a general-protection fault (#GP) if its source operand does not meet this requirement.

- **IA32_UINTR_STACKADJUST MSR (MSR address 987H).** This MSR is an interface to the UISTACKADJUST value. This value includes a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a #GP if its source operand does not meet this requirement.

---

2. CPUID.80000008H:EAX[15:8] enumerates the maximum linear-address width supported by the processor.
Bit 0 of this MSR corresponds to UISTACKADJUST[0], which controls how user-interrupt delivery updates the stack pointer. WRMSR may set it to either 0 or 1.

- IA32_UINTR_MISC MSR (MSR address 988H). This MSR is an interface to the UITT_SZ and UINV values. The MSR has the following format:
  - Bits 31:0 are UITT_SZ.
  - Bits 39:32 are UINV.
  - Bits 63:40 are reserved. WRMSR causes a #GP if it would set any of those bits (if EDX[31:8] ≠ 000000H).

Because this MSR will share an 8-byte portion of the XSAVE area with UIF (see Section 13.5.11 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1), bit 63 of the MSR will never be used and will always be reserved.

- IA32_UINTR_PD MSR (MSR address 989H). This MSR is an interface to the UPIDADDR address. This is a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a #GP if its source operand does not meet this requirement.
  - Bits 5:0 of this MSR are reserved. WRMSR causes a #GP if it would set any of those bits (if EAX[5:0] ≠ 000000b).

- IA32_UINTR_TT MSR (MSR address 98AH). This MSR is an interface to the UITTADDR address (in addition, bit 0 enables SENDUIPI).
  - Bit 63:4 of this MSR holds the current value of UITTADDR. This a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a #GP if its source operand does not meet this requirement.
  - Bits 3:1 of this MSR are reserved. WRMSR causes a #GP if it would set any of those bits (if EAX[3:1] ≠ 000b). 
  - Bit 0 of this MSR determines whether the SENDUIPI instruction is enabled. WRMSR may set it to either 0 or 1.

### 7.4 EVALUATION AND DELIVERY OF USER INTERRUPTS

A processor determines whether there is a user interrupt to deliver based on UIRR. Section 7.4.1 describes this recognition of pending user interrupts. Once a logical processor has recognized a pending user interrupt, it will deliver it on a subsequent instruction boundary by causing a control-flow change asynchronous to software execution. Section 7.4.2 details this process of user-interrupt delivery.

#### 7.4.1 User-Interrupt Recognition

There is a user interrupt pending whenever UIRR ≠ 0.

Any instruction or operation that modifies UIRR updates the logical processor’s recognition of a pending user interrupt. The following instructions and operations may do this:

- WRMSR to the IA32_UINTR_RR MSR (Section 7.3).
- XORSTORS of the user-interrupt state component.
- User-interrupt delivery (Section 7.4.2).
- User-interrupt notification processing (Section 7.5.2).
- VMX transitions that load the IA32_UINTR_RR MSR.

Each of these instructions or operations results in recognition of a pending user interrupt if it completes with UIRR ≠ 0; if it completes with UIRR = 0, no pending user interrupt is recognized.

Once recognized, a pending user interrupt may be delivered to software; see Section 7.4.2.
7.4.2 User-Interrupt Delivery

If CR4.UINTR = 1 and a user interrupt has been recognized (see Section 7.4.1), it will be delivered at an instruction boundary when the following conditions all hold: (1) UIF = 1; (2) there is no blocking by MOV SS or by POP SS; (3) CPL = 3; (4) IA32_EFER.LMA = CS.L = 1 (the logical processor is in 64-bit mode); and (5) software is not executing inside an enclave.

User-interrupt delivery has priority just below that of ordinary interrupts. It wakes a logical processor from the states entered using the TPAUSE and UMWAIT instructions; it does not wake a logical processor in the shutdown state or in the wait-for-SIPI state.

User-interrupt delivery does not change CPL (it occurs entirely with CPL = 3). The following pseudocode details the behavior of user-interrupt delivery:

IF UIHANDLER is not canonical in current paging mode
   THEN #GP(0);
FI;
holdRSP := RSP;
IF UISTACKADJUST[0] = 1
   THEN RSP := UISTACKADJUST;
   ELSE RSP := RSP – UISTACKADJUST;
FI;
RSP := RSP & ~FH;  // force the stack to be 16-byte aligned
Push holdRSP;
Push RFLAGS;
Push RIP;
Push UIRRV; // 64-bit push; upper 58 bits pushed as 0
IF shadow stack is enabled for CPL = 3
   THEN ShadowStackPush RIP;
FI;
IF end-branch is enabled for CPL = 3
   THEN IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
FI;
UIRR[Vector] := 0;
IF UIRR = 0
   THEN cease recognition of any pending user interrupt;
FI;
UIF := 0;
RFLAGS.TF := 0;
RFLAGS.RF := 0;
RIP := UIHANDLER;

If UISTACKADJUST[0] = 0, user-interrupt delivery decrements RSP by UISTACKADJUST; otherwise, it loads RSP with UISTACKADJUST. In either case, user-interrupt delivery aligns RSP to a 16-byte boundary by clearing RSP[3:0].

User-interrupt delivery that occurs during transactional execution causes transactional execution to abort and a transition to a non-transactional execution. The transactional abort loads EAX as it would had it been due to an ordinary interrupt. User-interrupt delivery occurs after the transactional abort.

3. Execution of the STI instruction does not block delivery of user interrupts for one instruction as it does ordinary interrupts. If a user interrupt is delivered immediately following execution of a STI instruction, ordinary interrupts are not blocked after delivery of the user interrupt.

4. User-interrupt delivery occurs only if CPL = 3. Since the HLT and MWAIT instructions can be executed only if CPL = 0, a user interrupt can never be delivered when a logical processor is an activity state that was entered using one of those instructions.
The stack accesses performed by user-interrupt delivery may incur faults (page faults, or stack faults due to canonicality violations). Before such a fault is delivered, RSP is restored to its original value (memory locations above the top of the stack may have been written). If such a fault produces an error code that uses the EXT bit, that bit will be cleared to 0.

If a fault occurs during user-interrupt delivery, UIRR is not updated and UIF is not cleared and, as a result, the logical processor continues to recognize that a user interrupt is pending, and user-interrupt delivery will normally recur after the fault is handled.

If the shadow-stack feature of control-flow enforcement technology (CET) is enabled for CPL = 3, user-interrupt delivery pushes the return instruction pointer on the shadow stack. If indirect-branch-tracking feature of CET is enabled, user-interrupt delivery transitions the indirect branch tracker to the WAIT_FOR_ENDBRANCH state; an ENDBR64 instruction is expected as first instruction of the user-interrupt handler.

User-interrupt delivery can be tracked by Architectural Last Branch Records (LBRs), Intel® Processor Trace (Intel® PT), and Performance Monitoring. For both Intel PT and LBRs, user-interrupt delivery is recorded in precisely the same manner as ordinary interrupt delivery. Hence for LBRs, user interrupts fall into the OTHER_BRANCH category, which implies that IA32_LBR_CTL.OTHER_BRANCH[bit 22] must be set to record user-interrupt delivery, and that the IA32_LBR_x_INFO.BR_TYPE field will indicate OTHER_BRANCH for any recorded user interrupt. For Intel PT, control flow tracing must be enabled by setting IA32_RTIT_CTL.BranchEn[bit 13].

User-interrupt delivery will also increment performance counters for which counting BR_INST_RETIRED.FAR_BRANCH is enabled. Some implementations may have dedicated events for counting user-interrupt delivery; see processor-specific event lists at https://download.01.org/perfmon/index/.

7.5 USER-INTERRUPT NOTIFICATION IDENTIFICATION AND PROCESSING

User-interrupt posting is the process by which a platform agent (or software operating on a CPU) records user interrupts in a user posted-interrupt descriptor (UPID) in memory. The platform agent (or software) may send an ordinary interrupt (called a user-interrupt notification) to the logical processor on which the target of the user interrupt is operating.

Table 7-1 gives the format of a UPID.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Outstanding notification</td>
<td>If this bit is set, there is a notification outstanding for one or more user interrupts in PIR.</td>
</tr>
<tr>
<td>1</td>
<td>Suppress notification</td>
<td>If this bit is set, agents (including SENDUIPI) should not send notifications when posting user interrupts in this descriptor.</td>
</tr>
<tr>
<td>15:2</td>
<td>Reserved</td>
<td>User-interrupt notification processing ignores these bits; must be zero for SENDUIPI.</td>
</tr>
<tr>
<td>23:16</td>
<td>Notification vector</td>
<td>Used by agents sending user-interrupt notifications (including SENDUIPI).</td>
</tr>
<tr>
<td>31:24</td>
<td>Reserved</td>
<td>User-interrupt notification processing ignores these bits; must be zero for SENDUIPI.</td>
</tr>
<tr>
<td>63:32</td>
<td>Notification destination</td>
<td>Target physical APIC ID – used by SENDUIPI. In xAPIC mode, bits 47:40 are the 8-bit APIC ID. In x2APIC mode, the entire field forms the 32-bit APIC ID.</td>
</tr>
<tr>
<td>127:64</td>
<td>Posted-interrupt requests (PIR)</td>
<td>One bit for each user-interrupt vector. There is a user-interrupt request for a vector if the corresponding bit is 1.</td>
</tr>
</tbody>
</table>

The notation PIR (posted-interrupt requests) refers to the 64 posted-interrupt requests in a UPID.

If an ordinary interrupt arrives while CR4.UINTR = IA32_EFER.LMA = 1, the logical processor determines whether the interrupt is a user-interrupt notification. This process is called user-interrupt notification identification and is described in Section 7.5.1.

Once a logical processor has identified a user-interrupt notification, it copies user interrupts in the UPID’s PIR into UIRR. This process is called user-interrupt notification processing and is described in Section 7.5.2.
A logical processor is not interruptible during either user-interrupt notification identification or user-interrupt notification processing or between those operations (when they occur in succession).

### 7.5.1 User-Interrupt Notification Identification

If CR4.UINTR = IA32_EFER.LMA = 1, a logical processor performs user-interrupt notification identification when it receives an ordinary interrupt. The following algorithm describes the response by the processor to an ordinary maskable interrupt when CR4.UINTR = IA32_EFER.LMA = 1:

1. The local APIC is acknowledged; this provides the processor core with an interrupt vector, V.
2. If V = UINV, the logical processor continues to the next step. Otherwise, an interrupt with vector V is delivered normally through the IDT; the remainder of this algorithm does not apply and user-interrupt notification processing does not occur.
3. The processor writes zero to the EOI register in the local APIC; this dismisses the interrupt with vector V = UINV from the local APIC.

User-interrupt notification identification involves acknowledgment of the local APIC and thus occurs only when ordinary interrupts are not masked.

If user-interrupt notification identification completes step #3, the logical processor then performs user-interrupt notification processing as described in Section 7.5.2.

An ordinary interrupt that occurs during transactional execution causes the transactional execution to abort and transition to a non-transactional execution. This occurs before user-interrupt notification identification.

An ordinary interrupt that occurs while software is executing inside an enclave causes an asynchronous enclave exit (AEX). This AEX occurs before user-interrupt notification identification.

### 7.5.2 User-Interrupt Notification Processing

Once a logical processor has identified a user-interrupt notification, it performs user-interrupt notification processing using the UPID at the linear address in the IA32_UINTR_PD MSR.

The following algorithm describes user-interrupt notification processing:

1. The logical processor clears the outstanding-notification bit (bit 0) in the UPID. This is done atomically so as to leave the remainder of the descriptor unmodified.
2. The logical processor reads PIR (bits 127:64 of the UPID) into a temporary register and writes all zeros to PIR. This is done atomically so as to ensure that each bit cleared in PIR is set in the temporary register.
3. If any bit is set in the temporary register, the logical processor sets in UIRR each bit corresponding to a bit set in the temporary register (e.g., with a logical OR) and recognizes a pending user interrupt (if it has not already done so).

The logical processor performs the steps above in an uninterruptible manner. Steps #1 and #2 may be combined into a single atomic step. If step #3 leads to recognition of a user interrupt, the processor may deliver that user interrupt on the following instruction boundary (see Section 7.4.2).

Although user-interrupt notification processing may occur at any privilege level, all of the memory accesses in steps #1 and #2 are performed with supervisor privilege.

Step #1 and step #2 each access the UPID using a linear address and may therefore incur faults (page faults, or general-protection faults due to canonicality violations). If such a fault produces an error code that uses the EXT bit, that bit will be set to 1.

If a fault occurs during user-interrupt notification processing, updates to architectural state performed by the earlier user-interrupt notification identification (Section 7.5.1) remain committed and are not undone; if such a fault occurs at step #2 (if it is not performed atomically with step #1), any update to architectural state performed by step #1 also remains committed. System software is advised to prevent such faults (e.g., by ensuring that no

5. If the interrupt arrives between iterations of a REP-prefixed string instruction, the processor first updates state as follows: RIP is loaded to reference the string instruction; RCX, RSI, and RDI are updated as appropriate to reflect the iterations completed; and RFLAGS.RF is set to 1.
If the user-interrupt notification identification that precedes user-interrupt notification processing occurred due to an ordinary interrupt that arrived while the logical processor was in the HLT state, the logical processor returns to the HLT state following user-interrupt notification processing.

7.6 USER-INTERRUPT INSTRUCTIONS

The user-interrupt feature defines instructions for control-flow transfer and access to new state. UIRET is an instruction to effect a return from a user-interrupt handler. CLUI, STUI, and TESTUI allow software to access UIF. SENDUIPI sends a user IPI. See Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D for details on the instructions’ operation.

The following items provide high-level overviews of the instructions:

- UIRET pops from the stack the state saved by user-interrupt delivery (see Section 7.4.2) and loads those values into the corresponding registers (software should pop the user-interrupt vector from the stack before executing UIRET). Because RIP is one of those registers, UIRET effect a return to the that point from which the user interrupt was delivered.
- CLUI clears UIF.
- STUI sets UIF.
- TESTUI copies UIF to RFLAGS.CF.
- SENDUIPI is discussed in Section 7.7.

7.7 USER IPIS

The SENDUIPI instruction sends a user interprocessor interrupt (IPI). The instruction uses a data structure called the user-interrupt target table (UITT). This table is located at the linear address UITTADDR and it comprises UITTSZ+1 16-byte entries (the values UITTADDR and UITTSZ are defined in Section 7.3.1). SENDUIPI uses the UITT entry (UITTE) indexed by the instruction’s register operand. Each UITTE has the following format:

- Bit 0: V, a valid bit.
- Bits 7:1 are reserved and must be 0.
- Bits 15:8: UV, the user-interrupt vector (in the range 0–63, so bits 15:14 must be 0).
- Bits 63:16 are reserved.
- Bits 127:64: UPIDADDR, the linear address of a UPID (64-byte aligned, so bits 69:64 must be 0).

SENDUIPI sends a user interrupt by posting a user interrupt with vector V in the UPID referenced by UPIDADDR and then sending, as an ordinary IPI, any notification interrupt specified in that UPID. Details appear in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.
USER INTERRUPTS
This chapter describes the IA-32 architecture's task management facilities. These facilities are only available when the processor is running in protected mode.

This chapter focuses on 32-bit tasks and the 32-bit TSS structure. For information on 16-bit tasks and the 16-bit TSS structure, see Section 8.6, “16-Bit Task-State Segment (TSS).” For information specific to task management in 64-bit mode, see Section 8.7, “Task Management in 64-bit Mode.”

### 8.1 TASK MANAGEMENT OVERVIEW

A task is a unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility.

The IA-32 architecture provides a mechanism for saving the state of a task, for dispatching tasks for execution, and for switching from one task to another. When operating in protected mode, all processor execution takes place from within a task. Even simple systems must define at least one task. More complex systems can use the processor's task management facilities to support multitasking applications.

#### 8.1.1 Task Structure

A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment, and one or more data segments (see Figure 8-1). If an operating system or executive uses the processor’s privilege-level protection mechanism, the task execution space also provides a separate stack for each privilege level.

The TSS specifies the segments that make up the task execution space and provides a storage place for task state information. In multitasking systems, the TSS also provides a mechanism for linking tasks.

A task is identified by the segment selector for its TSS. When a task is loaded into the processor for execution, the segment selector, base address, limit, and segment descriptor attributes for the TSS are loaded into the task register (see Section 2.4.4, ”Task Register (TR”).

If paging is implemented for the task, the base address of the page directory used by the task is loaded into control register CR3.
8.1.2 Task State

The following items define the state of the currently executing task:

- The task’s current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).
- The state of the general-purpose registers.
- The state of the EFLAGS register.
- The state of the EIP register.
- The state of control register CR3.
- The state of the task register.
- The state of the LDTR register.
- The I/O map base address and I/O map (contained in the TSS).
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS).
- Link to previously executed task (contained in the TSS).
- The state of the shadow stack pointer (SSP).

Prior to dispatching a task, all of these items are contained in the task’s TSS, except the state of the task register. Also, the complete contents of the LDTR register are not contained in the TSS, only the segment selector for the LDT.

8.1.3 Executing a Task

Software or the processor can dispatch a task for execution in one of the following ways:

- A explicit call to a task with the CALL instruction.
- A explicit jump to a task with the JMP instruction.
- An implicit call (by the processor) to an interrupt-handler task.
- An implicit call to an exception-handler task.
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods for dispatching a task identify the task to be dispatched with a segment selector that points to a task gate or the TSS for the task. When dispatching a task with a CALL or JMP instruction, the selector in the instruction may select the TSS directly or a task gate that holds the selector for the TSS. When dispatching a task
to handle an interrupt or exception, the IDT entry for the interrupt or exception must contain a task gate that holds
the selector for the interrupt- or exception-handler TSS.

When a task is dispatched for execution, a task switch occurs between the currently running task and the
dispatched task. During a task switch, the execution environment of the currently executing task (called the task’s
state or **context**) is saved in its TSS and execution of the task is suspended. The context for the dispatched task is
then loaded into the processor and execution of that task begins with the instruction pointed to by the newly loaded
EIP register. If the task has not been run since the system was last initialized, the EIP will point to the first instruc-
tion of the task’s code; otherwise, it will point to the next instruction after the last instruction that the task
executed when it was last active.

If the currently executing task (the calling task) called the task being dispatched (the called task), the TSS
segment selector for the calling task is stored in the TSS of the called task to provide a link back to the calling task.

For all IA-32 processors, tasks are not recursive. A task cannot call or jump to itself.

Interrupts and exceptions can be handled with a task switch to a handler task. Here, the processor performs a task
switch to handle the interrupt or exception and automatically switches back to the interrupted task upon returning
from the interrupt-handler task or exception-handler task. This mechanism can also handle interrupts that occur
during interrupt tasks.

As part of a task switch, the processor can also switch to another LDT, allowing each task to have a different logical-
to-physical address mapping for LDT-based segments. The page-directory base register (CR3) also is reloaded on a
task switch, allowing each task to have its own set of page tables. These protection facilities help isolate tasks and
prevent them from interfering with one another.

If protection mechanisms are not used, the processor provides no protection between tasks. This is true even with
operating systems that use multiple privilege levels for protection. A task running at privilege level 3 that uses the
same LDT and page tables as other privilege-level-3 tasks can access code and corrupt data and the stack of other
tasks.

Use of task management facilities for handling multitasking applications is optional. Multitasking can be handled in
software, with each software defined task executed in the context of a single IA-32 architecture task.

If shadow stack is enabled, then the SSP of the task is located at the 4 bytes at offset 104 in the 32-bit TSS and is
used by the processor to establish the SSP when a task switch occurs from a task associated with this TSS. Note
that the processor does not write the SSP of the task initiating the task switch to the TSS of that task, and instead
the SSP of the previous task is pushed onto the shadow stack of the new task.

## 8.2 TASK MANAGEMENT DATA STRUCTURES

The processor defines five data structures for handling task-related activities:

- Task-state segment (TSS).
- Task-gate descriptor.
- TSS descriptor.
- Task register.
- NT flag in the EFLAGS register.

When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the
segment selector for the TSS must be loaded into the task register (using the LTR instruction).

### 8.2.1 Task-State Segment (TSS)

The processor state information needed to restore a task is saved in a system segment called the task-state
segment (TSS). Figure 8-2 shows the format of a TSS for tasks designed for 32-bit CPUs. The fields of a TSS are
divided into two main categories: dynamic fields and static fields.

For information about 16-bit Intel 286 processor task structures, see Section 8.6, ”16-Bit Task-State Segment
(TSS).” For information about 64-bit mode task structures, see Section 8.7, ”Task Management in 64-bit Mode.”
The processor updates dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

- **General-purpose register fields** — State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.
- **Segment selector fields** — Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.
- **EFLAGS register field** — State of the EFLAGS register prior to the task switch.

![Figure 8-2. 32-Bit Task-State Segment (TSS)](image)

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDT Segment Selector</td>
<td>T 100</td>
</tr>
<tr>
<td>GS</td>
<td>98</td>
</tr>
<tr>
<td>FS</td>
<td>88</td>
</tr>
<tr>
<td>DS</td>
<td>84</td>
</tr>
<tr>
<td>SS</td>
<td>80</td>
</tr>
<tr>
<td>CS</td>
<td>76</td>
</tr>
<tr>
<td>ES</td>
<td>72</td>
</tr>
<tr>
<td>EDI</td>
<td>68</td>
</tr>
<tr>
<td>ESI</td>
<td>64</td>
</tr>
<tr>
<td>EBP</td>
<td>60</td>
</tr>
<tr>
<td>ESP</td>
<td>56</td>
</tr>
<tr>
<td>EBX</td>
<td>52</td>
</tr>
<tr>
<td>EDX</td>
<td>48</td>
</tr>
<tr>
<td>ECX</td>
<td>44</td>
</tr>
<tr>
<td>EAX</td>
<td>40</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>36</td>
</tr>
<tr>
<td>EIP</td>
<td>32</td>
</tr>
<tr>
<td>CR3 (PDBR)</td>
<td>28</td>
</tr>
<tr>
<td>SS2</td>
<td>24</td>
</tr>
<tr>
<td>ESP2</td>
<td>20</td>
</tr>
<tr>
<td>SS1</td>
<td>16</td>
</tr>
<tr>
<td>ESP1</td>
<td>12</td>
</tr>
<tr>
<td>SS0</td>
<td>8</td>
</tr>
<tr>
<td>ESP0</td>
<td>4</td>
</tr>
<tr>
<td>Previous Task Link</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Reserved bits. Set to 0.
• **EIP (instruction pointer) field** — State of the EIP register prior to the task switch.

• **Previous task link field** — Contains the segment selector for the TSS of the previous task (updated on a task switch that was initiated by a call, interrupt, or exception). This field (which is sometimes called the back link field) permits a task switch back to the previous task by using the IRET instruction.

The processor reads the static fields, but does not normally change them. These fields are set up when a task is created. The following are static fields:

• **LDT segment selector field** — Contains the segment selector for the task’s LDT.

• **CR3 control register field** — Contains the base physical address of the page directory to be used by the task. Control register CR3 is also known as the page-directory base register (PDBR).

• **Privilege level-0, -1, and -2 stack pointer fields** — These stack pointers consist of a logical address made up of the segment selector for the stack segment (SS0, SS1, and SS2) and an offset into the stack (ESP0, ESP1, and ESP2). Note that the values in these fields are static for a particular task; whereas, the SS and ESP values will change if stack switching occurs within the task.

• **T (debug trap) flag (byte 100, bit 0)** — When set, the T flag causes the processor to raise a debug exception when a task switch to this task occurs (see Section 18.3.1.5, “Task-Switch Exception Condition”).

• **I/O map base address field** — Contains a 16-bit offset from the base of the TSS to the I/O permission bitmap and interrupt redirection bitmap. When present, these maps are stored in the TSS at higher addresses. The I/O map base address points to the beginning of the I/O permission bitmap and the end of the interrupt redirection bitmap. See Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the I/O permission bitmap. See Section 21.3, “Interrupt and Exception Handling in Virtual-8086 Mode,” for a detailed description of the interrupt redirection bitmap.

• **Shadow Stack Pointer (SSP)** — Contains task’s shadow stack pointer. The shadow stack of the task should have a supervisor shadow stack token at the address pointed to by the task SSP (offset 104). This token will be verified and made busy when switching to that stack using a CALL/JMP instruction, and made free when switching out of that task using an IRET instruction.

If paging is used:

• Pages corresponding to the previous task’s TSS, the current task’s TSS, and the descriptor table entries for each all should be marked as read/write.

• Task switches are carried out faster if the pages containing these structures are present in memory before the task switch is initiated.

### 8.2.2 TSS Descriptor

The TSS, like all other segments, is defined by a segment descriptor. Figure 8-3 shows the format of a TSS descriptor. TSS descriptors may only be placed in the GDT; they cannot be placed in an LDT or the IDT.

An attempt to access a TSS using a segment selector with its TI flag set (which indicates the current LDT) causes a general-protection exception (#GP) to be generated during CALLs and JMPs; it causes an invalid TSS exception (#TS) during IRETS. A general-protection exception is also generated if an attempt is made to load a segment selector for a TSS into a segment register.

The busy flag (B) in the type field indicates whether the task is busy. A busy task is currently running or suspended. A type field with a value of 1001B indicates an inactive task; a value of 1011B indicates a busy task. Tasks are not recursive. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To ensure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.
The base, limit, and DPL fields and the granularity and present flags have functions similar to their use in data-segment descriptors (see Section 3.4.5, “Segment Descriptors”). When the G flag is 0 in a TSS descriptor for a 32-bit TSS, the limit field must have a value equal to or greater than 67H, one byte less than the minimum size of a TSS. Attempting to switch to a task whose TSS descriptor has a limit less than 67H generates an invalid-TSS exception (#TS). A larger limit is required if an I/O permission bit map is included or if the operating system stores additional data. The processor does not check for a limit greater than 67H on a task switch; however, it does check when accessing the I/O permission bit map or interrupt redirection bit map.

Any program or procedure with access to a TSS descriptor (that is, whose CPL is numerically equal to or less than the DPL of the TSS descriptor) can dispatch the task with a call or a jump.

In most systems, the DPLs of TSS descriptors are set to values less than 3, so that only privileged software can perform task switching. However, in multitasking applications, DPLs for some TSS descriptors may be set to 3 to allow task switching at the application (or user) privilege level.

### 8.2.3 TSS Descriptor in 64-bit mode

In 64-bit mode, task switching is not supported, but TSS descriptors still exist. The format of a 64-bit TSS is described in Section 8.7.

In 64-bit mode, the TSS descriptor is expanded to 16 bytes (see Figure 8-4). This expansion also applies to an LDT descriptor in 64-bit mode. Table 3-2 provides the encoding information for the segment type field.
**8.2.4 Task Register**

The task register holds the 16-bit segment selector and the entire segment descriptor (32-bit base address (64 bits in IA-32e mode), 16-bit segment limit, and descriptor attributes) for the TSS of the current task (see Figure 2-6). This information is copied from the TSS descriptor in the GDT for the current task. Figure 8-5 shows the path the processor uses to access the TSS (using the information in the task register).

The task register has a visible part (that can be read and changed by software) and an invisible part (maintained by the processor and is inaccessible by software). The segment selector in the visible portion points to a TSS descriptor in the GDT. The processor uses the invisible portion of the task register to cache the segment descriptor for the TSS. Caching these values in a register makes execution of the task more efficient. The LTR (load task register) and STR (store task register) instructions load and read the visible portion of the task register:

The LTR instruction loads a segment selector (source operand) into the task register that points to a TSS descriptor in the GDT. It then loads the invisible portion of the task register with information from the TSS descriptor. LTR is a privileged instruction that may be executed only when the CPL is 0. It’s used during system initialization to put an initial value in the task register. Afterwards, the contents of the task register are changed implicitly when a task switch occurs.

The STR (store task register) instruction stores the visible portion of the task register in a general-purpose register or memory. This instruction can be executed by code running at any privilege level in order to identify the currently running task. However, it is normally used only by operating system software. (If CR4.UMIP = 1, STR can be executed only when CPL = 0.)

On power up or reset of the processor, segment selector and base address are set to the default value of 0; the limit is set to FFFFH.

---

**Figure 8-4. Format of TSS and LDT Descriptors in 64-bit Mode**

The TSS (or LDT) Descriptor consists of several fields:

- **AVL** (Available for use by system software)
- **B** (Busy flag)
- **BASE** (Segment Base Address)
- **DPL** (Descriptor Privilege Level)
- **G** (Granularity)
- **LIMIT** (Segment Limit)
- **P** (Segment Present)
- **TYPE** (Segment Type)

The diagram shows the layout of the TSS descriptor in 64-bit mode, including base address fields and segment limit fields.
8.2.5 Task-Gate Descriptor

A task-gate descriptor provides an indirect, protected reference to a task (see Figure 8-6). It can be placed in the GDT, an LDT, or the IDT. The TSS segment selector field in a task-gate descriptor points to a TSS descriptor in the GDT. The RPL in this segment selector is not used.

The DPL of a task-gate descriptor controls access to the TSS descriptor during a task switch. When a program or procedure makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. Note that when a task gate is used, the DPL of the destination TSS descriptor is not used.
A task can be accessed either through a task-gate descriptor or a TSS descriptor. Both of these structures satisfy the following needs:

- **Need for a task to have only one busy flag** — Because the busy flag for a task is stored in the TSS descriptor, each task should have only one TSS descriptor. There may, however, be several task gates that reference the same TSS descriptor.

- **Need to provide selective access to tasks** — Task gates fill this need, because they can reside in an LDT and can have a DPL that is different from the TSS descriptor’s DPL. A program or procedure that does not have sufficient privilege to access the TSS descriptor for a task in the GDT (which usually has a DPL of 0) may be allowed access to the task through a task gate with a higher DPL. Task gates give the operating system greater latitude for limiting access to specific tasks.

- **Need for an interrupt or exception to be handled by an independent task** — Task gates may also reside in the IDT, which allows interrupts and exceptions to be handled by handler tasks. When an interrupt or exception vector points to a task gate, the processor switches to the specified task.

Figure 8-7 illustrates how a task gate in an LDT, a task gate in the GDT, and a task gate in the IDT can all point to the same task.

---

### 8.3 TASK SWITCHING

The processor transfers execution to another task in one of four cases:

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
• An interrupt or exception vector points to a task-gate descriptor in the IDT.
• The current task executes an IRET when the NT flag in the EFLAGS register is set.

JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task:

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).
2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for those identified in the next sentence), and the IRET and INT1 instructions are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT \( n \), INT3, and INTO instructions, the DPL is checked and a general-protection exception (#GP) results if it is less than the CPL.\(^1\)
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H). If the task switch was initiated by IRET and shadow stacks are enabled at the current CPL, then the SSP must be aligned to 8 bytes, else a #TS(current task TSS) fault is generated. If CR4.CET is 1, then the TSS must be a 32 bit TSS and the limit of the new task’s TSS must be greater than or equal to 107 bytes, else a #TS(new task TSS) fault is generated.
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).
5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. Saves the state of the current (old) task in the current task’s TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).
7. Loads the task register with the segment selector and descriptor for the new task’s TSS.
8. If CET is enabled, the processor performs following shadow stack actions:

   Read CS of new task from new task TSS
   Read EFLAGS of new task from new task TSS
   IF EFLAGS.VM = 1
   THEN
     new task CPL = 3;
   ELSE
     new task CPL = CS.RPL;
   FI;
   pushCsLipSsp = 0
   IF task switch was initiated by CALL instruction, exception or interrupt
   IF shadow stack enabled at current CPL
   IF new task CPL < CPL and current task CPL = 3
   THEN
     IA32_PL3_SSP = SSP (* user → supervisor *)
   ELSE
     pushCsLipSsp = 1 (* no privilege change; supv → supv; supv → user *) tempSSP = SSP

---

\(^1\) The INT1 has opcode F1; the INT \( n \) instruction with \( n=1 \) has opcode CD 01.
\[
\text{tempSsLIP} = \text{CSBASE} + \text{EIP}
\]
\[
\text{tempSsCS} = \text{CS}
\]
\[
\text{FI;}
\]
\[
\text{FI;}
\]
\[
\text{FI;}
\]
\[
\text{verifyCsLIP} = 0
\]
\[
\text{IF task switch was initiated by IRET}
\]
\[
\text{IF shadow stacks enabled at current CPL}
\]
\[
\text{IF (CPL of new Task = CPL of current Task) OR}
\]
\[
(CPL of new Task < 3 AND CPL of current Task < 3) OR
\]
\[
(CPL or new Task < 3 AND CPL of current task = 3)
\]
\[
(* \text{no privilege change or supervisor \rightarrow supervisor or user \rightarrow supervisor IRET} *)
\]
\[
\text{tempSsCS} = \text{shadow_stack_load 8 bytes from SSP+16;}
\]
\[
\text{tempSsLIP} = \text{shadow_stack_load 8 bytes from SSP+8;}
\]
\[
\text{tempSSP} = \text{shadow_stack_load 8 bytes from SSP;}
\]
\[
\text{SSP} = \text{SSP + 24;}
\]
\[
\text{verifyCsLIP} = 1
\]
\[
\text{FI;}
\]
\[
// \text{Clear busy flag on current shadow stack}
\]
\[
\text{IF ( SSP & 0x07 == 0 ) (* SSP must be aligned to 8B *)}
\]
\[
\text{THEN}
\]
\[
\text{expected_token_value} = (\text{SSP} & \sim 0x07) \mid \text{BUSY_BIT; (* busy - bit 0 - must be set*)}
\]
\[
\text{new_token_value} = \text{SSP} \quad (* \text{clear the busy bit *} *)
\]
\[
\text{shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)}
\]
\[
\text{FI;}
\]
\[
\text{SSP} = 0
\]
\[
\text{FI;}
\]
9. The TSS state is loaded into the processor. This includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment selectors. A fault during the load of this state may corrupt architectural state. (If paging is not enabled, a PDBR value is read from the new task’s TSS, but it is not loaded into CR3.)

10. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task’s TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt: the busy (B) flag is left set. (See Table 8-2.)

11. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.

12. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor will set the NT flag in the EFLAGS loaded from the new task. If initiated with an IRET instruction or JMP instruction, the NT flag will reflect the state of NT in the EFLAGS loaded from the new task (see Table 8-2).

13. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task’s TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.

14. The descriptors associated with the segment selectors are loaded and qualified. Any errors associated with this loading and qualification occur in the context of the new task and may corrupt architectural state.
15. If CET is enabled, the processor performs following shadow stack actions:

IF shadow stack enabled at current CPL OR indirect branch tracking at current CPL
THEN
    IF EFLAGS.VM = 1
        THEN #TSS(new-Task-TSS);FI;
FI;
IF shadow stack enabled at current CPL
IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception (* switch stack *)
new_SSP ← Load the 4 byte from offset 104 in the TSS
// Verify new SSP to be legal
IF new_SSP & 0x07 != 0
    THEN #TSS(New-Task-TSS); FI;
expected_token_value = SSP;                     (* busy - bit 0 - must be clear *)
new_token_value          = SSP | BUSY_BIT (* set the busy bit - bit 0*)
IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value,
                              expected_token_value) != expected_token_value
    THEN #TSS(New-Task-TSS); FI;
SSP = new_SSP
IF pushCsLipSsp = 1 (* call, int, exception from user → user or supv → supv or supv → user *)
    Push tempSsCS, tempSsLip, tempSsSSP on shadow stack using 8B pushes2
FI;
FI;
IF task switch initiated by IRET
IF verifyCsLIP = 1
    (* do 64 bit comparisons; CS zero padded to 64 bit; CSBASE+EIP zero padded to 64 bit *)
IF tempSsCS and tempSsLIP do not match CS and CSBASE+EIP
    THEN #CP(FAR-RET/IRET); FI;
FI;
IF ShadowStackEnabled(CPL)
THEN
    IF (verifyCsLIP == 0) tempSSP = IA32.PL3.SSP;
    IF tempSSP & 0x03 != 0 THEN #CP(FAR-RET/IRET) // verify aligned to 4 bytes
    IF tempSSP[63:32] != 0 THEN # CP(FAR-RET/IRET)
    SSP = tempSSP
FI;
FI;
IF EndbranchEnabled(CPL)
IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception
IF CPL = 3
THEN
    IA32.U.CET.TRACKER = WAIT_FOR_ENDBRANCH

2. If any of these pushes leads to an exception or a VM exit, the supervisor shadow-stack token remains busy.
IA32_U_CET.SUPPRESS = 0
ELSE
  IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
  IA32_S_CET.SUPPRESS = 0
FI;
FI;
FI;

16. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

NOTES

If all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 8, the processor does not complete the task switch and ensures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch.

If an unrecoverable error occurs in step 9, architectural state may be corrupted, but an attempt will be made to handle the error in the prior execution environment. If an unrecoverable error occurs after the commit point (in step 13), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task.

If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the new task. See Chapter 6, “Interrupt 10—Invalid TSS Exception (#TS),” for more information about the affect of exceptions on a task when they occur after the commit point of a task switch.

The state of the currently executing task is always saved when a successful task switch occurs. If the task is resumed, execution starts with the instruction pointed to by the saved EIP value, and the registers are restored to the values they held when the task was suspended.

When switching tasks, the privilege level of the new task does not inherit its privilege level from the suspended task. The new task begins executing at the privilege level specified in the CPL field of the CS register, which is loaded from the TSS. Because tasks are isolated by their separate address spaces and TSSs and because privilege rules control access to a TSS, software does not need to perform explicit privilege checks on a task switch.

Table 8-1 shows the exception conditions that the processor checks for when switching tasks. It also shows the exception that is generated for each check if an error is detected and the segment that the error code references. (The order of the checks in the table is the order used in the P6 family processors. The exact order is model specific and may be different for other IA-32 processors.) Exception handlers designed to handle these exceptions may be subject to recursive calls if they attempt to reload the segment selector that generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

<table>
<thead>
<tr>
<th>Condition Checked</th>
<th>Exception¹</th>
<th>Error Code Reference²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment selector for a TSS descriptor references the GDT and is within the limits of the table.</td>
<td>#GP</td>
<td>New Task’s TSS</td>
</tr>
<tr>
<td>P bit is set in TSS descriptor.</td>
<td>#TS (for IRET)</td>
<td>New Task’s TSS</td>
</tr>
<tr>
<td>TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).</td>
<td>#NP</td>
<td>Task’s back-link TSS</td>
</tr>
<tr>
<td>TSS descriptor is not busy (for task switch initiated by an IRET instruction).</td>
<td>#GP (for JMP, CALL, INT)</td>
<td>New Task’s TSS</td>
</tr>
<tr>
<td>TSS segment limit greater than or equal to 108 (for 32-bit TSS) or 44 (for 16-bit TSS).</td>
<td>#TS (for IRET)</td>
<td>New Task’s TSS</td>
</tr>
<tr>
<td></td>
<td>#TS</td>
<td>New Task’s TSS</td>
</tr>
</tbody>
</table>
The TS (task switched) flag in the control register CR0 is set every time a task switch occurs. System software uses the TS flag to coordinate the actions of floating-point unit when generating floating-point exceptions with the rest of the processor. The TS flag indicates that the context of the floating-point unit may be different from that of the current task. See Section 2.5, “Control Registers,” for a detailed description of the function and use of the TS flag.

### Table 8-1. Exception Conditions Checked During a Task Switch (Contd.)

<table>
<thead>
<tr>
<th>Condition Checked</th>
<th>Exception</th>
<th>Error Code Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSS segment limit greater than or equal to 108 (for 32-bit TSS) if CR4.CET = 1.3</td>
<td>#TS</td>
<td>New Task's TSS</td>
</tr>
<tr>
<td>If shadow stack enabled and SSP not aligned to 8 bytes (for task switch initiated by an IRET instruction).3</td>
<td>#TS</td>
<td>Current Task's TSS</td>
</tr>
<tr>
<td>Registers are loaded from the values in the TSS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDT segment selector of new task is valid 4.</td>
<td>#TS</td>
<td>New Task's LDT</td>
</tr>
<tr>
<td>If code segment is non-conforming, its DPL should equal its RPL.</td>
<td>#TS</td>
<td>New Code Segment</td>
</tr>
<tr>
<td>If code segment is conforming, its DPL should be less than or equal to its RPL.</td>
<td>#TS</td>
<td>New Code Segment</td>
</tr>
<tr>
<td>SS segment selector is valid 2.</td>
<td>#TS</td>
<td>New Stack Segment</td>
</tr>
<tr>
<td>P bit is set in stack segment descriptor.</td>
<td>#SS</td>
<td>New Stack Segment</td>
</tr>
<tr>
<td>Stack segment DPL should equal CPL.</td>
<td>#TS</td>
<td>New stack segment</td>
</tr>
<tr>
<td>P bit is set in new task's LDT descriptor.</td>
<td>#TS</td>
<td>New Task's LDT</td>
</tr>
<tr>
<td>CS segment selector is valid 4.</td>
<td>#TS</td>
<td>New Code Segment</td>
</tr>
<tr>
<td>P bit is set in code segment descriptor.</td>
<td>#NP</td>
<td>New Code Segment</td>
</tr>
<tr>
<td>Stack segment DPL should equal its RPL.</td>
<td>#TS</td>
<td>New Stack Segment</td>
</tr>
<tr>
<td>DS, ES, FS, and GS segment selectors are valid 4.</td>
<td>#TS</td>
<td>New Data Segment</td>
</tr>
<tr>
<td>DS, ES, FS, and GS segments are readable.</td>
<td>#TS</td>
<td>New Data Segment</td>
</tr>
<tr>
<td>P bits are set in descriptors of DS, ES, FS, and GS segments.</td>
<td>#NP</td>
<td>New Data Segment</td>
</tr>
<tr>
<td>DS, ES, FS, and GS segment DPL greater than or equal to CPL (unless these are conforming segments).</td>
<td>#TS</td>
<td>New Data Segment</td>
</tr>
<tr>
<td>Shadow Stack Pointer in a task not aligned to 8 bytes (for task switch initiated by a call, interrupt, or exception).3</td>
<td>#TS</td>
<td>New Task's TSS</td>
</tr>
<tr>
<td>If EFLAGS.VM=1 and shadow stacks are enabled.3</td>
<td>#TS</td>
<td>New Task's TSS</td>
</tr>
<tr>
<td>Supervisor Shadow Stack Token verification failures (for task switch initiated by a call, interrupt, jump, or exception):3</td>
<td>#TS</td>
<td>New Task's TSS</td>
</tr>
<tr>
<td>- Busy bit already set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Address in Shadow stack token does not match SSP value from TSS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>If task switch initiated by IRET, CS and LIP stored on old task shadow stack does not match CS and LIP of new task.3</td>
<td>#CP</td>
<td>FAR-RET/IRET</td>
</tr>
<tr>
<td>If task switch initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is &lt; 3) OR the SSP from IA32.PL3_SSP (if new task CPL = 3) fails the following checks:3</td>
<td>#CP</td>
<td>FAR-RET/IRET</td>
</tr>
<tr>
<td>- Not aligned to 4 bytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Is beyond 4G.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. #NP is segment-not-present exception, #GP is general-protection exception, #TS is invalid-TSS exception, and #SS is stack-fault exception.
2. The error code contains an index to the segment descriptor referenced in this column.
3. Valid when CET is enabled.
4. A segment selector is valid if it is in a compatible type of table (GDT or LDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (for example, a segment selector in the CS register only is valid when it points to a code-segment descriptor).
8.4 TASK LINKING

The previous task link field of the TSS (sometimes called the “backlink”) and the NT flag in the EFLAGS register are used to return execution to the previous task. EFLAGS.NT = 1 indicates that the currently executing task is nested within the execution of another task.

When a CALL instruction, an interrupt, or an exception causes a task switch: the processor copies the segment selector for the current TSS to the previous task link field of the TSS for the new task; it then sets EFLAGS.NT = 1. If software uses an IRET instruction to suspend the new task, the processor checks for EFLAGS.NT = 1; it then uses the value in the previous task link field to return to the previous task. See Figures 8-8.

When a JMP instruction causes a task switch, the new task is not nested. The previous task link field is not used and EFLAGS.NT = 0. Use a JMP instruction to dispatch a new task when nesting is not desired.

Table 8-2 shows the busy flag (in the TSS segment descriptor), the NT flag, the previous task link field, and TS flag (in control register CR0) during a task switch.

The NT flag may be modified by software executing at any privilege level. It is possible for a program to set the NT flag and execute an IRET instruction. This might randomly invoke the task specified in the previous link field of the current task's TSS. To keep such spurious task switches from succeeding, the operating system should initialize the previous task link field in every TSS that it creates to 0.

**Table 8-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag**

<table>
<thead>
<tr>
<th>Flag or Field</th>
<th>Effect of JMP instruction</th>
<th>Effect of CALL Instruction or Interrupt</th>
<th>Effect of IRET Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy (B) flag of new task.</td>
<td>Flag is set. Must have been clear before.</td>
<td>Flag is set. Must have been clear before.</td>
<td>No change. Must have been set.</td>
</tr>
<tr>
<td>Busy flag of old task.</td>
<td>Flag is cleared.</td>
<td>No change. Flag is currently set.</td>
<td>Flag is cleared.</td>
</tr>
<tr>
<td>NT flag of new task.</td>
<td>Set to value from TSS of new task.</td>
<td>Flag is set.</td>
<td>Set to value from TSS of new task.</td>
</tr>
<tr>
<td>NT flag of old task.</td>
<td>No change.</td>
<td>No change.</td>
<td>Flag is cleared.</td>
</tr>
<tr>
<td>Previous task link field of new task.</td>
<td>No change.</td>
<td>No change.</td>
<td>No change.</td>
</tr>
<tr>
<td>Previous task link field of old task.</td>
<td>No change.</td>
<td>No change.</td>
<td>No change.</td>
</tr>
<tr>
<td>TS flag in control register CR0.</td>
<td>Flag is set.</td>
<td>Flag is set.</td>
<td>Flag is set.</td>
</tr>
</tbody>
</table>
8.4.1 Use of Busy Flag To Prevent Recursive Task Switching

A TSS allows only one context to be saved for a task; therefore, once a task is called (dispatched), a recursive (or re-entrant) call to the task would cause the current state of the task to be lost. The busy flag in the TSS segment descriptor is provided to prevent re-entrant task switching and a subsequent loss of task state information. The processor manages the busy flag as follows:

1. When dispatching a task, the processor sets the busy flag of the new task.
2. If during a task switch, the current task is placed in a nested chain (the task switch is being generated by a CALL instruction, an interrupt, or an exception), the busy flag for the current task remains set.
3. When switching to the new task (initiated by a CALL instruction, interrupt, or exception), the processor generates a general-protection exception (#GP) if the busy flag of the new task is already set. If the task switch is initiated with an IRET instruction, the exception is not raised because the processor expects the busy flag to be set.
4. When a task is terminated by a jump to a new task (initiated with a JMP instruction in the task code) or by an IRET instruction in the task code, the processor clears the busy flag, returning the task to the “not busy” state.

The processor prevents recursive task switching by preventing a task from switching to itself or to any task in a nested chain of tasks. The chain of nested suspended tasks may grow to any length, due to multiple calls, interrupts, or exceptions. The busy flag prevents a task from being invoked if it is in this chain.

The busy flag may be used in multiprocessor configurations, because the processor follows a LOCK protocol (on the bus or in the cache) when it sets or clears the busy flag. This lock keeps two processors from invoking the same task at the same time. See Section 9.1.2.1, “Automatic Locking,” for more information about setting the busy flag in a multiprocessor applications.

8.4.2 Modifying Task Linkages

In a uniprocessor system, in situations where it is necessary to remove a task from a chain of linked tasks, use the following procedure to remove the task:

1. Disable interrupts.
2. Change the previous task link field in the TSS of the pre-empting task (the task that suspended the task to be removed). It is assumed that the pre-empting task is the next task (newer task) in the chain from the task to be removed. Change the previous task link field to point to the TSS of the next oldest task in the chain or to an even older task in the chain.
3. Clear the busy (B) flag in the TSS segment descriptor for the task being removed from the chain. If more than one task is being removed from the chain, the busy flag for each task being remove must be cleared.
4. Enable interrupts.

In a multiprocessing system, additional synchronization and serialization operations must be added to this procedure to ensure that the TSS and its segment descriptor are both locked when the previous task link field is changed and the busy flag is cleared.

8.5 TASK ADDRESS SPACE

The address space for a task consists of the segments that the task can access. These segments include the code, data, stack, and system segments referenced in the TSS and any other segments accessed by the task code. The segments are mapped into the processor’s linear address space, which is in turn mapped into the processor’s physical address space (either directly or through paging).

The LDT segment field in the TSS can be used to give each task its own LDT. Giving a task its own LDT allows the task address space to be isolated from other tasks by placing the segment descriptors for all the segments associated with the task in the task’s LDT.

It also is possible for several tasks to use the same LDT. This is a memory-efficient way to allow specific tasks to communicate with or control each other, without dropping the protection barriers for the entire system.
Because all tasks have access to the GDT, it also is possible to create shared segments accessed through segment descriptors in this table.

If paging is enabled, the CR3 register (PDBR) field in the TSS allows each task to have its own set of page tables for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

### 8.5.1 Mapping Tasks to the Linear and Physical Address Spaces

Tasks can be mapped to the linear address space and physical address space in one of two ways:

- **One linear-to-physical address space mapping is shared among all tasks.** — When paging is not enabled, this is the only choice. Without paging, all linear addresses map to the same physical addresses. When paging is enabled, this form of linear-to-physical address space mapping is obtained by using one page directory for all tasks. The linear address space may exceed the available physical space if demand-paged virtual memory is supported.

- **Each task has its own linear address space that is mapped to the physical address space.** — This form of mapping is accomplished by using a different page directory for each task. Because the PDBR (control register CR3) is loaded on task switches, each task may have a different page directory.

The linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share physical addresses.

With either method of mapping task linear address spaces, the TSSs for all tasks must lie in a shared area of the physical space, which is accessible to all tasks. This mapping is required so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear address space mapped by the GDT also should be mapped to a shared area of the physical space; otherwise, the purpose of the GDT is defeated. Figure 8-9 shows how the linear address spaces of two tasks can overlap in the physical space by sharing page tables.

![Figure 8-9. Overlapping Linear-to-Physical Mappings](image-url)
8.5.2 Task Logical Address Space

To allow the sharing of data among tasks, use the following techniques to create shared logical-to-physical address-space mappings for data segments:

- **Through the segment descriptors in the GDT** — All tasks must have access to the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear-address space that are mapped into an area of the physical-address space common to all tasks, then all tasks can share the data and code in those segments.

- **Through a shared LDT** — Two or more tasks can use the same LDT if the LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments that are mapped to a common area of the physical address space, the data and code in those segments can be shared among the tasks that share the LDT. This method of sharing is more selective than sharing through the GDT, because the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared segments.

- **Through segment descriptors in distinct LDTs that are mapped to common addresses in linear address space** — If this common area of the linear address space is mapped to the same area of the physical address space for each task, these segment descriptors permit the tasks to share segments. Such segment descriptors are commonly called aliases. This method of sharing is even more selective than those listed above, because, other segment descriptors in the LDTs may point to independent linear addresses which are not shared.

8.6 16-BIT TASK-STATE SEGMENT (TSS)

The 32-bit IA-32 processors also recognize a 16-bit TSS format like the one used in Intel 286 processors (see Figure 8-10). This format is supported for compatibility with software written to run on earlier IA-32 processors.

The following information is important to know about the 16-bit TSS:

- Do not use a 16-bit TSS to implement a virtual-8086 task.
- The valid segment limit for a 16-bit TSS is 2CH.
- The 16-bit TSS does not contain a field for the base address of the page directory, which is loaded into control register CR3. A separate set of page tables for each task is not supported for 16-bit tasks. If a 16-bit task is dispatched, the page-table structure for the previous task is used.
- The I/O base address is not included in the 16-bit TSS. None of the functions of the I/O map are supported.
- When task state is saved in a 16-bit TSS, the upper 16 bits of the EFLAGS register and the EIP register are lost.
- When the general-purpose registers are loaded or saved from a 16-bit TSS, the upper 16 bits of the registers are modified and not maintained.
8.7 TASK MANAGEMENT IN 64-BIT MODE

In 64-bit mode, task structure and task state are similar to those in protected mode. However, the task switching mechanism available in protected mode is not supported in 64-bit mode. Task management and switching must be performed by software. The processor issues a general-protection exception (#GP) if the following is attempted in 64-bit mode:

- Control transfer to a TSS or a task gate using JMP, CALL, INT n, INT3, INTO, INT1, or interrupt.
- An IRET with EFLAGS.NT (nested task) set to 1.

Although hardware task-switching is not supported in 64-bit mode, a 64-bit task state segment (TSS) must exist. Figure 8-11 shows the format of a 64-bit TSS. The TSS holds information important to 64-bit mode and that is not directly related to the task-switch mechanism. This information includes:

- **RSPn** — The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0-2.
- **ISTn** — The full 48-bit canonical forms of the interrupt stack table (IST) pointers.
- **I/O map base address** — The 16-bit offset to the I/O permission bit map from the 64-bit TSS base.

The operating system must create at least one 64-bit TSS after activating IA-32e mode. It must execute the LTR instruction (in 64-bit mode) to load the TR register with a pointer to the 64-bit TSS responsible for both 64-bit-mode programs and compatibility-mode programs.
### Figure 8-11. 64-Bit TSS Format

<table>
<thead>
<tr>
<th>I/O Map Base Address</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>100</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>96</td>
<td>96</td>
</tr>
</tbody>
</table>

Reserved bits. Set to 0.
The Intel 64 and IA-32 architectures provide mechanisms for managing and improving the performance of multiple processors connected to the same system bus. These include:

- Bus locking and/or cache coherency management for performing atomic operations on system memory.
- Serializing instructions.
- An advance programmable interrupt controller (APIC) located on the processor chip (see Chapter 11, “Advanced Programmable Interrupt Controller (APIC)”). This feature was introduced by the Pentium processor.
- A second-level cache (level 2, L2). For the Pentium 4, Intel Xeon, and P6 family processors, the L2 cache is included in the processor package and is tightly coupled to the processor. For the Pentium and Intel 486 processors, pins are provided to support an external L2 cache.
- A third-level cache (level 3, L3). For Intel Xeon processors, the L3 cache is included in the processor package and is tightly coupled to the processor.
- Intel Hyper-Threading Technology. This extension to the Intel 64 and IA-32 architectures enables a single processor core to execute two or more threads concurrently (see Section 9.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology”).

These mechanisms are particularly useful in symmetric-multiprocessing (SMP) systems. However, they can also be used when an Intel 64 or IA-32 processor and a special-purpose processor (such as a communications, graphics, or video processor) share the system bus.

These multiprocessing mechanisms have the following characteristics:

- To maintain system memory coherency — When two or more processors are attempting simultaneously to access the same address in system memory, some communication mechanism or memory access protocol must be available to promote data coherency and, in some instances, to allow one processor to temporarily lock a memory location.
- To maintain cache consistency — When one processor accesses data cached on another processor, it must not receive incorrect data. If it modifies data, all other processors that access that data must receive the modified data.
- To allow predictable ordering of writes to memory — In some circumstances, it is important that memory writes be observed externally in precisely the same order as programmed.
- To distribute interrupt handling among a group of processors — When several processors are operating in a system in parallel, it is useful to have a centralized mechanism for receiving interrupts and distributing them to available processors for servicing.
- To increase system performance by exploiting the multi-threaded and multi-process nature of contemporary operating systems and applications.

The caching mechanism and cache consistency of Intel 64 and IA-32 processors are discussed in Chapter 12. The APIC architecture is described in Chapter 11. Bus and memory locking, serializing instructions, memory ordering, and Intel Hyper-Threading Technology are discussed in the following sections.

### 9.1 Locked Atomic Operations

The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:

- Guaranteed atomic operations.
- Bus locking, using the LOCK# signal and the LOCK instruction prefix.
Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors. These mechanisms are interdependent in the following ways. Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor’s L1 or L2 caches, atomic operations can often be carried out inside a processor’s caches without asserting the bus lock. Here the processor’s cache coherency protocols ensure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

**NOTE**

Where there are contested lock accesses, software may need to implement algorithms that ensure fair access to resources in order to prevent lock starvation. The hardware provides no resource that guarantees fairness to participating agents. It is the responsibility of software to manage the fairness of semaphores and exclusive locking functions.

The mechanisms for handling locked atomic operations have evolved with the complexity of IA-32 processors. More recent IA-32 processors (such as the Pentium 4, Intel Xeon, and P6 family processors) and Intel 64 provide a more refined locking mechanism than earlier processors. These mechanisms are described in the following sections.

### 9.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte.
- Reading or writing a word aligned on a 16-bit boundary.
- Reading or writing a doubleword aligned on a 32-bit boundary.

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary.
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus.

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line.

Processors that enumerate support for Intel® AVX (by setting the feature flag CPUID.01H:ECX.AVX[bit 28]) guarantee that the 16-byte memory operations performed by the following instructions will always be carried out atomically:

- MOVAPD, MOVAPS, and MOVDQA.
- VMOVAPD, VMOVAPS, and VMOVQDQA when encoded with VEX.128.
- VMOVAPD, VMOVAPS, VMOVQDQA32, and VMOVQDQA64 when encoded with EVEX.128 and k0 (masking disabled).

(Note that these instructions require the linear addresses of their memory operands to be 16-byte aligned.)

Accesses to cacheable memory that are split across cache lines and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, and P6 family processors provide bus control signals that permit external memory subsystems to make split accesses atomic; however, nonaligned data accesses will seriously impact the performance of the processor and should be avoided.
Except as noted above, an x87 instruction or an SSE instruction that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory, some of the accesses may complete (writing to memory) while another causes the operation to fault for architectural reasons (e.g., due an page-table entry that is marked "not present"). In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault. If TLB invalidation has been delayed (see Section 4.10.4.4), such page faults may occur even if all accesses are to the same page.

9.1.2  Bus Locking

Intel 64 and IA-32 processors provide a LOCK# signal that is asserted automatically during certain critical memory operations to lock the system bus or equivalent link. Assertion of this signal is called a bus lock. While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked. Software can specify other occasions when the LOCK semantics are to be followed by prepending the LOCK prefix to an instruction.

In the case of the Intel386, Intel486, and Pentium processors, explicitly locked instructions will result in the assertion of the LOCK# signal. It is the responsibility of the hardware designer to make the LOCK# signal available in system hardware to control memory accesses among processors.

For the P6 and more recent processor families, if the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted; instead, locking is only applied to the processor's caches (see Section 9.1.4, "Effects of a LOCK Operation on Internal Processor Caches"). These processors will assert a bus lock for a locked access in either of the following situations: (1) the access is to multiple cache lines (a split lock); or (2) the access uses a memory type other than WB (a UC lock).1

9.1.2.1  Automatic Locking

The operations on which the processor automatically follows the LOCK semantics are as follows:

• When executing an XCHG instruction that references memory.

• When switching to a task, the processor tests and sets the busy flag in the type field of the TSS descriptor. To ensure that two processors do not switch to the same task simultaneously, the processor follows the LOCK semantics while testing and setting this flag.

• When loading a segment descriptor, the processor sets the accessed flag in the segment descriptor if the flag is clear. During this operation, the processor follows the LOCK semantics so that the descriptor will not be modified by another processor while it is being updated. For this action to be effective, operating-system procedures that update descriptors should use the following steps:
  — Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is not-present, and specify a value for the type field that indicates that the descriptor is being updated.
  — Update the fields of the segment descriptor. (This operation may require several memory accesses; therefore, locked operations cannot be used.)
  — Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is valid and present.
  — The Intel386 processor always updates the accessed flag in the segment descriptor, whether it is clear or not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors only update this flag if it is not already set.

• The processor uses locked cycles to set the accessed and dirty flag in paging-structure entries.

• After an interrupt request, an interrupt controller may use the data bus to send the interrupt's vector to the processor. The processor follows the LOCK semantics during this time to ensure that no other data appears on the data bus while the vector is being transmitted.

1. The term "UC lock" is used because the most common situation regards accesses to UC memory. Despite the name, locked accesses to WC, WP, and WT memory also cause bus locks.
9.1.2.2 Software Controlled Bus Locking

To explicitly force the LOCK semantics, software can use the LOCK prefix with the following instructions when they are used to modify a memory location. An invalid-opcode exception (#UD) is generated when the LOCK prefix is used with any other instruction or when no write operation is made to memory (that is, when the destination operand is in a register).

- The bit test and modify instructions (BTS, BTR, and BTC).
- The exchange instructions (XADD, CMPXCHG, CMPXCHG8B, and CMPXCHG16B).
- The LOCK prefix is automatically assumed for XCHG instruction.
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG.
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area.

Software should access semaphores (shared memory used for signalling between multiple processors) using identical addresses and operand lengths. For example, if one processor accesses a semaphore using a word access, other processors should not access the semaphore using a byte access.

**NOTE**

Do not implement semaphores using the WC memory type. Do not perform non-temporal stores to a cache line containing a location used to implement a semaphore.

The integrity of a bus lock is not affected by the alignment of the memory field. The LOCK semantics are followed for as many bus cycles as necessary to update the entire operand. However, it is recommend that locked accesses be aligned on their natural boundaries for better system performance:

- Any boundary for an 8-bit access (locked or otherwise).
- 16-bit boundary for locked word accesses.
- 32-bit boundary for locked doubleword accesses.
- 64-bit boundary for locked quadword accesses.

Locked operations are atomic with respect to all other memory operations and all externally visible events. Only instruction fetch and page table accesses can pass locked instructions. Locked instructions can be used to synchronize data written by one processor and read by another processor.

For the P6 family processors, locked operations serialize all outstanding load and store operations (that is, wait for them to complete). This rule is also true for the Pentium 4 and Intel Xeon processors, with one exception. Load operations that reference weakly ordered memory types (such as the WC memory type) may not be serialized.

Locked instructions should not be used to ensure that data written can be fetched as instructions.

**NOTE**

The locked instructions for the current versions of the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors allow data written to be fetched as instructions. However, Intel recommends that developers who require the use of self-modifying code use a different synchronizing mechanism, described in the following sections.

9.1.2.3 Features to Disable Bus Locks

Because bus locks may adversely affect performance in certain situations, processors may support two features that system software can use to disable bus locking. These are called **UC-lock disable** and **split-lock disable**.

A processor enumerates support for UC-lock disable by setting bit 4 of the IA32_CORE_CAPABILITIES MSR (MSR index CFH). Support for split-lock disable is enumerated by IA32_CORE_CAPABILITIES[5].

Software enables UC-lock disable by setting bit 28 of the MSR_MEMORY_CTRL MSR (MSR index 33H). When this bit is set, a locked access using a memory type other than WB causes a general-protection exception (#GP) with a zero error code. The locked access does not occur.
Software enables split-lock disable by setting MSR_MEMORY_CTRL[29]. When this bit is set, a locked access to multiple cache lines causes an alignment-check exception (#AC) with a zero error code.\(^2\) The locked access does not occur.

If both features are enabled, a locked access to multiple cache lines causes #AC(0) regardless of the memory type(s) being accessed.

While MSR_MEMORY_CTRL is not an architectural MSR, the behavior described above is consistent across processor models that enumerate the support in IA32_CORE_CAPABILITIES.

In addition to these features that disable bus locks, there are features that allow software to detect when a bus lock has occurred. See Section 18.3.1.6 for information about OS bus-lock detection and Section 26.2 for information about the VMM bus-lock detection.

### 9.1.3 Handling Self- and Cross-Modifying Code

The act of a processor writing data into a currently executing code segment with the intent of executing that data as code is called **self-modifying code**. IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified.

As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred. To write self-modifying code and ensure that it is compliant with current and future versions of the IA-32 architectures, use one of the following coding options:

(*) **OPTION 1** (*)  
Store modified code (as data) into code segment;  
Jump to new code or an intermediate location;  
Execute new code;

(*) **OPTION 2** (*)  
Store modified code (as data) into code segment;  
Execute a serializing instruction; (* For example, CPUID instruction *)  
Execute new code;

The use of one of these options is not required for programs intended to run on the Pentium or Intel486 processors, but are recommended to ensure compatibility with the P6 and more recent processor families.

Self-modifying code will execute at a lower level of performance than non-self-modifying or normal code. The degree of the performance deterioration will depend upon the frequency of modification and specific characteristics of the code.

The act of one processor writing data into the currently executing code segment of a second processor with the intent of having the second processor execute that data as code is called **cross-modifying code**. As with self-modifying code, IA-32 processors exhibit model-specific behavior when executing cross-modifying code, depending upon how far ahead of the executing processors current execution pointer the code has been modified.

To write cross-modifying code and ensure that it is compliant with current and future versions of the IA-32 architecture, the following processor synchronization algorithm must be implemented:

(*) **Action of Modifying Processor** (*)  
Memory_Flag := 0; (* Set Memory_Flag to value other than 1 *)  
Store modified code (as data) into code segment;

(*) **Action of Executing Processor** (*)  
WHILE (Memory_Flag \(\neq\) 1)  
Wait for code to update;

\(^2\) Other alignment-check exceptions occur only if CR0.AM = 1, EFLAGS.AC = 1, and CPL = 3. The alignment-check exceptions resulting from split-lock disable may occur even if CR0.AM = 0, EFLAGS.AC = 0, or CPL < 3.
9.1.4 Effects of a LOCK Operation on Internal Processor Caches

For the Intel486 and Pentium processors, the LOCK# signal is always asserted on the bus during a LOCK operation, even if the area of memory being locked is cached in the processor.

For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow it's cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called “cache locking.” The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.

9.2 MEMORY ORDERING

The term memory ordering refers to the order in which the processor issues reads (loads) and writes (stores) through the system bus to system memory. The Intel 64 and IA-32 architectures support several memory-ordering models depending on the implementation of the architecture. For example, the Intel386 processor enforces program ordering (generally referred to as strong ordering), where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

To allow performance optimization of instruction execution, the IA-32 architecture allows departures from strong-ordering model called processor ordering in Pentium 4, Intel Xeon, and P6 family processors. These processor-ordering variations (called here the memory-ordering model) allow performance enhancing operations such as allowing reads to go ahead of buffered writes. The goal of any of these variations is to increase instruction execution speeds, while maintaining memory coherency, even in multiple-processor systems.

Section 9.2.1 and Section 9.2.2 describe the memory-ordering implemented by Intel486, Pentium, Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors. Section 9.2.3 gives examples illustrating the behavior of the memory-ordering model on IA-32 and Intel-64 processors. Section 9.2.4 considers the special treatment of stores for string operations and Section 9.2.5 discusses how memory-ordering behavior may be modified through the use of specific instructions.

9.2.1 Memory Ordering in the Intel® Pentium® and Intel486™ Processors

The Pentium and Intel486 processors follow the processor-ordered memory model; however, they operate as strongly-ordered processors under most circumstances. Reads and writes always appear in programmed order at the system bus—except for the following situation where processor ordering is exhibited. Read misses are permitted to go ahead of buffered writes on the system bus when all the buffered writes are cache hits and, therefore, are not directed to the same address being accessed by the read miss.

In the case of I/O operations, both reads and writes always appear in programmed order.

Software intended to operate correctly in processor-ordered processors (such as the Pentium 4, Intel Xeon, and P6 family processors) should not depend on the relatively strong ordering of the Pentium or Intel486 processors. Instead, it should ensure that accesses to shared variables that are intended to control concurrent execution among processors are explicitly required to obey program ordering through the use of appropriate locking or serializing operations (see Section 9.2.5, “Strengthening or Weakening the Memory-Ordering Model”).
9.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as “write ordered with store-buffer forwarding.” This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (Note the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term “processor” refers to a logical processor. For example, a physical processor supporting multiple cores and/or Intel Hyper-Threading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
  - Streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
  - String operations (see Section 9.2.4.1).
- No write to memory may be reordered with an execution of the CLFLUSH instruction; a write may be reordered with an execution of the CLFLUSHOPT instruction that flushes a cache line other than the one being written.3 Executions of the CLFLUSH instruction are not reordered with each other. Executions of CLFLUSHOPT that access different cache lines may be reordered with each other. An execution of CLFLUSHOPT may be reordered with an execution of CLFLUSH that accesses a different cache line.
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes or executions of CLFLUSH and CLFLUSHOPT.
- MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.

In a multiple-processor system, the following ordering principles apply:

- Individual processors use the same ordering principles as in a single-processor system.
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are NOT ordered with respect to the writes from other processors.
- Memory ordering obeys causality (memory ordering respects transitive visibility).
- Any two stores are seen in a consistent order by processors other than those performing the stores.
- Locked instructions have a total order.

See the example in Figure 9-1. Consider three processors in a system and each processor performs three writes, one to each of three defined locations (A, B, and C). Individually, the processors perform the writes in the same program order, but because of bus arbitration and other memory access mechanisms, the order that the three processors write the individual memory locations can differ each time the respective code sequences are executed on the processors. The final values in location A, B, and C would possibly vary on each execution of the write sequence.

The processor-ordering model described in this section is virtually identical to that used by the Pentium and Intel486 processors. The only enhancements in the Pentium 4, Intel Xeon, and P6 family processors are:

- Added support for speculative reads, while still adhering to the ordering principles above.
- Store-buffer forwarding, when a read passes a write to the same memory location.

---

3. Earlier versions of this manual specified that writes to memory may be reordered with executions of the CLFLUSH instruction. No processors implementing the CLFLUSH instruction allow such reordering.
• Out of order store from long string store and string move operations (see Section 9.2.4, “Fast-String Operation and Out-of-Order Stores,” below).

9.2.3 Examples Illustrating the Memory-Ordering Principles

This section provides a set of examples that illustrate the behavior of the memory-ordering principles introduced in Section 9.2.2. They are designed to give software writers an understanding of how memory ordering may affect the results of different sequences of instructions.

These examples are limited to accesses to memory regions defined as write-back cacheable (WB). (Section 9.2.3.1 describes other limitations on the generality of the examples.) The reader should understand that they describe only software-visible behavior. A logical processor may reorder two accesses even if one of examples indicates that they may not be reordered. Such an example states only that software cannot detect that such a reordering occurred. Similarly, a logical processor may execute a memory access more than once as long as the behavior visible to software is consistent with a single execution of the memory access.

9.2.3.1 Assumptions, Terminology, and Notation

As noted above, the examples in this section are limited to accesses to memory regions defined as write-back cacheable (WB). They apply only to ordinary loads stores and to locked read-modify-write instructions. They do not necessarily apply to any of the following: out-of-order stores for string instructions (see Section 9.2.4); accesses with a non-temporal hint; reads from memory by the processor as part of address translation (e.g., page walks); and updates to segmentation and paging structures by the processor (e.g., to update “accessed” bits).

The principles underlying the examples in this section apply to individual memory accesses and to locked read-modify-write instructions. The Intel-64 memory-ordering model guarantees that, for each of the following memory-access instructions, the constituent memory operation appears to execute as a single memory access:
• Instructions that read or write a single byte.
• Instructions that read or write a word (2 bytes) whose address is aligned on a 2 byte boundary.
• Instructions that read or write a doubleword (4 bytes) whose address is aligned on a 4 byte boundary.
• Instructions that read or write a quadword (8 bytes) whose address is aligned on an 8 byte boundary.

Any locked instruction (either the XCHG instruction or another read-modify-write instruction with a LOCK prefix) appears to execute as an indivisible and uninterruptible sequence of load(s) followed by store(s) regardless of alignment.

Other instructions may be implemented with multiple memory accesses. From a memory-ordering point of view, there are no guarantees regarding the relative order in which the constituent memory accesses are made. There is also no guarantee that the constituent operations of a store are executed in the same order as the constituent operations of a load.

Section 9.2.3.2 through Section 9.2.3.7 give examples using the MOV instruction. The principles that underlie these examples apply to load and store accesses in general and to other instructions that load from or store to memory. Section 9.2.3.8 and Section 9.2.3.9 give examples using the XCHG instruction. The principles that underlie these examples apply to other locked read-modify-write instructions.

This section uses the term “processor” is to refer to a logical processor. The examples are written using Intel-64 assembly-language syntax and use the following notational conventions:

• Arguments beginning with an “r”, such as r1 or r2 refer to registers (e.g., EAX) visible only to the processor being considered.
• Memory locations are denoted with x, y, z.
• Stores are written as mov [_x], val, which implies that val is being stored into the memory location x.
• Loads are written as mov r, [_x], which implies that the contents of the memory location x are being loaded into the register r.

As noted earlier, the examples refer only to software visible behavior. When the succeeding sections make statement such as “the two stores are reordered,” the implication is only that “the two stores appear to be reordered from the point of view of software.”

9.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations

The Intel-64 memory-ordering model allows neither loads nor stores to be reordered with the same kind of operation. That is, it ensures that loads are seen in program order and that stores are seen in program order. This is illustrated by the following example:

Example 9-1. Stores Are Not Reordered with Other Stores

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [_x], 1</td>
<td>mov r1, [_y]</td>
</tr>
<tr>
<td>mov [_y], 1</td>
<td>mov r2, [_x]</td>
</tr>
</tbody>
</table>

Initially x = y = 0
r1 = 1 and r2 = 0 is not allowed

The disallowed return values could be exhibited only if processor 0’s two stores are reordered (with the two loads occurring between them) or if processor 1’s two loads are reordered (with the two stores occurring between them).

If r1 = 1, the store to y occurs before the load from y. Because the Intel-64 memory-ordering model does not allow stores to be reordered, the earlier store to x occurs before the load from y. Because the Intel-64 memory-ordering model does not allow loads to be reordered, the store to x also occurs before the later load from x. This r2 = 1.

9.2.3.3 Stores Are Not Reordered With Earlier Loads

The Intel-64 memory-ordering model ensures that a store by a processor may not occur before a previous load by the same processor. This is illustrated in Example 9-2.
Assume \( r_1 = 1 \).
- Because \( r_1 = 1 \), processor 1’s store to \( x \) occurs before processor 0’s load from \( x \).
- Because the Intel-64 memory-ordering model prevents each store from being reordered with the earlier load by the same processor, processor 1’s load from \( y \) occurs before its store to \( x \).
- Similarly, processor 0’s load from \( x \) occurs before its store to \( y \).
- Thus, processor 1’s load from \( y \) occurs before processor 0’s store to \( y \), implying \( r_2 = 0 \).

### 9.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may be reordered with an earlier store to a different location is illustrated by the following example:

**Example 9-3. Loads May Be Reordered with Older Stores**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ([_x]), 1</td>
<td>mov ([_y]), 1</td>
</tr>
<tr>
<td>mov ([_y]), 1</td>
<td>mov ([_x]), 1</td>
</tr>
</tbody>
</table>

Initially \( x = y = 0 \)

\( r_1 = 0 \) and \( r_2 = 0 \) is allowed

At each processor, the load and the store are to different locations and hence may be reordered. Any interleaving of the operations is thus allowed. One such interleaving has the two loads occurring before the two stores. This would result in each load returning value 0.

The fact that a load may not be reordered with an earlier store to the same location is illustrated by the following example:

**Example 9-4. Loads Are not Reordered with Older Stores to the Same Location**

<table>
<thead>
<tr>
<th>Processor 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ([_x]), 1</td>
</tr>
<tr>
<td>mov ([_x]), 1</td>
</tr>
</tbody>
</table>

Initially \( x = 0 \)

\( r_1 = 0 \) is not allowed

The Intel-64 memory-ordering model does not allow the load to be reordered with the earlier store because the accesses are to the same location. Therefore, \( r_1 = 1 \) must hold.
9.2.3.5  Intra-Processor Forwarding Is Allowed

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other. This is illustrated by the following example:

**Example 9-5. Intra-Processor Forwarding is Allowed**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [._x], 1</td>
<td>mov [._y], 1</td>
</tr>
<tr>
<td>mov r1, [._x]</td>
<td>mov r3, [._y]</td>
</tr>
<tr>
<td>mov r2, [._y]</td>
<td>mov r4, [._x]</td>
</tr>
</tbody>
</table>

Initially x = y = 0
r2 = 0 and r4 = 0 is allowed

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1’s, while processor 1 sees its store before seeing processor 0’s. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor’s own loads but is not visible to (and cannot satisfy) loads by other processors.

9.2.3.6  Stores Are Transitively Visible

The memory-ordering model ensures transitive visibility of stores; stores that are causally related appear to all processors to occur in an order consistent with the causality relation. This is illustrated by the following example:

**Example 9-6. Stores Are Transitively Visible**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [._x], 1</td>
<td>mov r1, [._x]</td>
<td>mov r2, [._y]</td>
</tr>
<tr>
<td>mov [._y], 1</td>
<td>mov r3, [._y]</td>
<td>mov r3, [._x]</td>
</tr>
</tbody>
</table>

Initially x = y = 0
r1 = 1, r2 = 1, r3 = 0 is not allowed

Assume that r1 = 1 and r2 = 1.
• Because r1 = 1, processor 0’s store occurs before processor 1’s load.
• Because the memory-ordering model prevents a store from being reordered with an earlier load (see Section 9.2.3.3), processor 1’s load occurs before its store. Thus, processor 0’s store causally precedes processor 1’s store.
• Because processor 0’s store causally precedes processor 1’s store, the memory-ordering model ensures that processor 0’s store appears to occur before processor 1’s store from the point of view of all processors.
• Because r2 = 1, processor 1’s store occurs before processor 2’s load.
• Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 9.2.3.2), processor 2’s load occur in order.
• The above items imply that processor 0’s store to x occurs before processor 2’s load from x. This implies that r3 = 1.
9.2.3.7 Stores Are Seen in a Consistent Order by Other Processors

As noted in Section 9.2.3.5, the memory-ordering model allows stores by two processors to be seen in different orders by those two processors. However, any two stores must appear to execute in the same order to all processors other than those performing the stores. This is illustrated by the following example:

Example 9-7. Stores Are Seen in a Consistent Order by Other Processors

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [ _x], 1</td>
<td>mov [ _y], 1</td>
<td>mov r1, [ _x]</td>
<td>mov r3, [ _y]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov r2, [ _y]</td>
<td>mov r4, [ _x]</td>
</tr>
</tbody>
</table>

Initially x = y = 0
r1 = 1, r2 = 0, r3 = 1, r4 = 0 is not allowed

By the principles discussed in Section 9.2.3.2:
• Processor 2’s first and second load cannot be reordered.
• Processor 3’s first and second load cannot be reordered.
• If r1 = 1 and r2 = 0, processor 0’s store appears to precede processor 1’s store with respect to processor 2.
• Similarly, r3 = 1 and r4 = 0 imply that processor 1’s store appears to precede processor 0’s store with respect to processor 1.

Because the memory-ordering model ensures that any two stores appear to execute in the same order to all processors (other than those performing the stores), this set of return values is not allowed.

9.2.3.8 Locked Instructions Have a Total Order

The memory-ordering model ensures that all processors agree on a single execution order of all locked instructions, including those that are larger than 8 bytes or are not naturally aligned. This is illustrated by the following example:

Example 9-8. Locked Instructions Have a Total Order

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg [ _x], r1</td>
<td>xchg [ _y], r2</td>
<td>mov r3, [ _x]</td>
<td>mov r5, [ _y]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov r4, [ _y]</td>
<td>mov r6, [ _x]</td>
</tr>
</tbody>
</table>

Initially r1 = r2 = 1, x = y = 0
r3 = 1, r4 = 0, r5 = 1, r6 = 0 is not allowed

Processor 2 and processor 3 must agree on the order of the two executions of XCHG. Without loss of generality, suppose that processor 0’s XCHG occurs first.
• If r5 = 1, processor 1’s XCHG into y occurs before processor 3’s load from y.
• Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 9.2.3.2), processor 3’s loads occur in order and, therefore, processor 1’s XCHG occurs before processor 3’s load from x.
• Since processor 0’s XCHG into x occurs before processor 1’s XCHG (by assumption), it occurs before processor 3’s load from x. Thus, r6 = 1.

A similar argument (referring instead to processor 2’s loads) applies if processor 1’s XCHG occurs before processor 0’s XCHG.

9.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions

The memory-ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a
load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The first example illustrates that loads may not be reordered with earlier locked instructions:

**Example 9-9. Loads Are not Reordered with Locks**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg [_x], r1</td>
<td>xchg [_y], r3</td>
</tr>
<tr>
<td>mov r2, [_y]</td>
<td>mov r4, [_x]</td>
</tr>
</tbody>
</table>

Initially x = y = 0, r1 = r3 = 1
r2 = 0 and r4 = 0 is not allowed

As explained in Section 9.2.3.8, there is a total order of the executions of locked instructions. Without loss of generality, suppose that processor 0’s XCHG occurs first.

Because the Intel-64 memory-ordering model prevents processor 1’s load from being reordered with its earlier XCHG, processor 0’s XCHG occurs before processor 1’s load. This implies r4 = 1.

A similar argument (referring instead to processor 2’s accesses) applies if processor 1’s XCHG occurs before processor 0’s XCHG.

The second example illustrates that a store may not be reordered with an earlier locked instruction:

**Example 9-10. Stores Are not Reordered with Locks**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchg [_x], r1</td>
<td>mov r2, [_y]</td>
</tr>
<tr>
<td>mov [_y], 1</td>
<td>mov r3, [_x]</td>
</tr>
</tbody>
</table>

Initially x = y = 0, r1 = 1
r2 = 1 and r3 = 0 is not allowed

Assume r2 = 1.

- Because r2 = 1, processor 0’s store to y occurs before processor 1’s load from y.
- Because the memory-ordering model prevents a store from being reordered with an earlier locked instruction, processor 0’s XCHG into x occurs before its store to y. Thus, processor 0’s XCHG into x occurs before processor 1’s load from y.
- Because the memory-ordering model prevents loads from being reordered (see Section 9.2.3.2), processor 1’s loads occur in order and, therefore, processor 1’s XCHG into x occurs before processor 1’s load from x. Thus, r3 = 1.

### 9.2.4 Fast-String Operation and Out-of-Order Stores

Section 7.3.9.3 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, described an optimization of repeated string operations called **fast-string operation**.

As explained in that section, the stores produced by fast-string operation may appear to execute out of order. Software dependent upon sequential store ordering should not use string operations for the entire data structure to be stored. Data and semaphores should be separated. Order-dependent code should write to a discrete semaphore variable after any string operations to allow correctly ordered data to be seen by all processors. Atomicity of load and store operations is guaranteed only for native data elements of the string with native data size, and only if they are included in a single cache line.

Section 9.2.4.1 and Section 9.2.4.2 provide further explain and examples.

### 9.2.4.1 Memory-Ordering Model for String Operations on Write-Back (WB) Memory

This section deals with the memory-ordering model for string operations on write-back (WB) memory for the Intel 64 architecture.
The memory-ordering model respects the follow principles:

1. Stores within a single string operation may be executed out of order.

2. Stores from separate string operations (for example, stores from consecutive string operations) do not execute out of order. All the stores from an earlier string operation will complete before any store from a later string operation.

3. String operations are not reordered with other store operations.

Fast string operations (e.g., string operations initiated with the MOVS/STOS instructions and the REP prefix) may be interrupted by exceptions or interrupts. The interrupts are precise but may be delayed - for example, the interruptions may be taken at cache line boundaries, after every few iterations of the loop, or after operating on every few bytes. Different implementations may choose different options, or may even choose not to delay interrupt handling, so software should not rely on the delay. When the interrupt/trap handler is reached, the source/destination registers point to the next string element to be operated on, while the EIP stored in the stack points to the string instruction, and the ECX register has the value it held following the last successful iteration. The return from that trap/interrupt handler should cause the string instruction to be resumed from the point where it was interrupted.

The string operation memory-ordering principles, (item 2 and 3 above) should be interpreted by taking the in-correctibility of fast string operations into account. For example, if a fast string operation gets interrupted after k iterations, then stores performed by the interrupt handler will become visible after the fast string stores from iteration 0 to k, and before the fast string stores from the (k+1)th iteration onward.

Stores within a single string operation may execute out of order (item 1 above) only if fast string operation is enabled. Fast string operations are enabled/disabled through the IA32_MISC_ENABLE model specific register.

9.2.4.2  Examples Illustrating Memory-Ordering Principles for String Operations

The following examples uses the same notation and convention as described in Section 9.2.3.1.

In Example 9-11, processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location _x (kept in ES:EDI) in ascending order. Since each operation stores a doubleword (4 bytes), the operation is repeated 128 times (value in ECX). The block of memory initially contained 0. Processor 1 is reading two memory locations that are part of the memory block being updated by processor 0, i.e, reading locations in the range _x to (_x+511).

It is possible for processor 1 to perceive that the repeated string stores in processor 0 are happening out of order. Assume that fast string operations are enabled on processor 0.

In Example 9-12, processor 0 does two separate rounds of rep stosd operation of 128 doubleword stores, writing the value 1 (value in EAX) into the first block of 512 bytes from location _x (kept in ES:EDI) in ascending order. It then writes 1 into a second block of memory from (_x+512) to (_x+1023). All of the memory locations initially contain 0. The block of memory initially contained 0. Processor 1 performs two load operations from the two blocks of memory.
Example 9-12. Stores Across String Operations Are not Reordered

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep:stosd [ _x]</td>
<td>mov r1,[_z]</td>
</tr>
<tr>
<td>mov ecx, $128</td>
<td>mov r2,[_y]</td>
</tr>
<tr>
<td>rep:stosd 512[ _x]</td>
<td></td>
</tr>
</tbody>
</table>

Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x
Initially [ _x] to 1023[ _x]= 0, _x <= _y < _x+512 < _z < _x+1024
r1 = 1 and r2 = 0 is not allowed

It is not possible in the above example for processor 1 to perceive any of the stores from the later string operation (to the second 512 block) in processor 0 before seeing the stores from the earlier string operation to the first 512 block.

The above example assumes that writes to the second block (_x+512 to _x+1023) does not get executed while processor 0’s string operation to the first block has been interrupted. If the string operation to the first block by processor 0 is interrupted, and a write to the second memory block is executed by the interrupt handler, then that change in the second memory block will be visible before the string operation to the first memory block resumes.

In Example 9-13, processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location _x (kept in ES:EDI) in ascending order. It then writes to a second memory location outside the memory block of the previous string operation. Processor 1 performs two read operations, the first read is from an address outside the 512-byte block but to be updated by processor 0, the second ready is from inside the block of memory of string operation.

Example 9-13. String Operations Are not Reordered with later Stores

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep:stosd [ _x]</td>
<td>mov r1,[_z]</td>
</tr>
<tr>
<td>mov [ _z], $1</td>
<td>mov r2,[_y]</td>
</tr>
</tbody>
</table>

Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x
Initially [ _y] = [ _z] = 0, [ _x] to 511[ _x]= 0, _x <= _y < _x+512, _z is a separate memory location
r1 = 1 and r2 = 0 is not allowed

Processor 1 cannot perceive the later store by processor 0 until it sees all the stores from the string operation. Example 9-13 assumes that processor 0’s store to [ _z] is not executed while the string operation has been interrupted. If the string operation is interrupted and the store to [ _z] by processor 0 is executed by the interrupt handler, then changes to [ _z] will become visible before the string operation resumes.

Example 9-14 illustrates the visibility principle when a string operation is interrupted.

Example 9-14. Interrupted String Operation

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep:stosd [ _x] // interrupted before es:edi reach _y</td>
<td>mov r1,[_z]</td>
</tr>
<tr>
<td>mov [ _z], $1 // interrupt handler</td>
<td>mov r2,[_y]</td>
</tr>
</tbody>
</table>

Initially on processor 0: EAX = 1, ECX=128, ES:EDI = _x
Initially [ _y] = [ _z] = 0, [ _x] to 511[ _x]= 0, _x <= _y < _x+512, _z is a separate memory location
r1 = 1 and r2 = 0 is allowed
In Example 9-14, processor 0 started a string operation to write to a memory block of 512 bytes starting at address \(_x\). Processor 0 got interrupted after \(k\) iterations of store operations. The address \(_y\) has not yet been updated by processor 0 when processor 0 got interrupted. The interrupt handler that took control on processor 0 writes to the address \(_z\). Processor 1 may see the store to \(_z\) from the interrupt handler, before seeing the remaining stores to the 512-byte memory block that are executed when the string operation resumes.

Example 9-15 illustrates the ordering of string operations with earlier stores. No store from a string operation can be visible before all prior stores are visible.

### Example 9-15. String Operations Are not Reordered with Earlier Stores

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov (_z), $1</td>
<td>mov (r1, _y)</td>
</tr>
<tr>
<td>rep stosl (_x)</td>
<td>mov (r2, _z)</td>
</tr>
</tbody>
</table>

Initially on processor 0: \(EAX = 1, ECX=128, ES:EDI = \_x\)

Initially \(\_y = \_z = 0, \_x \text{ to } 511\) \(\_x = \_y < \_x + 512, \_z\) is a separate memory location

\(r1 = 1\) and \(r2 = 0\) is not allowed

### 9.2.5 Strengthening or Weakening the Memory-Ordering Model

The Intel 64 and IA-32 architectures provide several mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations. These mechanisms include:

- The I/O instructions, locked instructions, the LOCK prefix, and serializing instructions force stronger ordering on the processor.
- The SFENCE instruction (introduced to the IA-32 architecture in the Pentium III processor) and the LFENCE and MFENCE instructions (introduced in the Pentium 4 processor) provide memory-ordering and serialization capabilities for specific types of memory operations.
- The memory type range registers (MTRRs) can be used to strengthen or weaken memory ordering for specific area of physical memory (see Section 12.11, “Memory Type Range Registers (MTRRs)”). MTRRs are available only in the Pentium 4, Intel Xeon, and P6 family processors.
- The page attribute table (PAT) can be used to strengthen memory ordering for a specific page or group of pages (see Section 12.12, “Page Attribute Table (PAT)”). The PAT is available only in the Pentium 4, Intel Xeon, and Pentium III processors.

These mechanisms can be used as follows:

Memory mapped devices and other I/O devices on the bus are often sensitive to the order of writes to their I/O buffers. I/O instructions can be used to (the IN and OUT instructions) impose strong write ordering on such accesses as follows. Prior to executing an I/O instruction, the processor waits for all previous instructions in the program to complete and for all buffered writes to drain to memory. Only instruction fetch and page tables walks can pass I/O instructions. Execution of subsequent instructions do not begin until the processor determines that the I/O instruction has been completed.

Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locked instruction such as the XCHG instruction or the LOCK prefix to ensure that a read-modify-write operation on memory is carried out atomically. Locked instructions typically operate like I/O instructions in that they wait for all previous memory accesses to complete and for all buffered writes to drain to memory (see Section 9.1.2, “Bus Locking”). Unlike I/O operations, locked instructions do not wait for all previous instructions to complete execution.

Program synchronization can also be carried out with serializing instructions (see Section 9.3). These instructions are typically used at critical procedure or task boundaries to force completion of all previous instructions before a jump to a new section of code or a context switch occurs. Like the I/O instructions, the processor waits until all previous instructions have been completed and all buffered writes have been drained to memory before executing the serializing instruction.
The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume that data. The functions of these instructions are as follows:

- **SFENCE** — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- **LFENCE** — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.\(^4\)
- **MFENCE** — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Note that the SFENCE, LFENCE, and MFENCE instructions provide a more efficient method of controlling memory ordering than the CPUID instruction.

The MTRRs were introduced in the P6 family processors to define the cache characteristics for specified areas of physical memory. The following are two examples of how memory types set up with MTRRs can be used to strengthen or weaken memory ordering for the Pentium 4, Intel Xeon, and P6 family processors:

- The strong uncached (UC) memory type forces a strong-ordering model on memory accesses. Here, all reads and writes to the UC memory region appear on the bus and out-of-order or speculative accesses are not performed. This memory type can be applied to an address range dedicated to memory mapped I/O devices to force strong memory ordering.
- For areas of memory where weak ordering is acceptable, the write back (WB) memory type can be chosen. Here, reads can be performed speculatively and writes can be buffered and combined. For this type of memory, cache locking is performed on atomic (locked) operations that do not split across cache lines, which helps to reduce the performance penalty associated with the use of the typical synchronization instructions, such as XCHG, that lock the bus during the entire read-modify-write operation. With the WB memory type, the XCHG instruction locks the cache instead of the bus if the memory access is contained within a cache line.

The PAT was introduced in the Pentium III processor to enhance the caching characteristics that can be assigned to pages or groups of pages. The PAT mechanism typically used to strengthen caching characteristics at the page level with respect to the caching characteristics established by the MTRRs. Table 12-7 shows the interaction of the PAT with the MTRRs.

Intel recommends that software written to run on Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors assume the processor-ordering model or a weaker memory-ordering model. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors do not implement a strong memory-ordering model, except when using the UC memory type. Despite the fact that Pentium 4, Intel Xeon, and P6 family processors support processor ordering, Intel does not guarantee that future processors will support this model. To make software portable to future processors, it is recommended that operating systems provide critical region and resource control constructs and API’s (application program interfaces) based on I/O, locking, and/or serializing instructions be used to synchronize access to shared areas of memory in multiple-processor systems. Also, software should not depend on processor ordering in situations where the system hardware does not support this memory-ordering model.

### 9.3 Serializing Instructions

The Intel 64 and IA-32 architectures define several **serializing instructions**. These instructions force the processor to complete all modifications to flags, registers, and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed. For example, when a MOV to control register instruction is used to load a new value into control register CR0 to enable protected mode, the processor must perform a serializing operation before it enters protected mode. This serializing operation ensures

---

4. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. As a result, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. An LFENCE that follows an instruction that stores to memory might complete before the data being stored have become globally visible. Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.
that all operations that were started while the processor was in real-address mode are completed before the switch to protected mode is made.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution.

It is important to note that executing of serializing instructions on P6 and more recent processor families constrain speculative execution because the results of speculatively executed instructions are discarded. The following instructions are serializing instructions:

- **Privileged serializing instructions** — INVD, INVEPT, INVLPG, INVPID, LGDT, LIDT, LLDT, LTR, MOV (to control register, with the exception of MOV CR8\(^5\)), MOV (to debug register), WBINVD, and WRMSR\(^6\).
- **Non-privileged serializing instructions** — CPUID, IRET, RSM, and SERIALIZE.

When the processor serializes instruction execution, it ensures that all pending memory transactions are completed (including writes stored in its store buffer) before it executes the next instruction. Nothing can pass a serializing instruction and a serializing instruction cannot pass any other instruction (read, write, instruction fetch, or I/O). For example, CPUID can be executed at any privilege level to serialize instruction execution with no effect on program flow, except that the EAX, EBX, ECX, and EDX registers are modified.

The following instructions are memory-ordering instructions, not serializing instructions. These drain the data memory subsystem. They do not serialize the instruction execution stream:\(^7\)

- **Non-privileged memory-ordering instructions** — SFENCE, LFENCE, and MFENCE.

The SFENCE, LFENCE, and MFENCE instructions provide more granularity in controlling the serialization of memory loads and stores (see Section 9.2.5, “Strengthening or Weakening the Memory-Ordering Model”).

The following additional information is worth noting regarding serializing instructions:

- The processor does not write back the contents of modified data in its data cache to external memory when it serializes instruction execution. Software can force modified data to be written back by executing the WBINVD instruction, which is a serializing instruction. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- When an instruction is executed that enables or disables paging (that is, changes the PG flag in control register CR0), the instruction should be followed by a jump instruction. The target instruction of the jump instruction is fetched with the new setting of the PG flag (that is, paging is enabled or disabled), but the jump instruction itself is fetched with the previous setting. The Pentium 4, Intel Xeon, and P6 family processors do not require the jump operation following the move to register CR0 (because any use of the MOV instruction in a Pentium 4, Intel Xeon, or P6 family processor to write to CR0 is completely serializing). However, to maintain backwards and forward compatibility with code written to run on other IA-32 processors, it is recommended that the jump operation be performed.
- Whenever an instruction is executed to change the contents of CR3 while paging is enabled, the next instruction is fetched using the translation tables that correspond to the new value of CR3. Therefore the next instruction and the sequentially following instructions should have a mapping based upon the new value of CR3. (Global entries in the TLBs are not invalidated, see Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches.”)
- The Pentium processor and more recent processor families use branch-prediction techniques to improve performance by prefetching the destination of a branch instruction before the branch instruction is executed. Consequently, instruction execution is not deterministically serialized when a branch instruction is executed.

---

5. MOV CR8 is not defined architecturally as a serializing instruction.
6. An execution of WRMSR to any non-serializing MSR is not serializing. Non-serializing MSRs include the following: IA32_SPEC_CTRL MSR (MSR index 48H), IA32_PRED_CMD MSR (MSR index 49H), IA32_TSX_CTRL MSR (MSR index 122H), IA32_TSC_DEADLINE MSR (MSR index 6E0H), IA32_PKRS MSR (MSR index 6E1H), IA32_HWP_REQUEST MSR (MSR index 774H), or any of the x2APIC MSRs (MSR indices 802H to 83FH).
7. LFENCE does provide some guarantees on instruction ordering. It does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes.
9.4 MULTIPLE-PROCESSOR (MP) INITIALIZATION

The IA-32 architecture (beginning with the P6 family processors) defines a multiple-processor (MP) initialization protocol called the Multiprocessor Specification Version 1.4. This specification defines the boot protocol to be used by IA-32 processors in multiple-processor systems. (Here, multiple processors is defined as two or more processors.) The MP initialization protocol has the following important features:

- It supports controlled booting of multiple processors without requiring dedicated system hardware.
- It allows hardware to initiate the booting of a system without the need for a dedicated signal or a predefined boot processor.
- It allows all IA-32 processors to be booted in the same manner, including those supporting Intel Hyper-Threading Technology.
- The MP initialization protocol also applies to MP systems using Intel 64 processors.

The mechanism for carrying out the MP initialization protocol differs depending on the Intel processor generations. The following bullets summarize the evolution of the changes:

- **For P6 family or older processors supporting MP operations**— The selection of the BSP and APs (see Section 9.4.1, “BSP and AP Processors”) is handled through arbitration on the APIC bus, using BIPI and FIPI messages. These processor generations have CPUID signatures of (family=06H, extended_model=0, model<=0DH), or family <06H. See Section 9.11.1, “Overview of the MP Initialization Process for P6 Family Processors,” for a complete discussion of MP initialization for P6 family processors.

- **Early generations of IA processors with family 0FH** — The selection of the BSP and APs (see Section 9.4.1, “BSP and AP Processors”) is handled through arbitration on the system bus, using BIPI and FIPI messages (see Section 9.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=0FH, model=0H, stepping<=09H.

- **Later generations of IA processors with family 0FH, and IA processors with system bus** — The selection of the BSP and APs is handled through a special system bus cycle, without using BIPI and FIPI message arbitration (see Section 9.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=0FH with (model=0H, stepping>=0AH) or (model >0, all steppings); or family=06H, extended_model=0, model>=0EH.

- **All other modern IA processor generations supporting MP operations**— The selection of the BSP and APs in the system is handled by platform-specific arrangement of the combination of hardware, BIOS, and/or configuration input options. The basis of the selection mechanism is similar to those of the Later generations of family 0FH and other Intel processor using system bus (see Section 9.4.3, “MP Initialization Protocol Algorithm for MP Systems”). These processor generations have CPUID signatures of family=06H, extended_model=0.

The family, model, and stepping ID for a processor is given in the EAX register when the CPUID instruction is executed with a value of 1 in the EAX register.

9.4.1 BSP and AP Processors

The MP initialization protocol defines two classes of processors: the bootstrap processor (BSP) and the application processors (APs). Following a power-up or RESET of an MP system, system hardware dynamically selects one of the processors on the system bus as the BSP. The remaining processors are designated as APs.

As part of the BSP selection mechanism, the BSP flag is set in the IA32_APIC_BASE MSR (see Figure 11-5) of the BSP, indicating that it is the BSP. This flag is cleared for all other processors.

The BSP executes the BIOS’s boot-strap code to configure the APIC environment, sets up system-wide data structures, and starts and initializes the APs. When the BSP and APs are initialized, the BSP then begins executing the operating-system initialization code.

Following a power-up or reset, the APs complete a minimal self-configuration, then wait for a startup signal (a SIPI message) from the BSP processor. Upon receiving a SIPI message, an AP executes the BIOS AP configuration code, which ends with the AP being placed in halt state.

For Intel 64 and IA-32 processors supporting Intel Hyper-Threading Technology, the MP initialization protocol treats each of the logical processors on the system bus or coherent link domain as a separate processor (with a unique
MULTIPLE-PROCESSOR MANAGEMENT

APIC ID). During boot-up, one of the logical processors is selected as the BSP and the remainder of the logical processors are designated as APs.

9.4.2 MP Initialization Protocol Requirements and Restrictions

The MP initialization protocol imposes the following requirements and restrictions on the system:

- The MP protocol is executed only after a power-up or RESET. If the MP protocol has completed and a BSP is chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each logical processor examines its BSP flag (in the IA32_APIC_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

9.4.3 MP Initialization Protocol Algorithm for MP Systems

Following a power-up or RESET of an MP system, the processors in the system execute the MP initialization protocol algorithm to initialize each of the logical processors on the system bus or coherent link domain. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each logical processor is assigned a unique APIC ID, based on system topology. The unique ID is a 32-bit value if the processor supports CPUID leaf 0BH, otherwise the unique ID is an 8-bit value. (see Section 9.4.5, “Identifying Logical Processors in an MP System”).
2. Each logical processor is assigned a unique arbitration priority based on its APIC ID.
3. Each logical processor executes its internal BIST simultaneously with the other logical processors in the system.
4. Upon completion of the BIST, the logical processors use a hardware-defined selection mechanism to select the BSP and the APs from the available logical processors on the system bus. The BSP selection mechanism differs depending on the family, model, and stepping IDs of the processors, as follows:
   - Later generations of IA processors within family 0FH (see Section 9.4), IA processors with system bus (family=06H, extended_model=0, model>=0EH), or all other modern Intel processors (family=06H, extended_model>0):
     - The logical processors begin monitoring the BNR# signal, which is toggling. When the BNR# pin stops toggling, each processor attempts to issue a NOP special cycle on the system bus.
     - The logical processor with the highest arbitration priority succeeds in issuing a NOP special cycle and is nominated the BSP. This processor sets the BSP flag in its IA32_APIC_BASE MSR, then fetches and begins executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
     - The remaining logical processors (that failed in issuing a NOP special cycle) are designated as APs. They leave their BSP flags in the clear state and enter a “wait-for-SIPI state.”
   - Early generations of IA processors within family 0FH (family=0FH, model=0H, stepping<=09H), P6 family or older processors supporting MP operations (family=06H, extended_model=0, model<=0DH; or family <06H):
     - Each processor broadcasts a BIPI to “all including self.” The first processor that broadcasts a BIPI (and thus receives its own BIPI vector), selects itself as the BSP and sets the BSP flag in its IA32_APIC_BASE MSR. (See Section 9.11.1, “Overview of the MP Initialization Process for P6 Family Processors,” for a description of the BIPI, FIPI, and SIPI messages.)
     - The remainder of the processors (which were not selected as the BSP) are designated as APs. They leave their BSP flags in the clear state and enter a “wait-for-SIPI state.”
     - The newly established BSP broadcasts an FIPI message to “all including self,” which the BSP and APs treat as an end of MP initialization signal. Only the processor with its BSP flag set responds to the FIPI
message. It responds by fetching and executing the BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).

5. As part of the boot-strap code, the BSP creates an ACPI table and/or an MP table and adds its initial APIC ID to these tables as appropriate.

6. At the end of the boot-strap procedure, the BSP sets a processor counter to 1, then broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000VV000H, where VV is the vector contained in the SIPI message).

7. The first action of the AP initialization code is to set up a race (among the APs) to a BIOS initialization semaphore. The first AP to the semaphore begins executing the initialization code. (See Section 9.4.4, "MP Initialization Example," for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and/or MP tables as appropriate and increments the processor counter by 1. At the completion of the initialization procedure, the AP executes a CLI instruction and halts itself.

8. When each of the APs has gained access to the semaphore and executed the AP initialization code, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.

9. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

The following section gives an example (with code) of the MP initialization protocol for of multiple processors operating in an MP configuration.

Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, describes how to program the LINT[0:1] pins of the processor’s local APICs after an MP configuration has been completed.

9.4.4 MP Initialization Example

The following example illustrates the use of the MP initialization protocol used to initialize processors in an MP system after the BSP and APs have been established. The code runs on Intel 64 or IA-32 processors that use a protocol. This includes P6 Family processors, Pentium 4 processors, Intel Core Duo, Intel Core 2 Duo and Intel Xeon processors.

The following constants and data definitions are used in the accompanying code examples. They are based on the addresses of the APIC registers defined in Table 11-1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICR_LOW</td>
<td>EQU 0FEE00300H</td>
</tr>
<tr>
<td>SVR</td>
<td>EQU 0FEE000F0H</td>
</tr>
<tr>
<td>APIC_ID</td>
<td>EQU 0FEE00020H</td>
</tr>
<tr>
<td>LVT3</td>
<td>EQU 0FEE00370H</td>
</tr>
<tr>
<td>APIC_ENABLED</td>
<td>EQU 0100H</td>
</tr>
<tr>
<td>BOOT_ID</td>
<td>DD ?</td>
</tr>
<tr>
<td>COUNT</td>
<td>EQU 00H</td>
</tr>
<tr>
<td>VACANT</td>
<td>EQU 00H</td>
</tr>
</tbody>
</table>

9.4.4.1 Typical BSP Initialization Sequence

After the BSP and APs have been selected (by means of a hardware protocol, see Section 9.4.3, "MP Initialization Protocol Algorithm for MP Systems"), the BSP begins executing BIOS boot-strap code (POST) at the normal IA-32 architecture starting address (FFFF FFF0H). The boot-strap code typically performs the following operations:

1. Initializes memory.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs.
4. Enables the caches.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the BSP is “GenuineIntel.”

6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.

7. Loads start-up code for the AP to execute into a 4-KByte page in the lower 1 MByte of memory.

8. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.

9. Determine the BSP’s APIC ID from the local APIC ID register (default is 0), the code snippet below is an example that applies to logical processors in a system whose local APIC units operate in xAPIC mode that APIC registers are accessed using memory mapped interface:

   MOV ESI, APIC_ID; Address of local APIC ID register  
   MOV EAX, [ESI];  
   AND EAX, 0FF000000H; Zero out all other bits except APIC ID  
   MOV BOOT_ID, EAX; Save in memory  

   Saves the APIC ID in the ACPI and/or MP tables and optionally in the system configuration space in RAM.

10. Converts the base address of the 4-KByte page for the AP’s bootup code into 8-bit vector. The 8-bit vector defines the address of a 4-KByte page in the real-address mode address space (1-MByte space). For example, a vector of 0BDH specifies a start-up memory address of 000BD000H.

11. Enables the local APIC by setting bit 8 of the APIC spurious vector register (SVR).

   MOV ESI, SVR; Address of SVR  
   MOV EAX, [ESI];  
   OR  EAX, APIC_ENABLED; Set bit 8 to enable (0 on reset)  
   MOV [ESI], EAX;  

12. Sets up the LVT error handling entry by establishing an 8-bit vector for the APIC error handler.

   MOV ESI, LVT3;  
   MOV EAX, [ESI];  
   AND EAX, FFFFFF00H; Clear out previous vector.  
   OR EAX, 000000xxH; xx is the 8-bit vector the APIC error handler.  
   MOV [ESI], EAX;  

13. Initializes the Lock Semaphore variable VACANT to 00H. The APs use this semaphore to determine the order in which they execute BIOS AP initialization code.

14. Performs the following operation to set up the BSP to detect the presence of APs in the system and the number of processors (within a finite duration, minimally 100 milliseconds):

   — Sets the value of the COUNT variable to 1.
   — In the AP BIOS initialization code, the AP will increment the COUNT variable to indicate its presence. The finite duration while waiting for the COUNT to be updated can be accomplished with a timer. When the timer expires, the BSP checks the value of the COUNT variable. If the timer expires and the COUNT variable has not been incremented, no APs are present or some error has occurred.

15. Broadcasts an INIT-SIPI-SIPI IPI sequence to the APs to wake them up and initialize them. Alternatively, following a power-up or RESET, since all APs are already in the “wait-for-SIPI state,” the BSP can broadcast just a single SIPI IPI to the APs to wake them up and initialize them. If software knows how many logical processors it expects to wake up, it may choose to poll the COUNT variable. If the expected processors show up before the 100 millisecond timer expires, the timer can be canceled and skip to step 16.

   The left-hand-side of the procedure illustrated in Table 9-1 provides an algorithm when the expected processor count is unknown. The right-hand-side of Table 9-1 can be used when the expected processor count is known.
9.4.4.2  Typical AP Initialization Sequence

When an AP receives the SIPI, it begins executing BIOS AP initialization code at the vector encoded in the SIPI. The AP initialization code typically performs the following operations:

1. Waits on the BIOS initialization Lock Semaphore. When control of the semaphore is attained, initialization continues.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs (using the same mapping that was used for the BSP).
4. Enables the cache.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the AP is “GenuineIntel.”
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
8. Determines the AP’s APIC ID from the local APIC ID register, and adds it to the MP and ACPI tables and optionally to the system configuration space in RAM.
9. Initializes and configures the local APIC by setting bit 8 in the SVR register and setting up the LVT3 (error LVT) for error handling (as described in steps 9 and 10 in Section 9.4.4.1, “Typical BSP Initialization Sequence”).
10. Configures the APs SMI execution environment. (Each AP and the BSP must have a different SMBASE address.)
11. Increments the COUNT variable by 1.
12. Releases the semaphore.
13. Executes one of the following:

<table>
<thead>
<tr>
<th>INIT-SIPI-SIPI when the expected processor count is unknown</th>
<th>INIT-SIPI-SIPI when the expected processor count is known</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ESI, ICR_LOW; Load address of ICR low dword into ESI.</td>
<td>MOV ESI, ICR_LOW; Load address of ICR low dword into ESI.</td>
</tr>
<tr>
<td>MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI to all APs into EAX.</td>
<td>MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI to all APs into EAX.</td>
</tr>
<tr>
<td>MOV [ESI], EAX; Broadcast INIT IPI to all APs; 10-millisecond delay loop.</td>
<td>MOV [ESI], EAX; Broadcast INIT IPI to all APs; 10-millisecond delay loop.</td>
</tr>
<tr>
<td>MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP to all APs into EAX, where xx is the vector computed in step 10.</td>
<td>MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP to all APs into EAX, where xx is the vector computed in step 10.</td>
</tr>
<tr>
<td>MOV [ESI], EAX; Broadcast SIPI IPI to all APs; 200-microsecond delay loop</td>
<td>MOV [ESI], EAX; Broadcast SIPI IPI to all APs; 200-microsecond delay loop with check to see if COUNT has reached the expected processor count. If COUNT reaches expected processor count, cancel timer and go to step 16.</td>
</tr>
<tr>
<td>; Waits for the timer interrupt until the timer expires</td>
<td>; Wait for the timer interrupt polling COUNT. If COUNT reaches expected processor count, cancel timer and go to step 16.</td>
</tr>
<tr>
<td>; If timer expires, go to step 16.</td>
<td>; If timer expires, go to step 16.</td>
</tr>
</tbody>
</table>

16. Reads and evaluates the COUNT variable and establishes a processor count.
17. If necessary, reconfigures the APIC and continues with the remaining system diagnostics as appropriate.
MULTIPLE-PROCESSOR MANAGEMENT

— the CLI and HLT instructions (if MONITOR/MWAIT is not supported), or
— the CLI, MONITOR, and MWAIT sequence to enter a deep C-state.

14. Waits for an INIT IPI.

9.4.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

• **Read APIC ID for a local APIC** — Code running on a logical processor can read APIC ID in one of two ways depending on the local APIC unit is operating in x2APIC mode or in xAPIC mode:
  
  — If the local APIC unit supports x2APIC and is operating in x2APIC mode, 32-bit APIC ID can be read by executing a RDMSR instruction to read the processor’s x2APIC ID register. This method is equivalent to executing CPUID leaf 0BH described below.
  
  — If the local APIC unit is operating in xAPIC mode, 8-bit APIC ID can be read by executing a MOV instruction to read the processor’s local APIC ID register (see Section 11.4.6, “Local APIC ID”). This is the ID to use for directing physical destination mode interrupts to the processor.

• **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.

• **Read Initial APIC ID** (If the processor does not support CPUID leaf 0BH) — An APIC ID is assigned to a logical processor during power up. This is the initial APIC ID reported by CPUID.1:EBX[31:24] and may be different from the current value read from the local APIC. The initial APIC ID can be used to determine the topological relationship between logical processors for multi-processor systems that do not support CPUID leaf 0BH.

  Bits in the 8-bit initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical domain of the multi-threading resource topology in an MP system (See Section 9.9.1, “Hierarchical Mapping of Shared Resources”). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.

• **Read 32-bit APIC ID from CPUID leaf 0BH** (If the processor supports CPUID leaf 0BH) — A unique APIC ID is assigned to a logical processor during power up. This APIC ID is reported by CPUID.0BH:EDX[31:0] as a 32-bit value. Use the 32-bit APIC ID and CPUID leaf 0BH to determine the topological relationship between logical processors if the processor supports CPUID leaf 0BH.

  Bits in the 32-bit x2APIC ID can be extracted into sub-fields using CPUID leaf 0BH parameters. (See Section 9.9.1, “Hierarchical Mapping of Shared Resources”).

Figure 9-2 shows two examples of APIC ID bit fields in earlier single-core processors. In single-core Intel Xeon processors, the APIC ID assigned to a logical processor during power-up and initialization is 8 bits. Bits 2:1 form a 2-bit physical package identifier (which can also be thought of as a socket identifier). In systems that configure physical processors in clusters, bits 4:3 form a 2-bit cluster ID. Bit 0 is used in the Intel Xeon processor MP to identify the two logical processors within the package (see Section 9.9.3, “Hierarchical ID of Logical Processors in an MP System”). For Intel Xeon processors that do not support Intel Hyper-Threading Technology, bit 0 is always set to 0; for Intel Xeon processors supporting Intel Hyper-Threading Technology, bit 0 performs the same function as it does for Intel Xeon processor MP.

For more recent multi-core processors, see Section 9.9.1, “Hierarchical Mapping of Shared Resources,” for a complete description of the topological relationships between logical processors and bit field locations within an initial APIC ID across Intel 64 and IA-32 processor families.

Note the number of bit fields and the width of bit-fields are dependent on processor and platform hardware capabilities. Software should determine these at runtime. When initial APIC IDs are assigned to logical processors, the value of APIC ID assigned to a logical processor will respect the bit-field boundaries corresponding core, physical package, etc. Additional examples of the bit fields in the initial APIC ID of multi-threading capable systems are shown in Section 9.9.
For P6 family processors, the APIC ID that is assigned to a processor during power-up and initialization is 4 bits (see Figure 9-2). Here, bits 0 and 1 form a 2-bit processor (or socket) identifier and bits 2 and 3 form a 2-bit cluster ID.

**9.5 INTEL® HYPER-THREADING TECHNOLOGY AND INTEL® MULTI-CORE TECHNOLOGY**

Intel Hyper-Threading Technology and Intel multi-core technology are extensions to Intel 64 and IA-32 architectures that enable a single physical processor to execute two or more separate code streams (called *threads*) concurrently. In Intel Hyper-Threading Technology, a single processor core provides two logical processors that share execution resources (see Section 9.7, "Intel® Hyper-Threading Technology Architecture"). In Intel multi-core technology, a physical processor package provides two or more processor cores. Both configurations require chipsets and a BIOS that support the technologies.

Software should not rely on processor names to determine whether a processor supports Intel Hyper-Threading Technology or Intel multi-core technology. Use the CPUID instruction to determine processor capability (see Section 9.6.2, "Initializing Multi-Core Processors").

**9.6 DETECTING HARDWARE MULTI-THREADING SUPPORT AND TOPOLOGY**

Use the CPUID instruction to detect the presence of hardware multi-threading support in a physical processor. Hardware multi-threading can support several varieties of multigrade and/or Intel Hyper-Threading Technology. CPUID instruction provides several sets of parameter information to aid software enumerating topology information. The relevant topology enumeration parameters provided by CPUID include:

- **Hardware Multi-Thread feature flag (CPUID.1:EDX[28] = 1)** — Indicates when set that the physical package is capable of supporting Intel Hyper-Threading Technology and/or multiple cores.
- **Processor topology enumeration parameters for 8-bit APIC ID**:
  - **Addressable IDs for Logical processors in the same Package (CPUID.1:EBX[23:16])** — Indicates the maximum number of addressable ID for logical processors in a physical package. Within a physical package, there may be addressable IDs that are not occupied by any logical processors. This parameter does not represents the hardware capability of the physical processor.
• **Addressable IDs for processor cores in the same Package**\(^9\) (**CPUID.(EAX=4, ECX=0):EAX[31:26] + 1 = Y**) — Indicates the maximum number of addressable IDs attributable to processor cores (Y) in the physical package.

• **Extended Processor Topology Enumeration parameters for 32-bit APIC ID** — Intel 64 processors supporting CPUID leaf 0BH will assign unique APIC IDs to each logical processor in the system. CPUID leaf 0BH reports the 32-bit APIC ID and provide topology enumeration parameters. See CPUID instruction reference pages in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

The CPUID feature flag may indicate support for hardware multi-threading when only one logical processor available in the package. In this case, the decimal value represented by bits 16 through 23 in the EBX register will have a value of 1.

Software should note that the number of logical processors enabled by system software may be less than the value of "Addressable IDs for Logical processors". Similarly, the number of cores enabled by system software may be less than the value of "Addressable IDs for processor cores".

Software can detect the availability of the CPUID extended topology enumeration leaf (0BH) by performing two steps:

• Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal or 11 (0BH), then proceed to next step,

• Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. Note the presence of CPUID leaf 0BH in a processor does not guarantee support that the local APIC supports x2APIC. If CPUID.(EAX=0BH, ECX=0H):EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

### 9.6.1 Initializing Processors Supporting Intel® Hyper-Threading Technology

The initialization process for an MP system that contains processors supporting Intel Hyper-Threading Technology is the same as for conventional MP systems (see Section 9.4, "Multiple-Processor (MP) Initialization"). One logical processor in the system is selected as the BSP and other processors (or logical processors) are designated as APs. The initialization process is identical to that described in Section 9.4.3, "MP Initialization Protocol Algorithm for MP Systems," and Section 9.4.4, "MP Initialization Example."

During initialization, each logical processor is assigned an APIC ID that is stored in the local APIC ID register for each logical processor. If two or more processors supporting Intel Hyper-Threading Technology are present, each logical processor on the system bus is assigned a unique ID (see Section 9.9.3, "Hierarchical ID of Logical Processors in an MP System"). Once logical processors have APIC IDs, software communicates with them by sending APIC IPI messages.

### 9.6.2 Initializing Multi-Core Processors

The initialization process for an MP system that contains multi-core Intel 64 or IA-32 processors is the same as for conventional MP systems (see Section 9.4, "Multiple-Processor (MP) Initialization"). A logical processor in one core is selected as the BSP; other logical processors are designated as APs.

During initialization, each logical processor is assigned an APIC ID. Once logical processors have APIC IDs, software may communicate with them by sending APIC IPI messages.

8. Operating system and BIOS may implement features that reduce the number of logical processors available in a platform to applications at runtime to less than the number of physical packages times the number of hardware-capable logical processors per package.

9. Software must check CPUID for its support of leaf 4 when implementing support for multi-core. If CPUID leaf 4 is not available at runtime, software should handle the situation as if there is only one core per package.

10. Maximum number of cores in the physical package must be queried by executing CPUID with EAX=4 and a valid ECX input value. Valid ECX input values start from 0.
9.6.3 Executing Multiple Threads on an Intel® 64 or IA-32 Processor Supporting Hardware Multi-Threading

Upon completing the operating system boot-up procedure, the bootstrap processor (BSP) executes operating system code. Other logical processors are placed in the halt state. To execute a code stream (thread) on a halted logical processor, the operating system issues an interprocessor interrupt (IPI) addressed to the halted logical processor. In response to the IPI, the processor wakes up and begins executing the code identified by the vector received as part of the IPI.

To manage execution of multiple threads on logical processors, an operating system can use conventional symmetric multiprocessing (SMP) techniques. For example, the operating system can use a time-slice or load balancing mechanism to periodically interrupt each of the active logical processors. Upon interrupting a logical processor, the operating system checks its run queue for a thread waiting to be executed and dispatches the thread to the interrupted logical processor.

9.6.4 Handling Interrupts on an IA-32 Processor Supporting Hardware Multi-Threading

Interrupts are handled on processors supporting Intel Hyper-Threading Technology as they are on conventional MP systems. External interrupts are received by the I/O APIC, which distributes them as interrupt messages to specific logical processors (see Figure 9-3).

Logical processors can also send IPIs to other logical processors by writing to the ICR register of its local APIC (see Section 11.6, “Issuing Interprocessor Interrupts”). This also applies to dual-core processors.

![Figure 9-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology](image-url)

9.7 INTEL® HYPER-THREADING TECHNOLOGY ARCHITECTURE

Figure 9-4 shows a generalized view of an Intel processor supporting Intel Hyper-Threading Technology, using the original Intel Xeon processor MP as an example. This implementation of the Intel Hyper-Threading Technology
MULTIPLE-PROCESSOR MANAGEMENT

consists of two logical processors (each represented by a separate architectural state) which share the processor’s execution engine and the bus interface. Each logical processor also has its own advanced programmable interrupt controller (APIC).

![Figure 9-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology](image)

9.7.1 State of the Logical Processors

The following features are part of the architectural state of logical processors within Intel 64 or IA-32 processors supporting Intel Hyper-Threading Technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)
- Segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS and EIP registers. Note that the CS and EIP/RIP registers for each logical processor point to the instruction stream for the thread being executed by the logical processor.
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- Control registers and system table pointer registers (GDTR, LDTR, IDTR, task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and the debug control MSRs
- Machine check global status (IA32_MCG_STATUS) and machine check capability (IA32_MCG_CAP) MSRs
- Thermal clock modulation and ACPI Power management control MSRs
- Time stamp counter MSRs
- Most of the other MSR registers, including the page attribute table (PAT). See the exceptions below.
- Local APIC registers.
- Additional general purpose registers (R8-R15), XMM registers (XMM8-XMM15), control register, IA32_EFER on Intel 64 processors.

The following features are shared by logical processors:
9.7.2 APIC Functionality

When a processor supporting Intel Hyper-Threading Technology support is initialized, each logical processor is assigned a local APIC ID (see Table 11-1). The local APIC ID serves as an ID for the logical processor and is stored in the logical processor’s APIC ID register. If two or more processors supporting Intel Hyper-Threading Technology are present in a dual processor (DP) or MP system, each logical processor on the system bus is assigned a unique local APIC ID (see Section 9.9.3, “Hierarchical ID of Logical Processors in an MP System”).

Software communicates with local processors using the APIC’s interprocessor interrupt (IPI) messaging facility. Setup and programming for APICs is identical in processors that support and do not support Intel Hyper-Threading Technology. See Chapter 11, “Advanced Programmable Interrupt Controller (APIC),” for a detailed discussion.

9.7.3 Memory Type Range Registers (MTRR)

MTRRs in a processor supporting Intel Hyper-Threading Technology are shared by logical processors. When one logical processor updates the setting of the MTRRs, settings are automatically shared with the other logical processors in the same physical package.

The architectures require that all MP systems based on Intel 64 and IA-32 processors (this includes logical processors) must use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running. See Section 12.11, “Memory Type Range Registers (MTRRs),” for information on setting up MTRRs.

9.7.4 Page Attribute Table (PAT)

Each logical processor has its own PAT MSR (IA32_PAT). However, as described in Section 12.12, “Page Attribute Table (PAT),” the PAT MSR settings must be the same for all processors in a system, including the logical processors.

9.7.5 Machine Check Architecture

In the Intel HT Technology context as implemented by processors based on Intel NetBurst® microarchitecture, all of the machine check architecture (MCA) MSRs (except for the IA32_MCG_STATUS and IA32_MCG_CAP MSRs) are duplicated for each logical processor. This permits logical processors to initialize, configure, query, and handle machine-check exceptions simultaneously within the same physical processor. The design is compatible with machine check exception handlers that follow the guidelines given in Chapter 16, “Machine-Check Architecture.”

The IA32_MCG_STATUS MSR is duplicated for each logical processor so that its machine check in progress bit field (MCIP) can be used to detect recursion on the part of MCA handlers. In addition, the MSR allows each logical processor to determine that a machine-check exception is in progress independent of the actions of another logical processor in the same physical package.

Because the logical processors within a physical package are tightly coupled with respect to shared hardware resources, both logical processors are notified of machine check errors that occur within a given physical processor. If machine-check exceptions are enabled when a fatal error is reported, all the logical processors within a physical package are dispatched to the machine-check exception handler. If machine-check exceptions are disabled, the logical processors enter the shutdown state and assert the IERR# signal.

When enabling machine-check exceptions, the MCE flag in control register CR4 should be set for each logical processor.
On Intel Atom family processors that support Intel Hyper-Threading Technology, the MCA facilities are shared between all logical processors on the same processor core.

### 9.7.6 Debug Registers and Extensions

Each logical processor has its own set of debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and its own debug control MSR. These can be set to control and record debug information for each logical processor independently. Each logical processor also has its own last branch records (LBR) stack.

### 9.7.7 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between the logical processors within a processor core for processors based on Intel NetBurst microarchitecture. As a result, software must manage the use of these resources. The performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 20.6.4, “Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture,” for a discussion of performance monitoring in the Intel Xeon processor MP.

In Intel Atom processor family that support Intel Hyper-Threading Technology, the performance counters (general-purpose and fixed-function counters) and their companion control MSRs are duplicated for each logical processor.

### 9.7.8 IA32_MISC_ENABLE MSR

The IA32_MISC_ENABLE MSR (MSR address 1A0H) is generally shared between the logical processors in a processor core supporting Intel Hyper-Threading Technology. However, some bit fields within IA32_MISC_ENABLE MSR may be duplicated per logical processor. The partition of shared or duplicated bit fields within IA32_MISC_ENABLE is implementation dependent. Software should program duplicated fields carefully on all logical processors in the system to ensure consistent behavior.

### 9.7.9 Memory Ordering

The logical processors in an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology obey the same rules for memory ordering as Intel 64 or IA-32 processors without Intel HT Technology (see Section 9.2, “Memory Ordering”). Each logical processor uses a processor-ordered memory model that can be further defined as “write-ordered with store buffer forwarding.” All mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations apply to each logical processor.

### 9.7.10 Serializing Instructions

As a general rule, when a logical processor in a processor supporting Intel Hyper-Threading Technology executes a serializing instruction, only that logical processor is affected by the operation. An exception to this rule is the execution of the WBINVD, INVD, and WRMSR instructions; and the MOV CR instruction when the state of the CD flag in control register CR0 is modified. Here, both logical processors are serialized.

### 9.7.11 Microcode Update Resources

In an Intel processor supporting Intel Hyper-Threading Technology, the microcode update facilities are shared between the logical processors; either logical processor can initiate an update. Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information. If logical processors initiate an update simultaneously, the processor core provides the necessary synchronization needed to ensure that only one update is performed at a time.
NOTE
Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 10.11.6.3.

9.7.12 Self Modifying Code
Intel processors supporting Intel Hyper-Threading Technology support self-modifying code, where data writes modify instructions cached or currently in flight. They also support cross-modifying code, where on an MP system writes generated by one processor modify instructions cached or currently in flight on another. See Section 9.1.3, "Handling Self- and Cross-Modifying Code," for a description of the requirements for self- and cross-modifying code in an IA-32 processor.

9.7.13 Implementation-Specific Intel® HT Technology Facilities
The following non-architectural facilities are implementation-specific in IA-32 processors supporting Intel Hyper-Threading Technology:

- Caches.
- Translation lookaside buffers (TLBs).
- Thermal monitoring facilities.

The Intel Xeon processor MP implementation is described in the following sections.

9.7.13.1 Processor Caches
For processors supporting Intel Hyper-Threading Technology, the caches are shared. Any cache manipulation instruction that is executed on one logical processor has a global effect on the cache hierarchy of the physical processor. Note the following:

- **WBINVD instruction** — The entire cache hierarchy is invalidated after modified data is written back to memory. All logical processors are stopped from executing until after the write-back and invalidate operation is completed. A special bus cycle is sent to all caching agents. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.

- **INVD instruction** — The entire cache hierarchy is invalidated without writing back modified data to memory. All logical processors are stopped from executing until after the invalidate operation is completed. A special bus cycle is sent to all caching agents.

- **CLFLUSH and CLFLUSHOPT instructions** — The specified cache line is invalidated from the cache hierarchy after any modified data is written back to memory and a bus cycle is sent to all caching agents, regardless of which logical processor caused the cache line to be filled.

- **CD flag in control register CR0** — Each logical processor has its own CR0 control register, and thus its own CD flag in CR0. The CD flags for the two logical processors are ORed together, such that when any logical processor sets its CD flag, the entire cache is nominally disabled.

9.7.13.2 Processor Translation Lookaside Buffers (TLBs)
In processors supporting Intel Hyper-Threading Technology, data cache TLBs are shared. The instruction cache TLB may be duplicated or shared in each logical processor, depending on implementation specificities of different processor families.

Entries in the TLBs are tagged with an ID that indicates the logical processor that initiated the translation. This tag applies even for translations that are marked global using the page-global feature for memory paging. See Section 4.10, "Caching Translation Information," for information about global translations.
When a logical processor performs a TLB invalidation operation, only the TLB entries that are tagged for that logical processor are guaranteed to be flushed. This protocol applies to all TLB invalidation operations, including writes to control registers CR3 and CR4 and uses of the INVLPG instruction.

9.7.13.3 Thermal Monitor

In a processor that supports Intel Hyper-Threading Technology, logical processors share the catastrophic shutdown detector and the automatic thermal monitoring mechanism (see Section 15.8, “Thermal Monitoring and Protection”). Sharing results in the following behavior:

- If the processor’s core temperature rises above the preset catastrophic shutdown temperature, the processor core halts execution, which causes both logical processors to stop execution.
- When the processor’s core temperature rises above the preset automatic thermal monitor trip temperature, the frequency of the processor core is automatically modulated, which effects the execution speed of both logical processors.

For software controlled clock modulation, each logical processor has its own IA32_CLOCK_MODULATION MSR, allowing clock modulation to be enabled or disabled on a logical processor basis. Typically, if software controlled clock modulation is going to be used, the feature must be enabled for all the logical processors within a physical processor and the modulation duty cycle must be set to the same value for each logical processor. If the duty cycle values differ between the logical processors, the processor clock will be modulated at the highest duty cycle selected.

9.7.13.4 External Signal Compatibility

This section describes the constraints on external signals received through the pins of a processor supporting Intel Hyper-Threading Technology and how these signals are shared between its logical processors.

- **STPCLK#** — A single STPCLK# pin is provided on the physical package of the Intel Xeon processor MP. External control logic uses this pin for power management within the system. When the STPCLK# signal is asserted, the processor core transitions to the stop-grant state, where instruction execution is halted but the processor core continues to respond to snoop transactions. Regardless of whether the logical processors are active or halted when the STPCLK# signal is asserted, execution is stopped on both logical processors and neither will respond to interrupts.

In MP systems, the STPCLK# pins on all physical processors are generally tied together. As a result this signal affects all the logical processors within the system simultaneously.

- **LINT0 and LINT1 pins** — A processor supporting Intel Hyper-Threading Technology has only one set of LINT0 and LINT1 pins, which are shared between the logical processors. When one of these pins is asserted, both logical processors respond unless the pin has been masked in the APIC local vector tables for one or both of the logical processors.

Typically in MP systems, the LINT0 and LINT1 pins are not used to deliver interrupts to the logical processors. Instead all interrupts are delivered to the local processors through the I/O APIC.

- **A20M# pin** — On an IA-32 processor, the A20M# pin is typically provided for compatibility with the Intel 286 processor. Asserting this pin causes bit 20 of the physical address to be masked (forced to zero) for all external bus memory accesses. Processors supporting Intel Hyper-Threading Technology provide one A20M# pin, which affects the operation of both logical processors within the physical processor.

The functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

9.8 Multi-Core Architecture

This section describes the architecture of Intel 64 and IA-32 processors supporting dual-core and quad-core technology. The discussion is applicable to the Intel Pentium processor Extreme Edition, Pentium D, Intel Core Duo, Intel Core 2 Duo, Dual-core Intel Xeon processor, Intel Core 2 Quad processors, and quad-core Intel Xeon processors. Features vary across different microarchitectures and are detectable using CPUID.
In general, each processor core has dedicated microarchitectural resources identical to a single-processor implementation of the underlying microarchitecture without hardware multi-threading capability. Each logical processor in a dual-core processor (whether supporting Intel Hyper-Threading Technology or not) has its own APIC functionality, PAT, machine check architecture, debug registers and extensions. Each logical processor handles serialization instructions or self-modifying code on its own. Memory order is handled the same way as in Intel Hyper-Threading Technology.

The topology of the cache hierarchy (with respect to whether a given cache level is shared by one or more processor cores or by all logical processors in the physical package) depends on the processor implementation. Software must use the deterministic cache parameter leaf of CPUID instruction to discover the cache-sharing topology between the logical processors in a multi-threading environment.

9.8.1 Logical Processor Support

The topological composition of processor cores and logical processors in a multi-core processor can be discovered using CPUID. Within each processor core, one or more logical processors may be available.

System software must follow the requirement MP initialization sequences (see Section 9.4, “Multiple-Processor (MP) Initialization”) to recognize and enable logical processors. At runtime, software can enumerate those logical processors enabled by system software to identify the topological relationships between these logical processors. (See Section 9.9.5, "Identifying Topological Relationships in an MP System").

9.8.2 Memory Type Range Registers (MTRR)

MTRR is shared between two logical processors sharing a processor core if the physical processor supports Intel Hyper-Threading Technology. MTRR is not shared between logical processors located in different cores or different physical packages.

The Intel 64 and IA-32 architectures require that all logical processors in an MP system use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running.

See Section 12.11, “Memory Type Range Registers (MTRRs).”

9.8.3 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between two logical processors sharing a processor core if the processor core supports Intel Hyper-Threading Technology and is based on Intel NetBurst microarchitecture. They are not shared between logical processors in different cores or different physical packages. As a result, software must manage the use of these resources, based on the topology of performance monitoring resources. Performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 20.6.4, “Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture.”

9.8.4 IA32_MISC_ENABLE MSR

Some bit fields in IA32_MISC_ENABLE MSR (MSR address 1A0H) may be shared between two logical processors sharing a processor core, or may be shared between different cores in a physical processor. See Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

9.8.5 Microcode Update Resources

Microcode update facilities are shared between two logical processors sharing a processor core if the physical package supports Intel Hyper-Threading Technology. They are not shared between logical processors in different
cores or different physical packages. Either logical processor that has access to the microcode update facility can initiate an update.

Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information.

All microcode update steps during processor initialization should use the same update data on all cores in all physical packages of the same stepping. Any subsequent microcode update must apply consistent update data to all cores in all physical packages of the same stepping. If the processor detects an attempt to load an older microcode update when a newer microcode update had previously been loaded, it may reject the older update to stay with the newer update.

**NOTE**

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 10.11.6.3.

### 9.9 PROGRAMMING CONSIDERATIONS FOR HARDWARE MULTI-THREADING CAPABLE PROCESSORS

In a multi-threading environment, there may be certain hardware resources that are physically shared at some level of the hardware topology. In the multi-processor systems, typically bus and memory sub-systems are physically shared between multiple sockets. Within a hardware multi-threading capable processors, certain resources are provided for each processor core, while other resources may be provided for each logical processors (see Section 9.7, "Intel® Hyper-Threading Technology Architecture," and Section 9.8, "Multi-Core Architecture").

From a software programming perspective, control transfer of processor operation is managed at the granularity of logical processor (operating systems dispatch a runnable task by allocating an available logical processor on the platform). To manage the topology of shared resources in a multi-threading environment, it may be useful for software to understand and manage resources that are shared by more than one logical processors.

#### 9.9.1 Hierarchical Mapping of Shared Resources

The APIC_ID value associated with each logical processor in a multi-processor system is unique (see Section 9.6, "Detecting Hardware Multi-Threaded Support and Topology"). This 8-bit or 32-bit value can be decomposed into sub-fields, where each sub-field corresponds a hierarchical domain of the topological mapping of hardware resources.

The decomposition of an APIC_ID may consist of several sub fields representing the topology within a physical processor package, the higher-order bits of an APIC ID may also be used by cluster vendors to represent the topology of cluster nodes of each coherent multiprocessor systems:

- **Cluster** — Some multi-threading environments consists of multiple clusters of multi-processor systems. The CLUSTER_ID sub-field is usually supported by vendor firmware to distinguish different clusters. For non-clustered systems, CLUSTER_ID is usually 0 and system topology is reduced.

- **Package** — A physical processor package mates with a socket. A package may contain one or more software visible die. The PACKAGE_ID sub-field distinguishes different physical packages within a cluster.

- **Die** — A software-visible chip inside a package. The DIE_ID sub-field distinguishes different die within a package. If there are no software visible die, the width of this bit field is 0.

- **DieGrp** — A group of die that share certain resources.

- **Tile** — A set of cores that share certain resources. The TILE_ID sub-field distinguishes different tiles. If there are no software visible tiles, the width of this bit field is 0.
• **Module** — A set of cores that share certain resources. The MODULE_ID sub-field distinguishes different modules. If there are no software visible modules, the width of this bit field is 0.

• **Core** — Processor cores may be contained within modules, within tiles, on software-visible die, or appear directly at the package domain. The CORE_ID sub-field distinguishes processor cores. For a single-core processor, the width of this bit field is 0.

• **Logical Processor** — A processor core provides one or more logical processors sharing execution resources. The LOGICAL_PROCESSOR_ID sub-field distinguishes logical processors in a core. The width of this bit field is non-zero if a processor core provides more than one logical processors.

The LOGICAL_PROCESSOR_ID and CORE_ID sub-fields are bit-wise contiguous in the APIC_ID field (see Figure 9-5).

If the processor supports CPUID leaf 0BH and leaf 1FH, the 32-bit APIC ID can represent cluster plus several domains of topology within the physical processor package. The exact number of hierarchical domains within a physical processor package must be enumerated through CPUID leaf 0BH and leaf 1FH. Common processor families may employ a topology similar to that represented by the 8-bit Initial APIC ID. In general, CPUID leaf 0BH and leaf 1FH can support a topology enumeration algorithm that decompose a 32-bit APIC ID into more than four sub-fields (see Figure 9-6).

**NOTE**

CPUID leaf 0BH and leaf 1FH can have differences in the number of domain types reported (CPUID leaf 1FH defines additional domain types). If the processor supports CPUID leaf 1FH, usage of this leaf is preferred over leaf 0BH. CPUID leaf 0BH is available for legacy compatibility going forward.

The width of each sub-field depends on hardware and software configurations. Field widths can be determined at runtime using the algorithm discussed below (Example 9-16 through Example 9-21).

Figure 7-6 depicts the relationships of three of the hierarchical sub-fields in a hypothetical MP system. The value of valid APIC_IDs need not be contiguous across package boundary or core boundaries.
9.9.2 Hierarchical Mapping of CPUID Extended Topology Leaf

CPUID leaf 0BH and leaf 1FH provide enumeration parameters for software to identify each hierarchy of the processor topology in a deterministic manner. Each hierarchical domain of the topology starting from the Logical Processor domain is represented numerically by a sub-leaf index within the CPUID 0BH leaf and 1FH leaf. Each domain of the topology is mapped to a sub-field in the APIC ID, following the general relationship depicted in Figure 9-6. This mechanism allows software to query the exact number of domains within a physical processor package and the bit-width of each sub-field of x2APIC ID directly. For example,

- Starting from sub-leaf index 0 and incrementing ECX until CPUID.(EAX=0BH or 1FH, ECX=N):ECX[15:8] returns an invalid “domain type” encoding. The number of domains within the physical processor package is "N" (excluding PACKAGE). Using Figure 9-6 as an example, CPUID.(EAX=0BH or 1FH, ECX=4):ECX[15:8] will report 00H, indicating sub leaf 04H is invalid. This is also depicted by a pseudo code example:

Example 9-16. Number of Domains Below the Physical Processor Package

Word NumberOfDomainsBelowPackage = 0;
DWord Subleaf = 0;

EAX = 0BH or 1FH; // query each sub leaf of CPUID leaf 0BH or 1FH; CPUID leaf 1FH is preferred over leaf 0BH if available.
ECX = Subleaf;
CPUID;
while(EBX != 0) // Enumerate until EBX reports 0
{
    if(EAX[4:0] != 0) // A Shift Value of 0 indicates this domain does not exist.
        // (Such as no SMT_ID, which is required entry at sub-leaf 0.)
    {
        NumberOfDomainsBelowPackage++;
    }
    Subleaf++;
    EAX = 0BH or 1FH;
    ECX = Subleaf;
    CPUID;
}
// NumberOfDomainsBelowPackage contains the absolute number of domains that exist below package.
N = Subleaf; // Sub-leaf supplies the number of entries CPUID will return.
• Sub-leaf index 0 (ECX = 0 as input) provides enumeration parameters to extract the LOGICAL_PROCESSOR_ID sub-field of x2APIC ID. If EAX = 0BH or 1FH, and ECX = 0 is specified as input when executing CPUID, CPUID.(EAX=0BH or 1FH, ECX=0):EAX[4:0] reports a value (a right-shift count) that allow software to extract part of x2APIC ID to distinguish the next higher topological entities above the LOGICAL_PROCESSOR_ID domain. This value also corresponds to the bit-width of the sub-field of x2APIC ID corresponding the hierarchical domain with sub-leaf index 0.

• For each subsequent higher sub-leaf index m, CPUID.(EAX=0BH or 1FH, ECX=m):EAX[4:0] reports the right-shift count that will allow software to extract part of x2APIC ID to distinguish higher-domain topological entities. This means the right-shift value at of sub-leaf m, corresponds to the least significant (m+1) sub-fields of the 32-bit x2APIC ID.

Example 9-17. BitWidth Determination of x2APIC ID Sub-fields

For m = 0, m < N, m ++;
{ cumulative_width[m] = CPUID.(EAX=0BH or 1FH, ECX=m): EAX[4:0]; }
BitWidth[0] = cumulative_width[0];
For m = 1, m < N, m ++;
BitWidth[m] = cumulative_width[m] - cumulative_width[m-1];

NOTE
CPUID leaf 1FH is a preferred superset to leaf 0BH. Leaf 1FH defines additional domain types, and it must be parsed by an algorithm that can handle the addition of future domain types.

Previously, only the following encoding of hierarchical domain types were defined: 0 (invalid), 1 (logical processor), and 2 (core). With the additional hierarchical domain types available (see Section 9.9.1, “Hierarchical Mapping of Shared Resources,” and Figure 9-5, “Generalized Seven-Domain Interpretation of the APIC ID”) software must not assume any “domain type” encoding value to be related to any sub-leaf index, except sub-leaf 0.

Example 9-18. Support Routines for Identifying Package, Die, Core, and Logical Processors from 32-bit x2APIC ID

a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.

// This example shows how to enumerate CPU topology domain types (domain types may or may not be known/supported by the software)
// Below is the list of sample domain types used in the example.
// Refer to the CPUID Leaf 1FH definition for the actual domain type numbers: “V2 Extended Topology Enumeration Leaf (Initial EAX Value = 1FH)”.
// LOGICAL_PROCESSOR
// CORE
// MODULE
// TILE
// DIE
// PACKAGE
// The example shows how to identify and derive the extraction bitmask for the domains with identify type LOGICAL_PROCESSOR_ID/CORE_ID/DIE_ID/PACKAGE_ID

int DeriveLogical_Processor_Mask_Offsets (void) {
MULTIPLE-PROCESSOR MANAGEMENT

IF (!HWMTSupported()) return -1;
execute cpuid with EAX = 0BH or 1FH, ECX = 0;
IF (returned domain type encoding in ECX[15:8] does not match LOGICAL_PROCESSOR_ID) return -1;

Mask_Logical_Processor_shift = EAX[4:0];        //# bits shift right of APIC ID to distinguish different cores, note this can be a shift // of zero if there is only one logical processor per core.
Logical Processor Mask = ~( -1) << Mask_Logical_Processor_shift;        //shift left to derive extraction bitmask for // LOGICAL_PROCESSOR_ID

return 0;
}

b. Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.

int DeriveCore_Mask_Offsets (void)
{
    IF (!HWMTSupported()) return -1;
    execute cpuid with EAX = 0BH or 1FH, ECX = 0;
    WHILE( ECX[15:8] ) {         //domain type encoding is valid
        Mask_last_known_shift = EAX[4:0]
        IF (returned domain type encoding in ECX[15:8] matches CORE) {
            Mask_Core_shift = EAX[4:0];
        }
        ELSE IF (returned domain type encoding in ECX[15:8] matches DIE {         // Keep enumerating. Check if the next domain is the desired domain and if not, keep enumerating until you reach a known // domain or the invalid domain ("0" domain type). If there are more domains between DIE and PACKAGE, the unknown // domains will be ignored and treated as an extension of the last known domain (i.e., DIE in this case).
            Mask_Die_shift = EAX[4:0];
        }
        ECX++; execute cpuid with EAX = 0BH or 1FH;
    }

    COREPlusLogical_Processor_MASK = ~( -1) << Mask_Core_shift;
    DIEPlusCORE_MASK = ~( -1) << Mask_Die_shift;

    // Treat domains between DIE and physical package as an extension of DIE for software choosing not to implement or recognize // these unknown domains.

    CORE_MASK = COREPlusLogical_Processor_MASK ^ Logical Processor Mask;
    DIE_MASK = DIEPlusCORE_MASK ^ COREPlusLogical_Processor_MASK;
    PACKAGE_MASK = (-1) << Mask_last_known_shift;

    return -1;
}
9.9.3 Hierarchical ID of Logical Processors in an MP System

For Intel 64 and IA-32 processors, system hardware establishes an 8-bit initial APIC ID (or 32-bit APIC ID if the processor supports CPUID leaf 0BH) that is unique for each logical processor following power-up or RESET (see Section 9.6.1). Each logical processor on the system is allocated an initial APIC ID. BIOS may implement features that tell the OS to support less than the total number of logical processors on the system bus. Those logical processors that are not available to applications at runtime are halted during the OS boot process. As a result, the number valid local APIC_IDs that can be queried by affinitizing-current-thread-context (See Example 9-23) is limited to the number of logical processors enabled at runtime by the OS boot process.

Table 9-2 shows an example of the 8-bit APIC IDs that are initially reported for logical processors in a system with four Intel Xeon MP processors that support Intel Hyper-Threading Technology (a total of 8 logical processors, each physical package has two processor cores and supports Intel Hyper-Threading Technology). Of the two logical processors within a Intel Xeon processor MP, logical processor 0 is designated the primary logical processor and logical processor 1 as the secondary logical processor.

![Figure 9-7. Topological Relationships Between Hierarchical IDs in a Hypothetical MP Platform](image)

Table 9-2. Initial APIC IDs for the Logical Processors in a System that has Four Intel Xeon MP Processors Supporting Intel Hyper-Threading Technology¹

<table>
<thead>
<tr>
<th>Initial APIC ID</th>
<th>PACKAGE_ID</th>
<th>CORE_ID</th>
<th>LOGICAL_PROCESSOR_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0H</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>1H</td>
<td>0H</td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>2H</td>
<td>1H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>3H</td>
<td>1H</td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>4H</td>
<td>2H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>5H</td>
<td>2H</td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>6H</td>
<td>3H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>7H</td>
<td>3H</td>
<td>0H</td>
<td>1H</td>
</tr>
</tbody>
</table>

**NOTE:**
1. Because information on the number of processor cores in a physical package was not available in early single-core processors supporting Intel Hyper-Threading Technology, the CORE_ID can be treated as 0.

Table 9-3 shows the initial APIC IDs for a hypothetical situation with a dual processor system. Each physical package providing two processor cores, and each processor core also supporting Intel Hyper-Threading Technology.
9.9.3.1 Hierarchical ID of Logical Processors with x2APIC ID

Table 9-3 shows an example of possible x2APIC ID assignments for a dual processor system that supports x2APIC. Each physical package providing four processor cores, and each processor core also supporting Intel Hyper-Threading Technology. Note that the x2APIC ID need not be contiguous in the system.

Table 9-4. Example of Possible x2APIC ID Assignment in a System that has Two Physical Processors Supporting x2APIC and Intel Hyper-Threading Technology

<table>
<thead>
<tr>
<th>x2APIC ID</th>
<th>PACKAGE_ID</th>
<th>CORE_ID</th>
<th>LOGICAL_PROCESSOR_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0H</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>1H</td>
<td>0H</td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>2H</td>
<td>0H</td>
<td>1H</td>
<td>0H</td>
</tr>
<tr>
<td>3H</td>
<td>0H</td>
<td>1H</td>
<td>1H</td>
</tr>
<tr>
<td>4H</td>
<td>1H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>5H</td>
<td>1H</td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>6H</td>
<td>1H</td>
<td>1H</td>
<td>0H</td>
</tr>
<tr>
<td>7H</td>
<td>1H</td>
<td>1H</td>
<td>1H</td>
</tr>
<tr>
<td>10H</td>
<td>1H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>11H</td>
<td>1H</td>
<td>0H</td>
<td>1H</td>
</tr>
<tr>
<td>12H</td>
<td>1H</td>
<td>1H</td>
<td>0H</td>
</tr>
<tr>
<td>13H</td>
<td>1H</td>
<td>1H</td>
<td>1H</td>
</tr>
<tr>
<td>14H</td>
<td>1H</td>
<td>2H</td>
<td>0H</td>
</tr>
<tr>
<td>15H</td>
<td>1H</td>
<td>2H</td>
<td>1H</td>
</tr>
<tr>
<td>16H</td>
<td>1H</td>
<td>3H</td>
<td>0H</td>
</tr>
<tr>
<td>17H</td>
<td>1H</td>
<td>3H</td>
<td>1H</td>
</tr>
</tbody>
</table>
9.9.4 Algorithm for Three-Domain Mappings of APIC_ID

Software can gather the initial APIC_IDs for each logical processor supported by the operating system at runtime and extract identifiers corresponding to the three domains of sharing topology (package, core, and logical processor). The three-domain algorithms below focus on a non-clustered MP system for simplicity. They do not assume APIC IDs are contiguous or that all logical processors on the platform are enabled.

Intel supports multi-threading systems where all physical processors report identical values in CPUID leaf 0BH, CPUID.1:EBX[23:16], CPUID.4\(^1\):EAX[31:26], and CPUID.4\(^2\):EAX[25:14]. The algorithms below assume the target system has symmetry across physical package boundaries with respect to the number of logical processors per package, number of cores per package, and cache topology within a package.

Software can choose to assume three-domain hierarchy if it was developed to understand only three domains. However, software implementation needs to ensure it does not break if it runs on systems that have more domains in the hierarchy even if it does not recognize them.

The extraction algorithm (for three-domain mappings from an APIC ID) uses the general procedure depicted in Example 9-19, and is supplemented by more detailed descriptions on the derivation of topology enumeration parameters for extraction bit masks:

1. Detect hardware multi-threading support in the processor.
2. Derive a set of bit masks that can extract the sub ID of each hierarchical domain of the topology. The algorithm to derive extraction bit masks for LOGICAL_PROCESSOR_ID/CORE_ID/PACKAGE_ID differs based on APIC ID is 32-bit (see step 3 below) or 8-bit (see step 4 below).
3. If the processor supports CPUID leaf 0BH, each APIC ID contains a 32-bit value, the topology enumeration parameters needed to derive three-domain extraction bit masks are:
   a. Query the right-shift value for the LOGICAL_PROCESSOR_ID domain of the topology using CPUID leaf 0BH with ECX = 0H as input. The number of bits to shift-right on x2APIC ID (EAX[4:0]) can distinguish different higher-domain entities above logical processor in the same physical package. This is also the width of the bit mask to extract the LOGICAL_PROCESSOR_ID. The shift value may be 0 and enumerate no logical processor bit mask to create. A platform where cores only have one logical processor are not required to enumerate a separate bit layout for logical processor, and the lowest bits may only identify the core (where core and logical processor are then synonymous).
   b. Enumerate until the desired domain is found (i.e., processor cores). Determine if the next domain is the expected domain. If the next domain is not known to the software, keep enumerating until the next known or the last domain. Software should use the previous domain before this to represent the last previously known domain (i.e., processor cores). If the software does not recognize or implement certain hierarchical domains, it should assume these unknown domains as an extension of the last known domain.
   c. Query CPUID leaf 0BH for the amount of bit shift to distinguish next higher-domain entities (e.g., physical processor packages) in the system. This describes an explicit three-domain-topology situation for commonly available processors. Consult Example 9-17 to adapt to situations beyond a three-domain topology of a physical processor. The width of the extraction bit mask can be used to derive the cumulative extraction bitmask to extract the sub IDs of logical processors (including different processor cores) in the same physical package. The extraction bit mask to distinguish merely different processor cores can be derived by xor’ing the logical processor extraction bit mask from the cumulative extraction bit mask.
   d. Query the 32-bit x2APIC ID for the logical processor where the current thread is executing.
   e. Derive the extraction bit masks corresponding to LOGICAL_PROCESSOR_ID, CORE_ID, and PACKAGE_ID, starting from LOGICAL_PROCESSOR_ID.
   f. Apply each extraction bit mask to the 32-bit x2APIC ID to extract sub-field IDs.

---

11. As noted in Section 9.6 and Section 9.9.3, the number of logical processors supported by the OS at runtime may be less than the total number logical processors available in the platform configuration.
12. Maximum number of addressable ID for processor cores in a physical processor is obtained by executing CPUID with EAX=4 and a valid ECX index. The ECX index starts at 0.
13. Maximum number addressable ID for processor cores sharing the target cache level is obtained by executing CPUID with EAX = 4 and the ECX index corresponding to the target cache level.
4. If the processor does not support CPUID leaf 0BH, each initial APIC ID contains an 8-bit value, the topology enumeration parameters needed to derive extraction bit masks are:
   a. Query the size of address space for sub IDs that can accommodate logical processors in a physical processor package. This size parameters (CPUID.1:EBX[23:16]) can be used to derive the width of an extraction bitmask to enumerate the sub IDs of different logical processors in the same physical package.
   b. Query the size of address space for sub IDs that can accommodate processor cores in a physical processor package. This size parameters can be used to derive the width of an extraction bitmask to enumerate the sub IDs of processor cores in the same physical package.
   c. Query the 8-bit initial APIC ID for the logical processor where the current thread is executing.
   d. Derive the extraction bit masks using respective address sizes corresponding to LOGICAL_PROCESSOR_ID, CORE_ID, and PACKAGE_ID, starting from LOGICAL_PROCESSOR_ID.
   e. Apply each extraction bit mask to the 8-bit initial APIC ID to extract sub-field IDs.

Example 9-19. Support Routines for Detecting Hardware Multi-Threading and Identifying the Relationships Between Package, Core, and Logical Processors

1. Detect support for Hardware Multi-Threading Support in a processor.

    // Returns a non-zero value if CPUID reports the presence of hardware multi-threading
    // support in the physical package where the current logical processor is located.
    // This does not guarantee BIOS or OS will enable all logical processors in the physical
    // package and make them available to applications.
    // Returns zero if hardware multi-threading is not present.

    #define HWMT_BIT 10000000H

    unsigned int HWMTSupported(void) {
        // ensure cpuid instruction is supported
        execute cpuid with eax = 0 to get vendor string
        execute cpuid with eax = 1 to get feature flag and signature

        // Check to see if this a Genuine Intel Processor
        if (vendor string EQ GenuineIntel) {
            return (feature_flag_edx & HWMT_BIT); // bit 28
        }
        return 0;
    }

Example 9-20. Support Routines for Identifying Package, Core, and Logical Processors from 32-bit x2APIC ID

a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.

    int DeriveLogical_Processor_Mask_Offsets (void) {
        if (!HWMTSupported()) return -1;
        execute cpuid with eax = 11, ECX = 0;
        If (returned domain type encoding in ECX[15:8] does not match logical processor) return -1;
        Mask_Logical_Processor_shift = EAX[4:0]; // # bits shift right of APIC ID to distinguish different cores, note this can be a shift
        // of zero if there is only one logical processor per core.
        Logical Processor Mask = ~( (-1) << Mask_Logical_Processor_shift); // shift left to derive extraction bitmask for
        // LOGICAL_PROCESSOR_ID

9-42 Vol. 3A
b. **Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.**

```c
int DeriveCore_Mask_Offsets (void)
{
    if (!HWMTSupported()) return -1;
    execute cpuid with eax = 11, ECX = 0;
    while( ECX[15:8] ) { // doma in type encoding is valid
        Mask_Core_shift = EAX[4:0];   // needed to distinguish different physical packages
        ECX ++;
        execute cpuid with eax = 11;
    }
    COREPlusLogical_Processor_MASK = ~( (-1) << Mask_Core_shift);
    // treat domains between core and physical package as a core for software choosing not to implement or recognize
    // these unknown domains
    CORE_MASK = COREPlusLogical_Processor_MASK ^ Logical Processor Mask;
    PACKAGE_MASK = (-1) << Mask_Core_shift;
    return -1;
}
```

c. **Query the x2APIC ID of a logical processor.**

```c
unsigned char Getx2APIC_ID (void)
{
    unsigned reg_edx = 0;
    execute cpuid with eax = 11, ECX = 0
    store returned value of edx
    return (unsigned) (reg_edx) ;
}
```

**Example 9-21. Support Routines for Identifying Package, Core, and Logical Processors from 8-bit Initial APIC ID**

a. **Find the size of address space for logical processors in a physical processor package.**

```c
#define NUM_LOGICAL_BITS 00FF0000H
// Use the mask above and CPUID.1.EBX[23:16] to obtain the max number of addressable IDs
// for logical processors in a physical package,

//Returns the size of address space of logical processors in a physical processor package;
// Software should not assume the value to be a power of 2.

unsigned char MaxLPIDsPerPackage(void)
{
    if (!HWMTSupported()) return 1;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned char) ((reg_ebx & NUM_LOGICAL_BITS) >> 16);
}
```
b. Find the size of address space for processor cores in a physical processor package.

// Returns the max number of addressable IDs for processor cores in a physical processor package;
// Software should not assume cpuid reports this value to be a power of 2.

unsigned MaxCoreIDsPerPackage(void)
{
    if (!HWMTSupported()) return (unsigned char) 1;
    if cpuid supports leaf number 4
        { // we can retrieve multi-core topology info using leaf 4
            execute cpuid with eax = 4, ecx = 0
            store returned value of eax
            return (unsigned) ((reg_eax >> 26) +1);
        }
    else // must be a single-core processor
        return 1;
}

c. Query the initial APIC ID of a logical processor.

#define INITIAL_APIC_ID_BITS FF000000H // CPUID.1.EBX[31:24] initial APIC ID

// Returns the 8-bit unique initial APIC ID for the processor running the code.
// Software can use OS services to affinitize the current thread to each logical processor
// available under the OS to gather the initial APIC_IDs for each logical processor.

unsigned GetInitAPIC_ID (void)
{
    unsigned int reg_ebx = 0;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned) ((reg_ebx & INITIAL_APIC_ID_BITS) >> 24;
}

d. Find the width of an extraction bitmask from the maximum count of the bit-field (address size).

// Returns the mask bit width of a bit field from the maximum count that bit field can represent.
// This algorithm does not assume ‘address size’ to have a value equal to power of 2.
// Address size for LOGICAL_PROCESSOR_ID can be calculated from MaxLPIDsPerPackage()/MaxCoreIDsPerPackage()
// Then use the routine below to derive the corresponding width of logical processor extraction bitmask
// Address size for CORE_ID is MaxCoreIDsPerPackage(),
// Derive the bitwidth for CORE extraction mask similarly

unsigned FindMaskWidth(Unsigned Max_Count)
{unsigned int mask_width, cnt = Max_Count;
 __asm {
     mov eax, cnt
     mov ecx, 0
     mov mask_width, ecx
dec eax
bsr cx, ax
jz next
inc cx
mov mask_width, ecx
next:
mov eax, mask_width
}
MULTIPLE-PROCESSOR MANAGEMENT

} return mask_width;
}

e. Extract a sub ID from an 8-bit full ID, using address size of the sub ID and shift count.

// The routine below can extract LOGICAL_PROCESSOR_ID, CORE_ID, and PACKAGE_ID respectively from the init APIC_ID
// To extract LOGICAL_PROCESSOR_ID, MaxSubIdvalue is set to the address size of LOGICAL_PROCESSOR_ID, Shift_Count = 0
// To extract CORE_ID, MaxSubIdvalue is the address size of CORE_ID, Shift_Count is width of logical processor extraction bitmask.
// Returns the value of the sub ID, this is not a zero-based value

Unsigned char GetSubID(unsigned char Full_ID, unsigned char MaxSubIdValue, unsigned char Shift_Count)
{
    MaskWidth = FindMaskWidth(MaxSubIdValue);
    MaskBits = ((uchar) (FFH << Shift_Count)) ^ ((uchar) (FFH << Shift_Count + MaskWidth));
    SubID = Full_ID & MaskBits;
    Return SubID;
}

Software must not assume local APIC_ID values in an MP system are consecutive. Non-consecutive local APIC_IDS may be the result of hardware configurations or debug features implemented in the BIOS or OS.
An identifier for each hierarchical domain can be extracted from an 8-bit APIC_ID using the support routines illustrated in Example 9-21. The appropriate bit mask and shift value to construct the appropriate bit mask for each domain must be determined dynamically at runtime.

9.9.5 Identifying Topological Relationships in an MP System

To detect the number of physical packages, processor cores, or other topological relationships in a MP system, the following procedures are recommended:
• Extract the three-domain identifiers from the APIC ID of each logical processor enabled by system software. The sequence is as follows (see the pseudo code shown in Example 9-22 and support routines shown in Example 9-19):
  • The extraction start from the right-most bit field, corresponding to LOGICAL_PROCESSOR_ID, the innermost hierarchy in a three-domain topology (See Figure 9-7). For the right-most bit field, the shift value of the working mask is zero. The width of the bit field is determined dynamically using the maximum number of logical processor per core, which can be derived from information provided from CPUID.
  • To extract the next bit-field, the shift value of the working mask is determined from the width of the bit mask of the previous step. The width of the bit field is determined dynamically using the maximum number of cores per package.
  • To extract the remaining bit-field, the shift value of the working mask is determined from the maximum number of logical processor per package. So the remaining bits in the APIC ID (excluding those bits already extracted in the two previous steps) are extracted as the third identifier. This applies to a non-clustered MP system, or if there is no need to distinguish between PACKAGE_ID and CLUSTER_ID.
If there is need to distinguish between PACKAGE_ID and CLUSTER_ID, PACKAGE_ID can be extracted using an algorithm similar to the extraction of CORE_ID, assuming the number of physical packages in each node of a clustered system is symmetric.
• Assemble the three-domain identifiers of LOGICAL_PROCESSOR_ID, CORE_ID, PACKAGE_IDs into arrays for each enabled logical processor. This is shown in Example 9-23a.
• To detect the number of physical packages: use PACKAGE_ID to identify those logical processors that reside in the same physical package. This is shown in Example 9-23b. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same package.
MULTIPLE-PROCESSOR MANAGEMENT

To detect the number of processor cores: use CORE_ID to identify those logical processors that reside in the same core. This is shown in Example 9-23. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same core.

In Example 9-22, the numerical ID value can be obtained from the value extracted with the mask by shifting it right by shift count. Algorithms below do not shift the value. The assumption is that the SubID values can be compared for equivalence without the need to shift.

Example 9-22. Pseudo Code Depicting Three-Domain Extraction Algorithm

For Each local_APIC_ID{
    // Calculate Logical Processor Mask, the bit mask pattern to extract LOGICAL_PROCESSOR_ID,
    // Logical Processor Mask is determined using topology enumeration parameters
    // from CPUID leaf 0BH (Example 9-20);
    // otherwise, Logical Processor Mask is determined using CPUID leaf 01H and leaf 04H (Example 9-21).
    // This algorithm assumes there is symmetry across core boundary, i.e., each core within a package has the same number of logical processors
    // LOGICAL_PROCESSOR_ID always starts from bit 0, corresponding to the right-most bit-field
    LOGICAL_PROCESSOR_ID = APIC_ID & Logical Processor Mask;

    // Extract CORE_ID:
    // Core Mask is determined in Example 9-20 or Example 9-21
    CORE_ID = (APIC_ID & Core Mask);

    // Extract PACKAGE_ID:
    // Assume single cluster.
    // Shift out the mask width for maximum logical processors per package
    // Package Mask is determined in Example 9-20 or Example 9-21
    PACKAGE_ID = (APIC_ID & Package Mask);
}

Example 9-23. Compute the Number of Packages, Cores, and Processor Relationships in a MP System

a) Assemble lists of PACKAGE_ID, CORE_ID, and LOGICAL_PROCESSOR_ID of each enabled logical processors

// The BIOS and/or OS may limit the number of logical processors available to applications after system boot.
// The below algorithm will compute topology for the processors visible to the thread that is computing it.

// Extract the 3-domains of IDs on every processor.
// SystemAffinity is a bitmask of all the processors started by the OS. Use OS specific APIs to obtain it.
// ThreadAffinityMask is used to affinitize the topology enumeration thread to each processor using OS specific APIs.
// Allocate per processor arrays to store the Package_ID, Core_ID, and LOGICAL_PROCESSOR_ID for every started processor.

ThreadAffinityMask = 1;
ProcessorNum = 0;
while (ThreadAffinityMask ≠ 0 && ThreadAffinityMask <= SystemAffinity) {
    // Check to make sure we can utilize this processor first.
    if (ThreadAffinityMask & SystemAffinity)
        Set thread to run on the processor specified in ThreadAffinityMask
        Wait if necessary and ensure thread is running on specified processor

        APIC_ID = GetAPIC_ID(); // 32 bit ID in Example 9-20 or 8-bit ID in Example 9-21
        Extract the Package_ID, Core_ID, and LOGICAL_PROCESSOR_ID as explained in three domain extraction algorithm of Example 9-22
        PackageID[ProcessorNUM] = PACKAGE_ID;
        CoreID[ProcessorNum] = CORE_ID;

}
MULTIPLE-PROCESSOR MANAGEMENT

LOGICAL_PROCESSOR_ID[ProcessorNum] = LOGICAL_PROCESSOR_ID;
ProcessorNum++;
}
ThreadAffinityMask <<= 1;
}
NumStartedLPs = ProcessorNum;

b) Using the list of PACKAGE_ID to count the number of physical packages in a MP system and construct, for each package, a multi-bit mask corresponding to those logical processors residing in the same package.

// Compute the number of packages by counting the number of processors with unique PACKAGE_IDs in the PackageID array.
// Compute the mask of processors in each package.

// PackageIDBucket is an array of unique PACKAGE_ID values. Allocate an array of NumStartedLPs count of entries in this array.
// PackageProcessorMask is a corresponding array of the bit mask of processors belonging to the same package, these are // processors with the same PACKAGE_ID.
// The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP //system.
// Bucket Package IDs and compute processor mask for every package.

PackageNum = 1;
PackageIDBucket[0] = PackageID[0];
ProcessorMask = 1;
PackageProcessorMask[0] = ProcessorMask;
For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
    ProcessorMask <<= 1;
    For (i=0; i < PackageNum; i++) {
        // we may be comparing bit-fields of logical processors residing in different packages, the code below assume package symmetry
        If (PackageID[ProcessorNum] = PackageIDBucket[i]) {
            PackageProcessorMask[i] |= ProcessorMask;
            Break; // found in existing bucket, skip to next iteration
        }
    }
    if (i =PackageNum) {
        //PACKAGE_ID did not match any bucket, start new bucket
        PackageIDBucket[i] = PackageID[ProcessorNum];
        PackageProcessorMask[i] = ProcessorMask;
        PackageNum++;
    }
}
// PackageNum has the number of Packages started in OS
// PackageProcessorMask[] array has the processor set of each package

c) Using the list of CORE_ID to count the number of cores in a MP system and construct, for each core, a multi-bit mask corresponding to those logical processors residing in the same core.

Processors in the same core can be determined by bucketing the processors with the same PACKAGE_ID and CORE_ID. Note that code below can BIT OR the values of PACKAGE and CORE ID because they have not been shifted right.
The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.

//Bucketing PACKAGE and CORE IDs and computing processor mask for every core
CoreNum = 1;
CoreIDBucket[0] = PackageID[0] | CoreID[0];
ProcessorMask = 1;
MULTIPLE-PROCESSOR MANAGEMENT

```c
CoreProcessorMask[0] = ProcessorMask;
For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
    ProcessorMask <<= 1;
    For (i=0; i < CoreNum; i++) {
        // we may be comparing bit-fields of logical processors residing in different
        // packages, the code below assume package symmetry
        If ((PackageID[ProcessorNum] | CoreID[ProcessorNum]) = CoreIDBucket[i]) {
            CoreProcessorMask[i] |= ProcessorMask;
            Break; // found in existing bucket, skip to next iteration
        }
    }
    // Did not match any bucket, start new bucket
    CoreIDBucket[i] = PackageID[ProcessorNum] | CoreID[ProcessorNum];
    CoreProcessorMask[i] = ProcessorMask;
    CoreNum++;
}
// CoreNum has the number of cores started in the OS
// CoreProcessorMask[] array has the processor set of each core
```

Other processor relationships such as processor mask of sibling cores can be computed from set operations of the PackageProcessorMask[] and CoreProcessorMask[].

The algorithm shown above can be adapted to work with earlier generations of single-core IA-32 processors that support Intel Hyper-Threading Technology and in situations that the deterministic cache parameter leaf is not supported (provided CPUID supports initial APIC ID). A reference code example is available (see Intel® 64 Architecture Processor Topology Enumeration Technical Paper).

### 9.10 MANAGEMENT OF IDLE AND BLOCKED CONDITIONS

When a logical processor in an MP system (including multi-core processor or processors supporting Intel Hyper-Threading Technology) is idle (no work to do) or blocked (on a lock or semaphore), additional management of the core execution engine resource can be accomplished by using the HLT (halt), PAUSE, or the MONITOR/MWAIT instructions.

#### 9.10.1 HLT Instruction

The HLT instruction stops the execution of the logical processor on which it is executed and places it in a halted state until further notice (see the description of the HLT instruction in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). When a logical processor is halted, active logical processors continue to have full access to the shared resources within the physical package. Here shared resources that were being used by the halted logical processor become available to active logical processors, allowing them to execute at greater efficiency. When the halted logical processor resumes execution, shared resources are again shared among all active logical processors. (See Section 9.10.6.3, "Halt Idle Logical Processors," for more information about using the HLT instruction with processors supporting Intel Hyper-Threading Technology.)

#### 9.10.2 PAUSE Instruction

The PAUSE instruction can improve the performance of processors supporting Intel Hyper-Threading Technology when executing "spin-wait loops” and other routines where one thread is accessing a shared lock or semaphore in a tight polling loop. When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor’s pipeline. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses
this hint to avoid the memory order violation and prevent the pipeline flush. In addition, the PAUSE instruction de-piplines the spin-wait loop to prevent it from consuming execution resources excessively and consume power needlessly. (See Section 9.10.6.1, "Use the PAUSE Instruction in Spin-Wait Loops," for more information about using the PAUSE instruction with IA-32 processors supporting Intel Hyper-Threading Technology.)

9.10.3 Detecting Support MONITOR/MWAIT Instruction

Streaming SIMD Extensions 3 introduced two instructions (MONITOR and MWAIT) to help multithreaded software improve thread synchronization. In the initial implementation, MONITOR and MWAIT are available to software at ring 0. The instructions are conditionally available at levels greater than 0. Use the following steps to detect the availability of MONITOR and MWAIT:

- Use CPUID to query the MONITOR bit (CPUID.1.ECX[3] = 1).
- If CPUID indicates support, execute MONITOR inside a TRY/EXCEPT exception handler and trap for an exception. If an exception occurs, MONITOR and MWAIT are not supported at a privilege level greater than 0. See Example 9-24.

Example 9-24. Verifying MONITOR/MWAIT Support

```java
boolean MONITOR_MWAIT_works = TRUE;
try {
    _asm {
        xor ecx, ecx
        xor edx, edx
        mov eax, MemArea
        monitor
    } // Use monitor
} except (UNWIND) {
    // if we get here, MONITOR/MWAIT is not supported
    MONITOR_MWAIT_works = FALSE;
}
```

9.10.4 MONITOR/MWAIT Instruction

Operating systems usually implement idle loops to handle thread synchronization. In a typical idle-loop scenario, there could be several “busy loops” and they would use a set of memory locations. An impacted processor waits in a loop and poll a memory location to determine if there is available work to execute. The posting of work is typically a write to memory (the work-queue of the waiting processor). The time for initiating a work request and getting it scheduled is on the order of a few bus cycles.

From a resource sharing perspective (logical processors sharing execution resources), use of the HLT instruction in an OS idle loop is desirable but has implications. Executing the HLT instruction on a idle logical processor puts the targeted processor in a non-execution state. This requires another processor (when posting work for the halted logical processor) to wake up the halted processor using an inter-processor interrupt. The posting and servicing of such an interrupt introduces a delay in the servicing of new work requests.

In a shared memory configuration, exits from busy loops usually occur because of a state change applicable to a specific memory location; such a change tends to be triggered by writes to the memory location by another agent (typically a processor).

MONITOR/MWAIT complement the use of HLT and PAUSE to allow for efficient partitioning and un-partitioning of shared resources among logical processors sharing physical resources. MONITOR sets up an effective address range that is monitored for write-to-memory activities; MWAIT places the processor in an optimized state (this may vary between different implementations) until a write to the monitored address range occurs.

In the initial implementation of MONITOR and MWAIT, they are available at CPL = 0 only.
MULTIPLE-PROCESSOR MANAGEMENT

Both instructions rely on the state of the processor’s monitor hardware. The monitor hardware can be either armed (by executing the MONITOR instruction) or triggered (due to a variety of events, including a store to the monitored memory region). If upon execution of MWAIT, monitor hardware is in a triggered state: MWAIT behaves as a NOP and execution continues at the next instruction in the execution stream. The state of monitor hardware is not architecturally visible except through the behavior of MWAIT.

Multiple events other than a write to the triggering address range can cause a processor that executed MWAIT to wake up. These include events that would lead to voluntary or involuntary context switches, such as:

- External interrupts, including NMI, SMI, INIT, BINIT, MCERR, A20M#
- Faults, Aborts (including Machine Check)
- Architectural TLB invalidations including writes to CR0, CR3, CR4, and certain MSR writes; execution of LMSW (occurring prior to issuing MWAIT but after setting the monitor)
- Voluntary transitions due to fast system call and far calls (occurring prior to issuing MWAIT but after setting the monitor)

Power management related events (such as Thermal Monitor 2 or chipset driven STPCLK# assertion) will not cause the monitor event pending flag to be cleared. Faults will not cause the monitor event pending flag to be cleared.

Software should not allow for voluntary context switches in between MONITOR/MWAIT in the instruction flow. Note that execution of MWAIT does not re-arm the monitor hardware. This means that MONITOR/MWAIT need to be executed in a loop. Also note that exits from the MWAIT state could be due to a condition other than a write to the triggering address; software should explicitly check the triggering data location to determine if the write occurred. Software should also check the value of the triggering address following the execution of the monitor instruction (and prior to the execution of the MWAIT instruction). This check is to identify any writes to the triggering address that occurred during the course of MONITOR execution.

The address range provided to the MONITOR instruction must be of write-back caching type. Only write-back memory type stores to the monitored address range will trigger the monitor hardware. If the address range is not in memory of write-back type, the address monitor hardware may not be set up properly or the monitor hardware may not be armed. Software is also responsible for ensuring that

- Writes that are not intended to cause the exit of a busy loop do not write to a location within the address region being monitored by the monitor hardware,
- Writes intended to cause the exit of a busy loop are written to locations within the monitored address region.

Not doing so will lead to more false wakeups (an exit from the MWAIT state not due to a write to the intended data location). These have negative performance implications. It might be necessary for software to use padding to prevent false wakeups. CPUID provides a mechanism for determining the size data locations for monitoring as well as a mechanism for determining the size of a the pad.

9.10.5 Monitor/Mwait Address Range Determination

To use the MONITOR/MWAIT instructions, software should know the length of the region monitored by the MONITOR/MWAIT instructions and the size of the coherence line size for cache-snoop traffic in a multiprocessor system. This information can be queried using the CPUID monitor leaf function (EAX = 05H). You will need the smallest and largest monitor line size:

- To avoid missed wake-ups: make sure that the data structure used to monitor writes fits within the smallest monitor line-size. Otherwise, the processor may not wake up after a write intended to trigger an exit from MWAIT.
- To avoid false wake-ups; use the largest monitor line size to pad the data structure used to monitor writes. Software must make sure that beyond the data structure, no unrelated data variable exists in the triggering area for MWAIT. A pad may be needed to avoid this situation.

These above two values bear no relationship to cache line size in the system and software should not make any assumptions to that effect. Within a single-cluster system, the two parameters should default to be the same (the size of the monitor triggering area is the same as the system coherence line size).

Based on the monitor line sizes returned by the CPUID, the OS should dynamically allocate structures with appropriate padding. If static data structures must be used by an OS, attempt to adapt the data structure and use a
dynamically allocated data buffer for thread synchronization. When the latter technique is not possible, consider not using MONITOR/MWAIT when using static data structures.

To set up the data structure correctly for MONITOR/MWAIT on multi-clustered systems: interaction between processors, chipsets, and the BIOS is required (system coherence line size may depend on the chipset used in the system; the size could be different from the processor’s monitor triggering area). The BIOS is responsible to set the correct value for system coherence line size using the IA32_MONITOR_FILTER_LINE_SIZE MSR. Depending on the relative magnitude of the size of the monitor triggering area versus the value written into the IA32_MONITOR_FILTER_LINE_SIZE MSR, the smaller of the parameters will be reported as the Smallest Monitor Line Size. The larger of the parameters will be reported as the Largest Monitor Line Size.

9.10.6 Required Operating System Support

This section describes changes that must be made to an operating system to run on processors supporting Intel Hyper-Threading Technology. It also describes optimizations that can help an operating system make more efficient use of the logical processors sharing execution resources. The required changes and suggested optimizations are representative of the types of modifications that appear in Windows* XP and Linux* kernel 2.4.0 operating systems for Intel processors supporting Intel Hyper-Threading Technology. Additional optimizations for processors supporting Intel Hyper-Threading Technology are described in the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

9.10.6.1 Use the PAUSE Instruction in Spin-Wait Loops

Intel recommends that a PAUSE instruction be placed in all spin-wait loops that run on Intel processors supporting Intel Hyper-Threading Technology and multi-core processors.

Software routines that use spin-wait loops include multiprocessor synchronization primitives (spin-locks, semaphores, and mutex variables) and idle loops. Such routines keep the processor core busy executing a load-compare-branch loop while a thread waits for a resource to become available. Including a PAUSE instruction in such a loop greatly improves efficiency (see Section 9.10.2, “PAUSE Instruction”). The following routine gives an example of a spin-wait loop that uses a PAUSE instruction:

```
Spin_Lock:
    CMP lockvar, 0 ;Check if lock is free
    JE Get_Lock
    PAUSE ;Short delay
    JMP Spin_Lock

Get_Lock:
    MOV EAX, 1
    XCHG EAX, lockvar ;Try to get lock
    CMP EAX, 0 ;Test if successful
    JNE Spin_Lock

Critical_Section:
    ;<critical section code>
    MOV lockvar, 0

Continue:
```

The spin-wait loop above uses a “test, test-and-set” technique for determining the availability of the synchronization variable. This technique is recommended when writing spin-wait loops.

In IA-32 processor generations earlier than the Pentium 4 processor, the PAUSE instruction is treated as a NOP instruction.

9.10.6.2 Potential Usage of MONITOR/MWAIT in C0 Idle Loops

An operating system may implement different handlers for different idle states. A typical OS idle loop on an ACPI-compatible OS is shown in Example 9-25:
Example 9-25. A Typical OS Idle Loop

// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue.
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshhold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1 handler
            // shown below
        }
    }
}

// C1 handler uses a Halt instruction
VOID C1Handler()
{
    STI
    HLT
}

The MONITOR and MWAIT instructions may be considered for use in the C0 idle state loops, if MONITOR and MWAIT are supported.

Example 9-26. An OS Idle Loop with MONITOR/MWAIT in the C0 Idle Loop

// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue.
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated.
        IF (IdleTime >= IdleTimeThreshold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1 handler
            // shown below
            MONITOR WorkQueue // Setup of eax with WorkQueue
            // LinearAddress,
            // ECX, EDX = 0
            IF (WorkQueue = 0) THEN {
                MWAIT
            }
        }
    }
}
9.10.6.3 Halt Idle Logical Processors

If one of two logical processors is idle or in a spin-wait loop of long duration, explicitly halt that processor by means of a HLT instruction.

In an MP system, operating systems can place idle processors into a loop that continuously checks the run queue for runnable software tasks. Logical processors that execute idle loops consume a significant amount of core’s execution resources that might otherwise be used by the other logical processors in the physical package. For this reason, halting idle logical processors optimizes the performance.\textsuperscript{14} If all logical processors within a physical package are halted, the processor will enter a power-saving state.

9.10.6.4 Potential Usage of MONITOR/MWAIT in C1 Idle Loops

An operating system may also consider replacing HLT with MONITOR/MWAIT in its C1 idle loop. An example is shown in Example 9-27:

Example 9-27. An OS Idle Loop with MONITOR/MWAIT in the C1 Idle Loop

// WorkQueue is a memory location indicating there is a thread
// ready to run. A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue
    }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshhold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1
            // handler shown below
        }
    }
}

VOID C1Handler()

{  MONITOR WorkQueue  // Setup of eax with WorkQueue LinearAddress,
    // ECX, EDX = 0
    IF (WorkQueue = 0) THEN {
        STI
    }
}

\textsuperscript{14} Excessive transitions into and out of the HALT state could also incur performance penalties. Operating systems should evaluate the performance trade-offs for their operating system.
9.10.6.5 Guidelines for Scheduling Threads on Logical Processors Sharing Execution Resources

Because the logical processors, the order in which threads are dispatched to logical processors for execution can affect the overall efficiency of a system. The following guidelines are recommended for scheduling threads for execution.

- Dispatch threads to one logical processor per processor core before dispatching threads to the other logical processor sharing execution resources in the same processor core.
- In an MP system with two or more physical packages, distribute threads out over all the physical processors, rather than concentrate them in one or two physical processors.
- Use processor affinity to assign a thread to a specific processor core or package, depending on the cache-sharing topology. The practice increases the chance that the processor’s caches will contain some of the thread’s code and data when it is dispatched for execution after being suspended.

9.10.6.6 Eliminate Execution-Based Timing Loops

Intel discourages the use of timing loops that depend on a processor’s execution speed to measure time. There are several reasons:

- Timing loops cause problems when they are calibrated on a IA-32 processor running at one frequency and then executed on a processor running at another frequency.
- Routines for calibrating execution-based timing loops produce unpredictable results when run on an IA-32 processor supporting Intel Hyper-Threading Technology. This is due to the sharing of execution resources between the logical processors within a physical package.

To avoid the problems described, timing loop routines must use a timing mechanism for the loop that does not depend on the execution speed of the logical processors in the system. The following sources are generally available:

- A high resolution system timer (for example, an Intel 8254).
- A high resolution timer within the processor (such as, the local APIC timer or the time-stamp counter).

For additional information, see the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

9.10.6.7 Place Locks and Semaphores in Aligned, 128-Byte Blocks of Memory

When software uses locks or semaphores to synchronize processes, threads, or other code sections; Intel recommends that only one lock or semaphore be present within a cache line (or 128 byte sector, if 128-byte sector is supported). In processors based on Intel NetBurst microarchitecture (which support 128-byte sector consisting of two cache lines), following this recommendation means that each lock or semaphore should be contained in a 128-byte block of memory that begins on a 128-byte boundary. The practice minimizes the bus traffic required to service locks.

9.11 MP INITIALIZATION FOR P6 FAMILY PROCESSORS

This section describes the MP initialization process for systems that use multiple P6 family processors. This process uses the MP initialization protocol that was introduced with the Pentium Pro processor (see Section 9.4, “Multiple-Processor (MP) Initialization”). For P6 family processors, this protocol is typically used to boot 2 or 4 processors that reside on single system bus; however, it can support from 2 to 15 processors in a multi-clustered system when the APIC buses are tied together. Larger systems are not supported.
9.11.1 Overview of the MP Initialization Process for P6 Family Processors

During the execution of the MP initialization protocol, one processor is selected as the bootstrap processor (BSP) and the remaining processors are designated as application processors (APs), see Section 9.4.1, "BSP and AP Processors." Thereafter, the BSP manages the initialization of itself and the APs. This initialization includes executing BIOS initialization code and operating-system initialization code.

The MP protocol imposes the following requirements and restrictions on the system:

- An APIC clock (APICLK) must be provided.
- The MP protocol will be executed only after a power-up or RESET. If the MP protocol has been completed and a BSP has been chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each processor examines its BSP flag (in the APIC_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

The following special-purpose interprocessor interrupts (IPIs) are used during the boot phase of the MP initialization protocol. These IPIs are broadcast on the APIC bus.

- **Boot IPI (BIPI)**—Initiates the arbitration mechanism that selects a BSP from the group of processors on the system bus and designates the remainder of the processors as APs. Each processor on the system bus broadcasts a BIPI to all the processors following a power-up or RESET.
- **Final Boot IPI (FIPI)**—Initiates the BIOS initialization procedure for the BSP. This IPI is broadcast to all the processors on the system bus, but only the BSP responds to it. The BSP responds by beginning execution of the BIOS initialization code at the reset vector.
- **Startup IPI (SIPI)**—Initiates the initialization procedure for an AP. The SIPI message contains a vector to the AP initialization code in the BIOS.

Table 9-5 describes the various fields of the boot phase IPIs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Destination Field</th>
<th>Destination Shorthand</th>
<th>Trigger Mode</th>
<th>Level</th>
<th>Destination Mode</th>
<th>Delivery Mode</th>
<th>Vector (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIPI</td>
<td>Not used</td>
<td>All including self</td>
<td>Edge</td>
<td>Deassert</td>
<td>Don’t Care</td>
<td>Fixed (000)</td>
<td>40 to 4E*</td>
</tr>
<tr>
<td>FIPI</td>
<td>Not used</td>
<td>All including self</td>
<td>Edge</td>
<td>Deassert</td>
<td>Don’t Care</td>
<td>Fixed (000)</td>
<td>10</td>
</tr>
<tr>
<td>SIPI</td>
<td>Used</td>
<td>All excluding self</td>
<td>Edge</td>
<td>Assert</td>
<td>Physical</td>
<td>StartUp (110)</td>
<td>00 to FF</td>
</tr>
</tbody>
</table>

**NOTE:**

* For all P6 family processors.

For BIPI messages, the lower 4 bits of the vector field contain the APIC ID of the processor issuing the message and the upper 4 bits contain the “generation ID” of the message. All P6 family processor will have a generation ID of 4H. BIPIs will therefore use vector values ranging from 40H to 4E (4FH can not be used because FH is not a valid APIC ID).

9.11.2 MP Initialization Protocol Algorithm

Following a power-up or RESET of a system, the P6 family processors in the system execute the MP initialization protocol algorithm to initialize each of the processors on the system bus. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:
1. Each processor on the system bus is assigned a unique APIC ID, based on system topology (see Section 9.4.5, "Identifying Logical Processors in an MP System"). This ID is written into the local APIC ID register for each processor.

2. Each processor executes its internal BIST simultaneously with the other processors on the system bus. Upon completion of the BIST (at T0), each processor broadcasts a BIPI to “all including self” (see Figure 9-8).

3. APIC arbitration hardware causes all the APICs to respond to the BIPIs one at a time (at T1, T2, T3, and T4).

4. When the first BIPI is received (at time T1), each APIC compares the four least significant bits of the BIPI’s vector field with its APIC ID. If the vector and APIC ID match, the processor selects itself as the BSP by setting the BSP flag in its IA32_APIC_BASE MSR. If the vector and APIC ID do not match, the processor selects itself as an AP by entering the “wait for SIPI” state. (Note that in Figure 9-8, the BIPI from processor 1 is the first BIPI to be handled, so processor 1 becomes the BSP.)

5. The newly established BSP broadcasts an FIPI message to “all including self.” The FIPI is guaranteed to be handled only after the completion of the BIPIs that were issued by the non-BSP processors.

6. After the BSP has been established, the outstanding BIPIs are received one at a time (at T2, T3, and T4) and ignored by all processors.

7. When the FIPI is finally received (at T5), only the BSP responds to it. It responds by fetching and executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).

8. As part of the boot-strap code, the BSP creates an ACPI table and an MP table and adds its initial APIC ID to these tables as appropriate.

9. At the end of the boot-strap procedure, the BSP broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 0000 V000H, where VV is the vector contained in the SIPI message).

10. All APs respond to the SIPI message by racing to a BIOS initialization semaphore. The first one to the semaphore begins executing the initialization code. (See MP init code for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and MP tables as appropriate. At the completion of the initialization procedure, the AP executes a CLI instruction (to clear the IF flag in the EFLAGS register) and halts itself.

11. When each of the APs has gained access to the semaphore and executed the AP initialization code and all written their APIC IDs into the appropriate places in the ACPI and MP tables, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.

---

**Figure 9-8. MP System With Multiple Pentium III Processors**
12. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMI$s$, and SMI$s$. They will also respond to snoops and to assertions of the STPCLK# pin.

See Section 9.4.4, “MP Initialization Example,” for an annotated example the use of the MP protocol to boot IA-32 processors in an MP. This code should run on any IA-32 processor that used the MP protocol.

9.11.2.1 Error Detection and Handling During the MP Initialization Protocol

Errors may occur on the APIC bus during the MP initialization phase. These errors may be transient or permanent and can be caused by a variety of failure mechanisms (for example, broken traces, soft errors during bus usage, etc.). All serial bus related errors will result in an APIC checksum or acceptance error.

The MP initialization protocol makes the following assumptions regarding errors that occur during initialization:

- If errors are detected on the APIC bus during execution of the MP initialization protocol, the processors that detect the errors are shut down.
- The MP initialization protocol will be executed by processors even if they fail their BIST sequences.
CHAPTER 10
PROCESSOR MANAGEMENT AND INITIALIZATION

This chapter describes the facilities provided for managing processor wide functions and for initializing the processor. The subjects covered include: processor initialization, x87 FPU initialization, processor configuration, feature determination, mode switching, the MSRs (in the Pentium, P6 family, Pentium 4, and Intel Xeon processors), and the MTRRs (in the P6 family, Pentium 4, and Intel Xeon processors).

10.1 INITIALIZATION OVERVIEW

Following power-up or an assertion of the RESET# pin, each processor on the system bus performs a hardware initialization of the processor (known as a hardware reset) and an optional built-in self-test (BIST). A hardware reset sets each processor’s registers to a known state and places the processor in real-address mode. It also invalidates the internal caches, translation lookaside buffers (TLBs) and the branch target buffer (BTB). At this point, the action taken depends on the processor family:

- **Pentium 4 processors (CPUID DisplayFamily 0FH)** — All the processors on the system bus (including a single processor in a uniprocessor system) execute the multiple processor (MP) initialization protocol. The processor that is selected through this protocol as the bootstrap processor (BSP) then immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The application (non-BSP) processors (APs) go into a Wait For Startup IPI (SIPI) state while the BSP is executing initialization code. See Section 9.4, “Multiple-Processor (MP) Initialization,” for more details. Note that in a uniprocessor system, the single Pentium 4 or Intel Xeon processor automatically becomes the BSP.

- **IA-32 and Intel 64 processors (CPUID DisplayFamily 06H)** — The action taken is the same as for the Pentium 4 processors (as described in the previous paragraph).

- **Pentium processors** — In either a single- or dual-processor system, a single Pentium processor is always pre-designated as the primary processor. Following a reset, the primary processor behaves as follows in both single- and dual-processor systems. Using the dual-processor (DP) ready initialization protocol, the primary processor immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The secondary processor (if there is one) goes into a halt state.

- **Intel486 processor** — The primary processor (or single processor in a uniprocessor system) immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. (The Intel486 does not automatically execute a DP or MP initialization protocol to determine which processor is the primary processor.)

The software-initialization code performs all system-specific initialization of the BSP or primary processor and the system logic.

At this point, for MP (or DP) systems, the BSP (or primary) processor wakes up each AP (or secondary) processor to enable those processors to execute self-configuration code.

When all processors are initialized, configured, and synchronized, the BSP or primary processor begins executing an initial operating-system or executive task.

The x87 FPU is also initialized to a known state during hardware reset. x87 FPU software initialization code can then be executed to perform operations such as setting the precision of the x87 FPU and the exception masks. No special initialization of the x87 FPU is required to switch operating modes.

Asserting the INIT# pin on the processor invokes a similar response to a hardware reset. The major difference is that during an INIT, the internal caches, MSRs, MTRRs, and x87 FPU state are left unchanged (although, the TLBs and BTB are invalidated as with a hardware reset). An INIT provides a method for switching from protected to real-address mode while maintaining the contents of the internal caches.
### 10.1.1 Processor State After Reset

Following power-up, the state of control register CR0 is 60000010H (see Figure 10-1). This places the processor in real-address mode with paging disabled.

![Figure 10-1. Contents of CR0 Register after Reset](image)

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 23.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.

Table 10-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET; the variant cases are marked in Table 10-1 and described in more detail in Table 10-2.

<table>
<thead>
<tr>
<th>Register</th>
<th>Power up</th>
<th>Reset</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS¹</td>
<td>00000002H</td>
<td>00000002H</td>
<td>00000002H</td>
</tr>
<tr>
<td>EIP</td>
<td>0000FFFFH</td>
<td>0000FFFFH</td>
<td>0000FFFFH</td>
</tr>
<tr>
<td>CR0</td>
<td>60000010H²</td>
<td>60000010H²</td>
<td>60000010H²</td>
</tr>
<tr>
<td>CR2, CR3, CR4</td>
<td>00000000H</td>
<td>00000000H</td>
<td>00000000H</td>
</tr>
<tr>
<td>CS</td>
<td>Selector = F000H Base = FFFFFF000H Limit = FFFFH AR = Present, R/W, Accessed</td>
<td>Selector = F000H Base = FFFFFF000H Limit = FFFFH AR = Present, R/W, Accessed</td>
<td>Selector = F000H Base = FFFFFF000H Limit = FFFFH AR = Present, R/W, Accessed</td>
</tr>
<tr>
<td>EDX</td>
<td>000n06xxH⁴</td>
<td>000n06xxH⁴</td>
<td>000n06xxH⁴</td>
</tr>
<tr>
<td>EAX</td>
<td>0⁴</td>
<td>0⁴</td>
<td>0⁴</td>
</tr>
<tr>
<td>EBX, ECX, ESI, EDI, EBP, ESP</td>
<td>00000000H</td>
<td>00000000H</td>
<td>00000000H</td>
</tr>
</tbody>
</table>

¹ EFLAGS indicates the state of the processor's flags.

² CR0 indicates the state of control register CR0.

³ External x87 FPU error reporting and no task switch are not used.

⁴ EBX, ECX, ESI, EDI, EBP, ESP indicates the state of these registers.

---

10-2 Vol. 3A
<table>
<thead>
<tr>
<th>Register</th>
<th>Power up</th>
<th>Reset</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 through ST7</td>
<td>+0.0</td>
<td>+0.0</td>
<td>FINIT/FNINIT: Unchanged</td>
</tr>
<tr>
<td>x87 FPU Control Word</td>
<td>0040H</td>
<td>0040H</td>
<td>FINIT/FNINIT: 037FH</td>
</tr>
<tr>
<td>x87 FPU Status Word</td>
<td>0000H</td>
<td>0000H</td>
<td>FINIT/FNINIT: 0000H</td>
</tr>
<tr>
<td>x87 FPU Tag Word</td>
<td>5555H</td>
<td>5555H</td>
<td>FINIT/FNINIT: FFFFH</td>
</tr>
<tr>
<td>x87 FPU Data Operand and CS Seg. Selectors</td>
<td>0000H</td>
<td>0000H</td>
<td>FINIT/FNINIT: 0000H</td>
</tr>
<tr>
<td>x87 FPU Data Operand and Inst. Pointers</td>
<td>00000000H</td>
<td>00000000H</td>
<td>FINIT/FNINIT: 00000000H</td>
</tr>
<tr>
<td>MM0 through MM7</td>
<td>0000000000000000H</td>
<td>0000000000000000H</td>
<td>INIT or FINIT/FNINIT: Unchanged</td>
</tr>
<tr>
<td>XMM0 through XMM7</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>MXCSR</td>
<td>1F80H</td>
<td>1F80H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>DR0, DR1, DR2, DR3</td>
<td>00000000H</td>
<td>00000000H</td>
<td>00000000H</td>
</tr>
<tr>
<td>DR6</td>
<td>FFFF0F0H</td>
<td>FFFF0F0H</td>
<td>FFFF0F0H</td>
</tr>
<tr>
<td>DR7</td>
<td>00000400H</td>
<td>00000400H</td>
<td>00000400H</td>
</tr>
<tr>
<td>R8-R15</td>
<td>0000000000000000H</td>
<td>0000000000000000H</td>
<td>0000000000000000H</td>
</tr>
<tr>
<td>XMM8-XMM15</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>XCR0</td>
<td>1H</td>
<td>1H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_XSS</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>YMM_H[255:128]</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>BNDCFGU</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>BND0-BND3</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>IA32_BNDCFGGS</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>OPMASK</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>ZMM_H[511:256]</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>ZMMH16[511:0]</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>PKRU</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Intel Processor Trace MSRs</td>
<td>0H</td>
<td>0H^a</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Time-Stamp Counter</td>
<td>0H</td>
<td>0H^a</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_TSC_AUX</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_TSC_ADJUST</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_TSC_DEADLINE</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_SYSENTER_CS/ESP/EIP</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_EFER</td>
<td>0000000000000000H</td>
<td>0000000000000000H</td>
<td>0000000000000000H</td>
</tr>
<tr>
<td>IA32_STAR/LSTAR</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>
### Table 10-1. IA-32 and Intel® 64 Processor States Following Power-up, Reset, or INIT (Contd.)

<table>
<thead>
<tr>
<th>Register</th>
<th>Power up</th>
<th>Reset</th>
<th>INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_FS_BASE/GS_BASE</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>IA32_PMCx, IA32_PERFEVTSELx</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td>Sets bits $n$-1:0 and clears the upper bits.</td>
<td>Sets bits $n$-1:0 and clears the upper bits.</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_FIXED_CTRx, IA32_FIXED_CTR_CTRL</td>
<td>0H</td>
<td>0H</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Data and Code Cache, TLBs</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Fixed MTRRs</td>
<td>Disabled</td>
<td>Disabled</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Variable MTRRs</td>
<td>Disabled</td>
<td>Disabled</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Machine-Check Banks</td>
<td>Undefined</td>
<td>Undefined</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Last Branch Record Stack</td>
<td>0</td>
<td>0$^W$</td>
<td>Unchanged</td>
</tr>
<tr>
<td>APIC</td>
<td>Enabled</td>
<td>Enabled</td>
<td>Unchanged</td>
</tr>
<tr>
<td>X2APIC</td>
<td>Disabled</td>
<td>Disabled</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_DEBUG_INTERFACE</td>
<td>0</td>
<td>0$^W$</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where “n” is the Extended Model Value for the respective processor, and “xx” = don’t care.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
7. Where “n” is the number of general-purpose counters available in the processor. See Chapter 20, “Performance Monitoring,” for additional details.

$^W$: Warm RESET behavior differs from power-on RESET with details listed in Table 10-2.

### Table 10-2. Variance of RESET Values in Selected Intel Architecture Processors

<table>
<thead>
<tr>
<th>State</th>
<th>XREF</th>
<th>Value</th>
<th>Feature Flag or DisplayFamily_DisplayModel Signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-Stamp Counter</td>
<td>Warm RESET</td>
<td>Unmodified across warm Reset</td>
<td>06_2DH, 06_3EH</td>
</tr>
<tr>
<td>Machine-Check Banks</td>
<td>Warm RESET</td>
<td>IA32_MCI_Status banks are unmodified across warm Reset</td>
<td>06_2DH, 06_3EH, 06_3FH, 06_4FH, 06_56H</td>
</tr>
<tr>
<td>Last Branch Record Stack</td>
<td>Warm RESET</td>
<td>LBR stack MSRs are unmodified across warm Reset</td>
<td>06_1AH, 06_1CH, DisplayFamily $= 06$ and DisplayModel $&gt; 1$DH</td>
</tr>
<tr>
<td>Intel Processor Trace MSRs</td>
<td>Warm RESET</td>
<td>Clears IA32_RTIT_CTL.TraceEn, the rest of MSRs are unmodified</td>
<td>If CPUID.(EAX=14H, ECX=0H):EBX[bit 2] = 1</td>
</tr>
<tr>
<td>IA32_DEBUG_INTERFACE</td>
<td>Warm RESET</td>
<td>Unmodified across warm Reset</td>
<td>If CPUID.01H:ECX[11] = 1</td>
</tr>
</tbody>
</table>
10.1.2 Processor Built-In Self-Test (BIST)

Hardware may request that the BIST be performed at power-up. The EAX register is cleared (0H) if the processor passes the BIST. A nonzero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

The overhead for performing a BIST varies between processor families. For example, the BIST takes approximately 30 million processor clock periods to execute on the Pentium 4 processor. This clock count is model-specific; Intel reserves the right to change the number of periods for any Intel 64 or IA-32 processor, without notification.

10.1.3 Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 10-2). For example, the model, family, and processor type returned for the first processor in the Intel Pentium 4 family is as follows: model (0000B), family (1111B), and processor type (00B).

The stepping ID field contains a unique identifier for the processor’s stepping ID or revision level. The extended family and extended model fields were added to the IA-32 architecture in the Pentium 4 processors.

10.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor’s uppermost physical address. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To ensure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

10.2 X87 FPU Initialization

Software-initialization code can determine whether the processor contains an x87 FPU by using the CPUID instruction. The code must then initialize the x87 FPU and set flags in control register CR0 to reflect the state of the x87 FPU environment.
A hardware reset places the x87 FPU in the state shown in Table 10-1. This state is different from the state the x87 FPU is placed in following the execution of an FINIT or FNINIT instruction (also shown in Table 10-1). If the x87 FPU is to be used, the software-initialization code should execute an FINIT/FNINIT instruction following a hardware reset. These instructions, tag all data registers as empty, clear all the exception masks, set the TOP-of-stack value to 0, and select the default rounding and precision controls setting (round to nearest and 64-bit precision).

If the processor is reset by asserting the INIT# pin, the x87 FPU state is not changed.

### 10.2.1 Configuring the x87 FPU Environment

Initialization code must load the appropriate values into the MP, EM, and NE flags of control register CR0. These bits are cleared on hardware reset of the processor. Figure 10-3 shows the suggested settings for these flags, depending on the IA-32 processor being initialized. Initialization code can test for the type of processor present before setting or clearing these flags.

<table>
<thead>
<tr>
<th>EM</th>
<th>MP</th>
<th>NE</th>
<th>IA-32 processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Intel486™ SX, Intel386™ DX, and Intel386™ SX processors only, without the presence of a math coprocessor.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 or 0*</td>
<td>Pentium 4, Intel Xeon, P6 family, Pentium, Intel486™ DX, and Intel 487 SX processors, and Intel386 DX and Intel386 SX processors when a companion math coprocessor is present.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 or 0*</td>
<td>More recent Intel 64 or IA-32 processors</td>
</tr>
</tbody>
</table>

* The setting of the NE flag depends on the operating system being used.

The EM flag determines whether floating-point instructions are executed by the x87 FPU (EM is cleared) or a device-not-available exception (#NM) is generated for all floating-point instructions so that an exception handler can emulate the floating-point operation (EM = 1). Ordinarily, the EM flag is cleared when an x87 FPU or math coprocessor is present and set if they are not present. If the EM flag is set and no x87 FPU, math coprocessor, or floating-point emulator is present, the processor will hang when a floating-point instruction is executed.

The MP flag determines whether WAIT/FWAIT instructions react to the setting of the TS flag. If the MP flag is clear, WAIT/FWAIT instructions ignore the setting of the TS flag; if the MP flag is set, they will generate a device-not-available exception (#NM) if the TS flag is set. Generally, the MP flag should be set for processors with an integrated x87 FPU and clear for processors without an integrated x87 FPU and without a math coprocessor present. However, an operating system can choose to save the floating-point context at every context switch, in which case there would be no need to set the MP bit.

Table 2-2 shows the actions taken for floating-point and WAIT/FWAIT instructions based on the settings of the EM, MP, and TS flags.

The NE flag determines whether unmasked floating-point exceptions are handled by generating a floating-point error exception internally (NE is set, native mode) or through an external interrupt (NE is cleared). In systems where an external interrupt controller is used to invoke numeric exception handlers (such as MS-DOS-based systems), the NE bit should be cleared.

### 10.2.2 Setting the Processor for x87 FPU Software Emulation

Setting the EM flag causes the processor to generate a device-not-available exception (#NM) and trap to a software exception handler whenever it encounters a floating-point instruction. (Table 10-3 shows when it is appropriate to use this flag.) Setting this flag has two functions:

- It allows x87 FPU code to run on an IA-32 processor that has neither an integrated x87 FPU nor is connected to an external math coprocessor, by using a floating-point emulator.
- It allows floating-point code to be executed using a special or nonstandard floating-point emulator, selected for a particular application, regardless of whether an x87 FPU or math coprocessor is present.
To emulate floating-point instructions, the EM, MP, and NE flag in control register CR0 should be set as shown in Table 10-4.

<table>
<thead>
<tr>
<th>CR0 Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>1</td>
</tr>
<tr>
<td>MP</td>
<td>0</td>
</tr>
<tr>
<td>NE</td>
<td>1</td>
</tr>
</tbody>
</table>

Regardless of the value of the EM bit, the Intel486 SX processor generates a device-not-available exception (#NM) upon encountering any floating-point instruction.

### 10.3 CACHE ENABLING

IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors contain internal instruction and data caches. These caches are enabled by clearing the CD and NW flags in control register CR0. (They are set during a hardware reset.) Because all internal cache lines are invalid following reset initialization, it is not necessary to invalidate the cache before enabling caching. Any external caches may require initialization and invalidation using a system-specific initialization and invalidation code sequence.

Depending on the hardware and operating system or executive requirements, additional configuration of the processor’s caching facilities will probably be required. Beginning with the Intel486 processor, page-level caching can be controlled with the PCD and PWT flags in page-directory and page-table entries. Beginning with the P6 family processors, the memory type range registers (MTRRs) control the caching characteristics of the regions of physical memory. (For the Intel486 and Pentium processors, external hardware can be used to control the caching characteristics of regions of physical memory.) See Chapter 12, "Memory Cache Control," for detailed information on configuration of the caching facilities in the Pentium 4, Intel Xeon, and P6 family processors and system memory.

### 10.4 MODEL-SPECIFIC REGISTERS (MSRS)

Most IA-32 processors (starting from Pentium processors) and Intel 64 processors contain a model-specific registers (MSRs). A given MSR may not be supported across all families and models for Intel 64 and IA-32 processors. Some MSRs are designated as architectural to simplify software programming; a feature introduced by an architectural MSR is expected to be supported in future processors. Non-architectural MSRs are not guaranteed to be supported or to have the same functions on future processors.

MSRs that provide control for a number of hardware and software-related features, include:

- Performance-monitoring counters (see Chapter 20, “Performance Monitoring”).
- Debug extensions (see Chapter 18, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features").
- Machine-check exception capability and its accompanying machine-check architecture (see Chapter 16, "Machine-Check Architecture").
- MTRRs (see Section 12.11, "Memory Type Range Registers (MTRRs)").
- Thermal and power management.
- Instruction-specific support (for example: SYSENTER, SYSEXIT, SWAPGS, etc.).
- Processor feature/mode support (for example: IA32_EFER, IA32_FEATURE_CONTROL).

The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

When performing software initialization of an IA-32 or Intel 64 processor, many of the MSRs will need to be initialized to set up things like performance-monitoring events, run-time machine checks, and memory types for physical memory.
Lists of available performance-monitoring events can be found at: https://perfmon-events.intel.com/, and lists of available MSRs are given in Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The references earlier in this section show where the functions of the various groups of MSRs are described in this manual.

10.5 MEMORY TYPE RANGE REGISTERS (MTRRs)

Memory type range registers (MTRRs) were introduced into the IA-32 architecture with the Pentium Pro processor. They allow the type of caching (or no caching) to be specified in system memory for selected physical address ranges. They allow memory accesses to be optimized for various types of memory such as RAM, ROM, frame buffer memory, and memory-mapped I/O devices.

In general, initializing the MTRRs is normally handled by the software initialization code or BIOS and is not an operating system or executive function. At the very least, all the MTRRs must be cleared to 0, which selects the uncached (UC) memory type. See Section 12.11, “Memory Type Range Registers (MTRRs),” for detailed information on the MTRRs.

10.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS

For processors that contain SSE/SSE2/SSE3/SSSE3 extensions, steps must be taken when initializing the processor to allow execution of these instructions.

1. Check the CPUID feature flags for the presence of the SSE/SSE2/SSE3/SSSE3 extensions (respectively: EDX bits 25 and 26, ECX bit 0 and 9) and support for the FXSAVE and FXRSTOR instructions (EDX bit 24). Also check for support of the CLFLUSH instruction (EDX bit 19). The CPUID feature flags are loaded in the EDX and ECX registers when the CPUID instruction is executed with a 1 in the EAX register.

2. Set the OSFXSR flag (bit 9 in control register CR4) to indicate that the operating system supports saving and restoring the SSE/SSE2/SSE3/SSSE3 execution environment (XMM and MXCSR registers) with the FXSAVE and FXRSTOR instructions, respectively. See Section 2.5, “Control Registers,” for a description of the OSFXSR flag.

3. Set the OSXMMEXCPT flag (bit 10 in control register CR4) to indicate that the operating system supports the handling of SSE/SSE2/SSE3 SIMD floating-point exceptions (#XM). See Section 2.5, “Control Registers,” for a description of the OSXMMEXCPT flag.

4. Set the mask bits and flags in the MXCSR register according to the mode of operation desired for SSE/SSE2/SSE3 SIMD floating-point instructions. See "MXCSR Control and Status Register” in Chapter 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a detailed description of the bits and flags in the MXCSR register.

10.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION

Following a hardware reset (either through a power-up or the assertion of the RESET# pin) the processor is placed in real-address mode and begins executing software initialization code from physical address FFFFFFF0H. Software initialization code must first set up the necessary data structures for handling basic system functions, such as a real-mode IDT for handling interrupts and exceptions. If the processor is to remain in real-address mode, software must then load additional operating-system or executive code modules and data structures to allow reliable execution of application programs in real-address mode.

If the processor is going to operate in protected mode, software must load the necessary data structures to operate in protected mode and then switch to protected mode. The protected-mode data structures that must be loaded are described in Section 10.8, “Software Initialization for Protected-Mode Operation.”

10.7.1 Real-Address Mode IDT

In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the “interrupt vector table”). By default, the address of the base of the IDT is physical address 0H. This address can be
changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled.

The actual interrupt- and exception-handler code can be contained either in EPROM or RAM; however, the code must be located within the 1-MByte addressable range of the processor in real-address mode. If the handler code is to be stored in RAM, it must be loaded along with the IDT.

10.7.2 NMI Interrupt Handling

The NMI interrupt is always enabled (except when multiple NMIs are nested). If the IDT and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following hardware reset when an NMI interrupt cannot be handled. During this time, hardware must provide a mechanism to prevent an NMI interrupt from halting code execution until the IDT and the necessary NMI handler software is loaded. Here are two examples of how NMIs can be handled during the initial states of processor initialization:

• A simple IDT and NMI interrupt handler can be provided in EPROM. This allows an NMI interrupt to be handled immediately after reset initialization.

• The system hardware can provide a mechanism to enable and disable NMIs by passing the NMI# signal through an AND gate controlled by a flag in an I/O port. Hardware can clear the flag when the processor is reset, and software can set the flag when it is ready to handle NMI interrupts.

10.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION

The processor is placed in real-address mode following a hardware reset. At this point in the initialization process, some basic data structures and code modules must be loaded into physical memory to support further initialization of the processor, as described in Section 10.7, "Software Initialization for Real-Address Mode Operation." Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:

• A IDT.
• A GDT.
• A TSS.
• (Optional) An LDT.
• If paging is to be used, at least one page directory and one page table.
• A code segment that contains the code to be executed when the processor switches to protected mode.
• One or more code modules that contain the necessary interrupt and exception handlers.

Software initialization code must also initialize the following system registers before the processor can be switched to protected mode:

• The GDTR.
• (Optional.) The IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
• Control registers CR1 through CR4.
• (Pentium 4, Intel Xeon, and P6 family processors only.) The memory type range registers (MTRRs).

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by loading control register CR0 with a value that sets the PE flag (bit 0).

10.8.1 Protected-Mode System Data Structures

The contents of the protected-mode system data structures loaded into memory during software initialization, depend largely on the type of memory management the protected-mode operating-system or executive is going to support: flat, flat with paging, segmented, or segmented with paging.
To implement a flat memory model without paging, software initialization code must at a minimum load a GDT with one code and one data-segment descriptor. A null descriptor in the first GDT entry is also required. The stack can be placed in a normal read/write data segment, so no dedicated descriptor for the stack is required. A flat memory model with paging also requires a page directory and at least one page table (unless all pages are 4 MBytes in which case only a page directory is required). See Section 10.8.3, “Initializing Paging.”

Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

A multi-segmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Some operating systems allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment. However, many operating systems use a single LDT for all tasks, allocating GDT entries in advance. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a real-time system.

### 10.8.2 Initializing Protected-Mode Exceptions and Interrupts

Software initialization code must at a minimum load a protected-mode IDT with gate descriptor for each exception vector that the processor can generate. If interrupt or trap gates are used, the gate descriptors can all point to the same code segment, which contains the necessary exception handlers. If task gates are used, one TSS and accompanying code, data, and task segments are required for each exception handler called with a task gate.

If hardware allows interrupts to be generated, gate descriptors must be provided in the IDT for one or more interrupt handlers.

Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction. This operation is typically carried out immediately after switching to protected mode.

### 10.8.3 Initializing Paging

Paging is controlled by the PG flag in control register CR0. When this flag is clear (its state following a hardware reset), the paging mechanism is turned off; when it is set, paging is enabled. Before setting the PG flag, the following data structures and registers must be initialized:

- Software must load at least one page directory and one page table into physical memory. The page table can be eliminated if the page directory contains a directory entry pointing to itself (here, the page directory and page table reside in the same page), or if only 4-MByte pages are used.
- Control register CR3 (also called the PDBR register) is loaded with the physical base address of the page directory.
- (Optional) Software may provide one set of code and data descriptors in the GDT or in an LDT for supervisor mode and another set for user mode.

With this paging initialization complete, paging is enabled and the processor is switched to protected mode at the same time by loading control register CR0 with an image in which the PG and PE flags are set. (Paging cannot be enabled before the processor is switched to protected mode.)

### 10.8.4 Initializing Multitasking

If the multitasking mechanism is not going to be used and changes between privilege levels are not allowed, it is not necessary load a TSS into memory or to initialize the task register.

If the multitasking mechanism is going to be used and/or changes between privilege levels are allowed, software initialization code must load at least one TSS and an accompanying TSS descriptor. (A TSS is required to change privilege levels because pointers to the privileged-level 0, 1, and 2 stack segments and the stack pointers for these stacks are obtained from the TSS.) TSS descriptors must not be marked as busy when they are created; they should be marked busy by the processor only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT.
After the processor has switched to protected mode, the LTR instruction can be used to load a segment selector for a TSS descriptor into the task register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The processor can, however, use the TSS to locate pointers to privilege-level 0, 1, and 2 stacks. The segment selector for the TSS must be loaded before software performs its first task switch in protected mode, because a task switch copies the current task state into the TSS.

After the LTR instruction has been executed, further operations on the task register are performed by task switching. As with other segments and LDTs, TSSs and TSS descriptors can be either pre-allocated or allocated as needed.

10.8.5 Initializing IA-32e Mode

On Intel 64 processors, the IA32_EFER MSR is cleared on system reset. The operating system must be in protected mode with paging enabled before attempting to initialize IA-32e mode. IA-32e mode operation also requires physical-address extensions with four or five levels of enhanced paging structures (see Section 4.5, “4-Level Paging and 5-Level Paging”).

Operating systems should follow this sequence to initialize IA-32e mode:

1. Starting from protected mode, disable paging by setting CR0.PG = 0. Use the MOV CR0 instruction to disable paging (the instruction must be located in an identity-mapped page).
2. Enable physical-address extensions (PAE) by setting CR4.PAE = 1. Failure to enable PAE will result in a #GP fault when an attempt is made to initialize IA-32e mode.
3. Load CR3 with the physical base address of the Level 4 page map table (PML4) or Level 5 page map table (PML5).
4. Enable IA-32e mode by setting IA32_EFER.LME = 1.
5. Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page (until such time that a branch to non-identity mapped pages can be effected).

64-bit mode paging structures must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode (setting CR0.PG = 1 to enable paging). Because MOV CR3 is executed in protected mode, only the lower 32 bits of the register are written, limiting the table location to the low 4 GBytes of memory. Software can relocate the page tables anywhere in physical memory after IA-32e mode is activated.

The processor performs 64-bit mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating IA-32e mode (IA32_EFER.LME, CR0.PG, and CR4.PAE). It will generate a general protection fault (#GP) if consistency checks fail. 64-bit mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

64-bit mode consistency checks fail in the following circumstances:

- An attempt is made to enable or disable IA-32e mode while paging is enabled.
- IA-32e mode is enabled and an attempt is made to enable paging prior to enabling physical-address extensions (PAE).
- IA-32e mode is active and an attempt is made to disable physical-address extensions (PAE).
- If the current CS has the L-bit set on an attempt to activate IA-32e mode.
- If the TR contains a 16-bit TSS on an attempt to activate IA-32e mode.

10.8.5.1 IA-32e Mode System Data Structures

After activating IA-32e mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy protected-mode descriptor tables. Tables referenced by the descriptors all reside in the lower 4 GBytes of linear-address space. After activating IA-32e mode, 64-bit operating-systems should use the LGDT, LLDT, LIDT, and LTR instructions to load the system-descriptor-table registers with references to 64-bit descriptor tables.
10.8.5.2 IA-32e Mode Interrupts and Exceptions

Software must not allow exceptions or interrupts to occur between the time IA-32e mode is activated and the update of the interrupt-descriptor-table register (IDTR) that establishes references to a 64-bit interrupt-descriptor table (IDT). This is because the IDT remains in legacy form immediately after IA-32e mode is activated.

If an interrupt or exception occurs prior to updating the IDTR, a legacy 32-bit interrupt gate will be referenced and interpreted as a 64-bit interrupt gate with unpredictable results. External interrupts can be disabled by using the CLI instruction.

Non-maskable interrupts (NMI) must be disabled using external hardware.

10.8.5.3 64-bit Mode and Compatibility Mode Operation

IA-32e mode uses two code segment-descriptor bits (CS.L and CS.D, see Figure 3-8) to control the operating modes after IA-32e mode is initialized. If CS.L = 1 and CS.D = 0, the processor is running in 64-bit mode. With this encoding, the default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, operand size can be changed to 64 bits or 16 bits; address size can be changed to 32 bits.

When IA-32e mode is active and CS.L = 0, the processor operates in compatibility mode. In this mode, CS.D controls default operand and address sizes exactly as it does in the IA-32 architecture. Setting CS.D = 1 specifies default operand and address size as 32 bits. Clearing CS.D to 0 specifies default operand and address size as 16 bits (the CS.L = 1, CS.D = 1 bit combination is reserved).

Compatibility mode execution is selected on a code-segment basis. This mode allows legacy applications to coexist with 64-bit applications running in 64-bit mode. An operating system running in IA-32e mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor’s CS.L bit to 0.

In compatibility mode, the following system-level mechanisms continue to operate using the IA-32e-mode architectural semantics:

- Linear-to-physical address translation uses the 64-bit mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the 64-bit mode mechanisms.
- System calls (calls through call gates and SYSENTER/SYSEXIT) are handled using the IA-32e mode mechanisms.

10.8.5.4 Switching Out of IA-32e Mode Operation

To return from IA-32e mode to paged-protected mode operation operating systems must use the following sequence:

1. Switch to compatibility mode.
2. Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and subsequent instructions must be located in an identity-mapped page.
3. Load CR3 with the physical base address of the legacy page-table-directory base address.
4. Disable IA-32e mode by setting IA32_EFER.LME = 0.
5. Enable legacy paged-protected mode by setting CR0.PG = 1
6. A branch instruction must follow the MOV CR0 that enables paging. Both the MOV CR0 and the branch instruction must be located in an identity-mapped page.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.
10.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

10.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 10.8, “Software Initialization for Protected-Mode Operation.” Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To ensure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)
5. The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.
6. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
7. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
8. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
9. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
   - Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
   - Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
10. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
11. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.
10.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.

2. If paging is enabled, perform the following operations:
   - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
   - Ensure that the GDT and IDT are in identity mapped pages.
   - Clear the PG bit in the CR0 register.
   - Move 0H into the CR3 register to flush the TLB.

3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.

4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
   - Limit = 64 KBytes (0FFFFH)
   - Byte granular (G = 0)
   - Expand up (E = 0)
   - Writable (W = 1)
   - Present (P = 1)
   - Base = any value

   The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.

5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.

6. Clear the PE flag in the CR0 register to switch to real-address mode.

7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.

8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.

9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

   **NOTE**

   All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

10.10 Initialization and Mode Switching Example

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.
• Load the necessary protected-mode system data structures into RAM.
• Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
• Switch the processor to protected mode.

Figure 10-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor’s physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 10-5. The source listing for the example (with the filename STARTUP.ASM) is given in Example 10-1. The line numbers given in Table 10-5 refer to the source listing.

The following are some additional notes concerning this example:
• When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
• Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
• The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
• This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.
Assembler Usage

In this example, the Intel assembler ASM386 and build tools BLD386 are used to assemble and build the initialization code module. The following assumptions are used when using the Intel ASM386 and BLD386 tools.

- The ASM386 will generate the right operand size opcodes according to the code-segment attribute. The attribute is assigned either by the ASM386 invocation controls or in the code-segment definition.

- If a code segment that is going to run in real-address mode is defined, it must be set to a USE 16 attribute. If a 32-bit operand is used in an instruction in this code segment (for example, MOV EAX, EBX), the assembler automatically generates an operand prefix for the instruction that forces the processor to execute a 32-bit operation, even though its default code-segment attribute is 16-bit.

- Intel's ASM386 assembler allows specific use of the 16- or 32-bit instructions, for example, LGDTW, LGDTD, IRETD. If the generic instruction LGDT is used, the default-segment attribute will be used to generate the right opcode.

STARTUP.ASM Listing

Example 10-1 provides high-level sample code designed to move the processor into protected mode. This listing does not include any opcode and offset information.
Example 10-1. STARTUP.ASM

MS-DOS* 5.0(045-N) 386(TM) MACRO ASSEMBLER STARTUP  09:44:51 08/19/92 PAGE 1

MS-DOS 5.0(045-N) 386(TM) MACRO ASSEMBLER V4.0, ASSEMBLY OF MODULE STARTUP
OBJECT MODULE PLACED IN startup.obj
ASSEMBLER INVOKED BY: f:\386tools\ASM386.EXE startup.a58 pw (132 )

LINE   SOURCE

NAME STARTUP

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; ASSUMPTIONS:
;
1. The bottom 64K of memory is ram, and can be used for
   scratch space by this module.

2. The system has sufficient free usable ram to copy the
   initial GDT, IDT, and TSS

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; configuration data - must match with build definition

CS_BASE EQU 0FFFF0000H

; CS_BASE is the linear address of the segment STARTUP_CODE
; - this is specified in the build language file

RAM_START EQU 400H

; RAM_START is the start of free, usable ram in the linear
; memory space. The GDT, IDT, and initial TSS will be
; copied above this space, and a small data segment will be
; discarded at this linear address. The 32-bit word at
; RAM_START will contain the linear address of the first
; free byte above the copied tables - this may be useful if
; a memory manager is used.

TSS_INDEX EQU 10

; TSS_INDEX is the index of the TSS of the first task to
; run after startup

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; ------------------------- STRUCTURES and EQU ---------------
; structures for system data

TASK_STATE STRUC
   link DW ?
PROCESSOR MANAGEMENT AND INITIALIZATION

        link_h  DW  ?
        ESP0   DD  ?
        SS0    DW  ?
        SS0_h  DW  ?
        ESP1   DD  ?
        SS1    DW  ?
        SS1_h  DW  ?
        ESP2   DD  ?
        SS2    DW  ?
        SS2_h  DW  ?
        CR3_reg DD  ?
        EIP_reg DD  ?
        EFLAGS_regDD  ?
        EAX_reg  DD  ?
        ECX_reg  DD  ?
        EDX_reg  DD  ?
        EBX_reg  DD  ?
        ESP_reg  DD  ?
        EBP_reg  DD  ?
        ESI_reg  DD  ?
        EDI_reg  DD  ?
        ES_reg  DW  ?
        ES_h    DW  ?
        CS_reg  DW  ?
        CS_h    DW  ?
        SS_reg  DW  ?
        SS_h    DW  ?
        DS_reg  DW  ?
        DS_h    DW  ?
        FS_reg  DW  ?
        FS_h    DW  ?
        GS_reg  DW  ?
        GS_h    DW  ?
        LDT_reg DW  ?
        LDT_h   DW  ?
        TRAP_reg DW  ?
        IO_map_baseDW  ?
        TASK_STATE ENDS

        ; basic structure of a descriptor
        DESC  STRUC
        lim_0_15 DW  ?
        bas_0_15 DW  ?
        bas_16_23DB  ?
        access  DB  ?
        gran  DB  ?
        bas_24_31DB  ?
        DESC  ENDS

        ; structure for use with LGDT and LIDT instructions
        TABLE_REG  STRUC
        table_limDW  ?
        table_linearDD  ?
        TABLE_REG  ENDS

10-18  Vol. 3A
PROCESSOR MANAGEMENT AND INITIALIZATION

100  ; offset of GDT and IDT descriptors in builder generated GDT
101  GDT_DESC_OFF  EQU 1*SIZE(DESC)
102  IDT_DESC_OFF  EQU 2*SIZE(DESC)
104
105  ; equates for building temporary GDT in RAM
106  LINEAR_SEL   EQU 1*SIZE (DESC)
107  LINEAR_PROTO_LO   EQU 00000FFFFH ; LINEAR_ALIAS
108  LINEAR_PROTO_HI   EQU 000CF9200H
109
110  ; Protection Enable Bit in CR0
111  PE_BIT  EQU 1B
112
113  ; ------------------------------------------------------------
114
115  ; ------------------------- DATA SEGMENT----------------------
116
117  ; Initially, this data segment starts at linear 0, according
118  ; to the processor’s power-up state.
119
120  STARTUP_DATA SEGMENT RW
121
122  free_mem_linear_base   LABEL   DWORD
123  TEMP_GDT   LABEL   DWORD ; must be first in segment
124  TEMP_GDT_NULL_DESC DESC  <>
125  TEMP_GDT_LINEAR_DESC DESC  <>
126
127  ; scratch areas for LGDT and LIDT instructions
128  TEMP_GDT_SCRATCH TABLE_REG  <>
129  APP_GDT_RAM     TABLE_REG  <>
130  APP_IDT_RAM     TABLE_REG  <>
131  ; align end_data
132  fill    DW      ?
133
134  ; last thing in this segment - should be on a dword boundary
135  end_data   LABEL   BYTE
136
137  STARTUP_DATA ENDS
138
139  ; ------------------------------------------------------------
140
141  ; ------------------------- CODE SEGMENT----------------------
142  STARTUP_CODE SEGMENT ER PUBLIC USE16
143
144  ; filled in by builder
145      PUBLIC  GDTEPROM
146  GDTEPROM   TABLE_REG  <>
147
148  ; filled in by builder
149      PUBLIC  IDTEPROM
150  IDTEPROM   TABLE_REG  <>
151
152  ; entry point into startup code - the bootstrap will vector
153  ; here with a near JMP generated by the builder. This
154 ; label must be in the top 64K of linear memory.
155
156      PUBLIC  STARTUP
157  STARTUP:
158
159 ; DS,ES address the bottom 64K of flat linear memory
160      ASSUME DS:STARTUP_DATA, ES:STARTUP_DATA
161 ; See Figure 10-4
162 ; load GDTR with temporary GDT
163      LEA     EBX,TEMP_GDT ; build the TEMP_GDT in low ram,
164      MOV     DWORD PTR [EBX],0    ; where we can address
165      MOV     DWORD PTR [EBX]+4,0
166      MOV     DWORD PTR [EBX]+8, LINEAR_PROTO_LO
167      MOV     DWORD PTR [EBX]+12, LINEAR_PROTO_HI
168      MOV     TEMP_GDT_scratch.table_linear,EBX
169      MOV     TEMP_GDT_scratch.table_lim,15
170
171 DB 66H; execute a 32 bit LGDT
172      LGDT    TEMP_GDT_scratch
173
174 ; enter protected mode
175      MOV     EBX,CR0
176      OR      EBX,PE_BIT
177      MOV     CR0,EBX
178
179 ; clear prefetch queue
180      JMP      CLEAR_LABEL
181  CLEAR_LABEL:
182
183 ; make DS and ES address 4G of linear memory
184      MOV     CX,LINEAR_SEL
185      MOV     DS,CX
186      MOV     ES,CX
187
188 ; do board specific initialization
189 ;
190
191 ; ......
192
193
194
195 ; See Figure 10-5
196 ; copy EPROM GDT to ram at:
197 ;     RAM_START + size (STARTUP_DATA)
198      MOV     EAX,RAM_START
199      ADD     EAX,OFFSET (end_data)
200      MOV     EBX,RAM_START
201      MOV     ECX, CS_BASE
202      ADD     ECX, OFFSET (GDT_EPROM)
203      MOV     ESI, [ECX].table_linear
204      MOV     EDI,EAX
205      MOVZX   ECX, [ECX].table_lim
206      MOV     APP_GDT_ram[EBX].table_lim,CX
207          INC     ECX
208          MOV     EDX,EAX
209          MOV     APP_GDT_ram[EBX].table_linear,EAX
210          ADD     EAX,ECX
211          REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
212
213          ; fixup GDT base in descriptor
214          MOV     ECX,EDX
215          MOV     [EDX].bas_0_15+GDT_DESC_OFF,CX
216          ROR     ECX,16
217          MOV     [EDX].bas_16_23+GDT_DESC_OFF,CL
218          MOV     [EDX].bas_24_31+GDT_DESC_OFF,CH
219
220          ; copy EPROM IDT to ram at:
221          ; RAM_START+size(STARTUP_DATA)+SIZE (EPROM GDT)
222          MOV     ECX, CS_BASE
223          ADD     ECX, OFFSET (IDT_EEPROM)
224          MOV     ESI, [ECX].table_linear
225          MOV     EDI,EAX
226          MOVZX   ECX, [ECX].table_lim
227          MOV     APP_IDT_ram[EBX].table_lim,CX
228          INC     ECX
229          MOV     APP_IDT_ram[EBX].table_linear,EAX
230          MOV     EBX,EAX
231          ADD     EAX,ECX
232          REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
233
234          ; fixup IDT pointer in GDT
235          MOV     [EDX].bas_0_15+IDT_DESC_OFF,BX
236          ROR     EBX,16
237          MOV     [EDX].bas_16_23+IDT_DESC_OFF,BL
238          MOV     [EDX].bas_24_31+IDT_DESC_OFF,BH
239
240          ; load GDTR and IDTR
241          MOV     EBX, RAM_START
242          DB      66H         ; execute a 32 bit LGDT
243          LGDT    APP_GDT_ram[EBX]
244          DB      66H         ; execute a 32 bit LIDT
245          LIDT    APP_IDT_ram[EBX]
246
247          ; move the TSS
248          MOV     EDI,EAX
249          MOV     EBX, TSS_INDEX*SIZE(DESC)
250          MOV     ECX, GDT_DESC_OFF ; build linear address for TSS
251          MOV     GS,CX
252          MOV     DH,GS:[EBX].bas_24_31
253          MOV     DL,GS:[EBX].bas_16_23
254          ROL     EDX,16
255          MOV     DX,GS:[EBX].bas_0_15
256          MOV     ESI,EDX
257          LSL     ECX,EBX
258          INC     ECX
259          MOV     EDX, EAX
260          ADD     EAX,ECX
PROCESSOR MANAGEMENT AND INITIALIZATION

REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]

; fixup TSS pointer
MOV GS:[EBX].bas_0_15,DX
ROL EDX,16
MOV GS:[EBX].bas_24_31,DH
MOV GS:[EBX].bas_16_23,DL
ROL EDX,16

; save start of free ram at linear location RAMSTART
MOV free_mem_linear_base+RAM_START,EAX

; assume no LDT used in the initial task - if necessary, code to move the LDT could be added, and should resemble that used to move the TSS

; load task register
LTR BX ; No task switch, only descriptor loading
; See Figure 10-6
; load minimal set of registers necessary to simulate task
; switch

MOV AX,[EDX].SS_reg ; start loading registers
MOV EDI,[EDX].ESP_reg
MOV SS,AX
MOV ESP,EDI ; stack now valid
PUSH DWORD PTR [EDX].EFLAGS_reg
PUSH DWORD PTR [EDX].CS_reg
PUSH DWORD PTR [EDX].EIP_reg
MOV AX,[EDX].DS_reg
MOV BX,[EDX].ES_reg
MOV DS,AX ; DS and ES no longer linear memory
MOV ES,BX

; simulate far jump to initial task
IRETD

STARTUP_CODE ENDS

*** WARNING #377 IN 298, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)

END STARTUP, DS:STARTUP_DATA, SS:STARTUP_DATA

ASSEMBLY COMPLETE, 1 WARNING, NO ERRORS.
Figure 10-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)
Figure 10-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

- Move the GDT, IDT, TSS from ROM to RAM
- Fix Aliases
- LTR
10.10.3 MAIN.ASM Source Code

The file MAIN.ASM shown in Example 10-2 defines the data and stack segments for this application and can be substituted with the main module task written in a high-level language that is invoked by the IRET instruction executed by STARTUP.ASM.

Example 10-2. MAIN.ASM

```
NAME    main_module
data    SEGMENT RW
    dw 1000 dup(?)
DATA    ENDS
stack stackseg 800
CODE SEGMENT ER  use32 PUBLIC
main_start:
    nop
    nop
    nop
CODE  ENDS
END main_start, ds:data, ss:stack
```

10.10.4 Supporting Files

The batch file shown in Example 10-3 can be used to assemble the source code files STARTUP.ASM and MAIN.ASM and build the final application.
Example 10-3. Batch File to Assemble and Build the Application

ASM386 STARTUP.ASM
ASM386 MAIN.ASM
BLD386 STARTUP.OBJ, MAIN.OBJ buildfile(EPROM.BLD) bootstrap(STARTUP) Bootload

BLD386 performs several operations in this example:
It allocates physical memory location to segments and tables.
It generates tables using the build file and the input files.
It links object files and resolves references.
It generates a boot-loadable file to be programmed into the EPROM.

Example 10-4 shows the build file used as an input to BLD386 to perform the above functions.

Example 10-4. Build File

INIT_BLD_EXAMPLE;

SEGMENT
  *SEGMENTS(DPL = 0)
  , startup.startup_code(BASE = 0FFFF0000H)
  ;

TASK
  , BOOT_TASK(OBJECT = startup, INITIAL, DPL = 0, NOT INTENABLED)
  , PROTECTED_MODE_TASK(OBJECT = main_module, DPL = 0, NOT INTENABLED)
  ;

TABLE
  GDT {
    LOCATION = GDT_EPROM
    , ENTRY = {
      10: PROTECTED_MODE_TASK
      , startup.startup_code
      , startup.startup_data
      , main_module.data
      , main_module.code
      , main_module.stack
    }
  },

  IDT {
    LOCATION = IDT_EPROM
  };

MEMORY
{
  RESERVE = (0..3FFFH
    -- Area for the GDT, IDT, TSS copied from ROM
  , 60000H..0FFFFFFFH)
  , RANGE = (ROM_AREA = ROM (0FFFF0000H..0FFFFFFFH))
    -- Eprom size 64K
  , RANGE = (RAM_AREA = RAM (4000H..0FFFFH))

  ,
Table 10-6 shows the relationship of each build item with an ASM source file.

<table>
<thead>
<tr>
<th>Item</th>
<th>ASM386 and Startup.A58</th>
<th>BLD386 Controls and BLD file</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>public startup</td>
<td>bootstrap start(startup)</td>
<td>Near jump at OFFFFFFFFFOH to start.</td>
</tr>
<tr>
<td>GDT location</td>
<td>public GDT_EPR PROM GDT_EPRM TABLE_REG &lt;&gt;</td>
<td>TABLE GDT(location = GDT_EPRM)</td>
<td>The location of the GDT will be programmed into the GDT_EPRM location.</td>
</tr>
<tr>
<td>IDT location</td>
<td>public IDT_EPRM IDT_EPRM TABLE_REG &lt;&gt;</td>
<td>TABLE IDT(location = IDT_EPRM)</td>
<td>The location of the IDT will be programmed into the IDT_EPRM location.</td>
</tr>
<tr>
<td>RAM start</td>
<td>RAM_START equ 400H</td>
<td>memory (reserve = (0..3FFFH))</td>
<td>RAM_START is used as the ram destination for moving the tables. It must be excluded from the application's segment area.</td>
</tr>
<tr>
<td>Location of the application TSS in the GDT</td>
<td>TSS_INDEX EQU 10</td>
<td>TABLE GDT(ENTRY = (10: PROTECTED_MODE_TASK))</td>
<td>Put the descriptor of the application TSS in GDT entry 10.</td>
</tr>
<tr>
<td>EPROM size and location</td>
<td>size and location of the initialization code</td>
<td>SEGMENT startup.code (base = OFFFF0000H) ...memory (RANGE(ROM_AREA = ROM(x..y)))</td>
<td>Initialization code size must be less than 64K and resides at upper most 64K of the 4-GByte memory space.</td>
</tr>
</tbody>
</table>

10.11 MICROCODE UPDATE FACILITIES

The P6 family and later processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 10-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.
10.11.1 Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor’s signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0FH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 10.11.2 for details).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 10-7 provides a definition of the fields; Table 10-8 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the microcode update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The total size field of the microcode update header specifies the encrypted data size plus the header size; its value must be in multiples of 1024 bytes (1 KBytes). The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

NOTE
The optional extended signature table is supported starting with processor family 0FH, model 03H.

Table 10-7. Microcode Update Field Definitions

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header Version</td>
<td>0</td>
<td>4</td>
<td>Version number of the update header.</td>
</tr>
<tr>
<td>Update Revision</td>
<td>4</td>
<td>4</td>
<td>Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.</td>
</tr>
<tr>
<td>Date</td>
<td>8</td>
<td>4</td>
<td>Date of the update creation in binary format: mmddyyyy (e.g., 07/18/98 is 07181998H).</td>
</tr>
</tbody>
</table>
Table 10-7. Microcode Update Field Definitions (Contd.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Signature</td>
<td>12</td>
<td>4</td>
<td>Extended family, extended model, type, family, model, and stepping of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, type, family, model, and stepping of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.</td>
</tr>
<tr>
<td>Checksum</td>
<td>16</td>
<td>4</td>
<td>Checksum of update header and data. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all the DWORDs (including the extended Processor Signature Table) that comprise the microcode update result in 00000000H.</td>
</tr>
<tr>
<td>Loader Revision</td>
<td>20</td>
<td>4</td>
<td>Version number of the loader program needed to correctly load this update. The initial version is 00000001H.</td>
</tr>
<tr>
<td>Processor Flags</td>
<td>24</td>
<td>4</td>
<td>Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.</td>
</tr>
<tr>
<td>Data Size</td>
<td>28</td>
<td>4</td>
<td>Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).</td>
</tr>
<tr>
<td>Total Size</td>
<td>32</td>
<td>4</td>
<td>Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table. This value is always a multiple of 1024.</td>
</tr>
<tr>
<td>Reserved</td>
<td>36</td>
<td>12</td>
<td>Reserved fields for future expansion.</td>
</tr>
<tr>
<td>Update Data</td>
<td>48</td>
<td>Data Size or 2000</td>
<td>Update data.</td>
</tr>
<tr>
<td>Extended Signature Count</td>
<td>Data Size + 48</td>
<td>4</td>
<td>Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.</td>
</tr>
<tr>
<td>Extended Checksum</td>
<td>Data Size + 52</td>
<td>4</td>
<td>Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.</td>
</tr>
<tr>
<td>Reserved</td>
<td>Data Size + 56</td>
<td>12</td>
<td>Reserved fields.</td>
</tr>
</tbody>
</table>
Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.

Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform ID bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.

To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.

### Table 10-8. Microcode Update Format

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Signature[n]</td>
<td>Data Size + 68 + (n * 12)</td>
<td>4</td>
<td><em>Extended family, extended model, type, family, model, and stepping of processor that requires this particular update revision (e.g., 00000650H).</em> Each microcode update is designed specifically for a given extended family, extended model, type, family, model, and stepping of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.</td>
</tr>
<tr>
<td>Processor Flags[n]</td>
<td>Data Size + 72 + (n * 12)</td>
<td>4</td>
<td>Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform ID bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.</td>
</tr>
<tr>
<td>Checksum[n]</td>
<td>Data Size + 76 + (n * 12)</td>
<td>4</td>
<td>Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table. To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.</td>
</tr>
</tbody>
</table>

### Table 10-7. Microcode Update Field Definitions (Contd.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Offset (bytes)</th>
<th>Length (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header Version</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Update Revision</td>
<td>4</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Month: 8 Day: 8 Year: 16</td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Processor Signature (CPUID)</td>
<td>12</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>Res: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Family: 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Mode: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved: 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type: 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Family: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stepping: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>16</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Loader Revision</td>
<td>20</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Processor Flags</td>
<td>24</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Reserved (24 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Size</td>
<td>28</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>Total Size</td>
<td>32</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Reserved (12 Bytes)</td>
<td>36</td>
<td></td>
<td>36</td>
</tr>
</tbody>
</table>

10-30 Vol. 3A
**10.11.2 Optional Extended Signature Table**

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 10-9). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 10-10).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

### Table 10-9. Extended Processor Signature Table Header Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended Signature Count ‘n’</td>
<td>Data Size + 48</td>
</tr>
<tr>
<td>Extended Processor Signature Table Checksum</td>
<td>Data Size + 52</td>
</tr>
<tr>
<td>Reserved (12 Bytes)</td>
<td>Data Size + 56</td>
</tr>
</tbody>
</table>

### Table 10-10. Processor Signature Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Signature[n]</td>
<td>Data Size + 68 + (n * 12)</td>
</tr>
<tr>
<td>Processor Flags[n]</td>
<td>Data Size + 72 + (n * 12)</td>
</tr>
<tr>
<td>Checksum[n]</td>
<td>Data Size + 76 + (n * 12)</td>
</tr>
</tbody>
</table>
10.11.3 Processor Identification

Each microcode update is designed to be for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the BIOS to reject the update.

Example 10-5 shows how to check for a valid processor signature match between the processor and microcode update.

Example 10-5. Pseudo Code to Validate the Processor Signature

```pseudo
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion = 00000001h)
{
    // first check the ProcessorSignature field
    If (ProcessorSignature = Update.ProcessorSignature)
    Success

    // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[0].

        For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
        (ProcessorSignature ≠ Update.ProcessorSignature[N])); N++)
        {
            // if the loops ended when the iteration count is
            // less than the number of processor signatures in
            // the table, we have a match
            If (N < Update.ExtendedSignatureCount)
            Success
            Else
            Fail
        }
    }
    Else
    Fail
Else
Fail
```

10.11.4 Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32_PLATFORM_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header’s processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

<table>
<thead>
<tr>
<th>Register Name:</th>
<th>IA32_PLATFORM_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR Address:</td>
<td>017H</td>
</tr>
</tbody>
</table>

10-32 Vol. 3A
IA32_PLATFORM_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 10-6.

### Example 10-6. Pseudo Code Example of Processor Flags Test

Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion = 00000001h)
{
    If (Update.ProcessorFlags & Flag)
    {
        Load Update
    }
    Else
    {
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[N] and a match
        // on Update.ProcessorSignature[N] has already succeeded
        //
        If (Update.ProcessorFlags[n] & Flag)
        {
            Load Update
        }
    }
}

### 10.11.5 Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software’s responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a
unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 10-7 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = \(\frac{\text{Total Size}}{4}\), where the total size is a multiple of 1024 bytes (1 KBytes).

Example 10-7. Pseudo Code Example of Checksum Test

\[
\begin{align*}
N & \leftarrow 512 \\
\text{If} \ (\text{Update.DataSize} \neq 00000000H) \\
& \quad N \leftarrow \text{Update.TotalSize} / 4 \\
\text{ChkSum} & \leftarrow 0 \\
\text{For} \ (I \leftarrow 0; \ I < N; \ I++) \\
& \quad \text{ChkSum} \leftarrow \text{ChkSum} + \text{MicrocodeUpdate}[I] \\
\text{If} \ (\text{ChkSum} = 00000000H) \\
& \quad \text{Success} \\
& \quad \text{Else} \\
& \quad \text{Fail}
\end{align*}
\]

10.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a P6 family or later processors. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 10-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

Example 10-8. Assembly Code Example of Simple Microcode Update Loader

\[
\begin{align*}
\text{mov} \ ecx,79h & \quad ; \text{MSR to write in ECX} \\
\text{xor} \ eax,eax & \quad ; \text{clear EAX} \\
\text{xor} \ ebx,ebx & \quad ; \text{clear EBX} \\
\text{mov} \ ax,cs & \quad ; \text{Segment of microcode update} \\
\text{shl} \ eax,4 & \\
\text{mov} \ bx,\text{offset Update} & \quad ; \text{Offset of microcode update} \\
\text{add} \ eax,ebx & \quad ; \text{Linear Address of Update in EAX} \\
\text{add} \ eax,48d & \quad ; \text{Offset of the Update Data within the Update} \\
\text{xor} \ edx,edx & \quad ; \text{Zero in EDX} \\
\text{WRMSR} & \quad ; \text{microcode update trigger}
\end{align*}
\]

The loader shown in Example 10-8 assumes that \textit{update} is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:
• In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
• In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
• ECX contains 79H (address of IA32_BIOS_UPDT_TRIG).

Other requirements are:
• The addresses for the microcode update data must be in canonical form.
• If paging is enabled, the microcode update data must map that data as present.
• The microcode update data must start at a 16-byte aligned linear address.

10.11.6.1 Hard Resets in Update Loading
The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

10.11.6.2 Update in a Multiprocessor System
A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

10.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology
Intel Hyper-Threading Technology has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor supporting Intel Hyper-Threading Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g., a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

10.11.6.4 Update in a System Supporting Dual-Core Technology
Dual-core technology has implications on the loading of the microcode update. The microcode update facility is not shared between processor cores in the same physical package. The update must be loaded for each core in a physical processor.

If processor core supports Intel Hyper-Threading Technology, the guideline described in Section 10.11.6.3 also applies.

10.11.6.5 Update Loader Enhancements
The update loader presented in Section 10.11.6, "Microcode Update Loader," is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:
• BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.
• A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 10.11.7, "Update Signature and Verification."
• A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 10.11.5, "Microcode Update Checksum."
• A loader can provide power-on messages indicating successful loading of an update.

10.11.7 Update Signature and Verification

The P6 family and later processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32_BIOS_SIGN_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different stepings.

10.11.7.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 10-9.

Example 10-9. Assembly Code to Retrieve the Update Revision

```
MOV ECX, 08BH ;IA32_BIOS_SIGN_ID
XOR EAX, EAX ;clear EAX
XOR EDX, EDX ;clear EDX
WRMSR ;Load 0 to MSR at 8BH
MOV EAX, 1
cpuid
MOV ECX, 08BH ;IA32_BIOS_SIGN_ID
rdmsr ;Read Model Specific Register
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID Microcode Update Signature Register
MSR Address: 08BH Accessed as a Qword
Default Value: XXXX XXXX XXXX XXXXh
Access: Read/Write

The IA32_BIOS_SIGN_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 10-12).
10.11.7.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 10-10.

Example 10-10. Pseudo Code to Authenticate the Update

\[
Z \leftarrow \text{Obtain Update Revision from the Update Header to be authenticated}; \\
X \leftarrow \text{Obtain Current Update Signature from MSR 8BH}; \\
\text{If } (Z > X) \{
    \text{Load Update that is to be authenticated;}
    Y \leftarrow \text{Obtain New Signature from MSR 8BH};
    \text{If } (Z = Y)
        \text{Success}
    \text{Else}
        \text{Fail}
\text{Else}
    \text{Fail}
\]

Example 10-10 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

10.11.8 Optional Processor Microcode Update Specifications

This section an interface that an OEM-BIOS may provide to its client system software to manage processor microcode updates. System software may choose to build its own facility to manage microcode updates (e.g., similar to the facility described in Section 9.11.6) or rely on a facility provided by the BIOS to perform microcode updates.

Sections 10.11.8.1-10.11.8.9 describes an extension (Function 0D042H) to the real mode INT 15H service. INT 15H 0D042H function is one of several alternatives that a BIOS may choose to implement microcode update facility and offer to its client application (e.g., an OS). Other alternative microcode update facility that BIOS can choose are dependent on platform-specific capabilities, including the Capsule Update mechanism from the UEFI specification (www.uefi.org). In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.
10.11.8.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor stepping, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM’s system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

NOTES

For IA-32 processors starting with family 0FH and model 03H and Intel 64 processors, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16-KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 10-11) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

Example 10-11. Pseudo Code, Checks Required Prior to Loading an Update

For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version = 00000001H)
    }
If ((Update.ProcessorSignature = Processor Signature) && (Update.ProcessorFlags & Platform Bits))
{
    Load Update.UpdateData into the Processor;
    Verify update was correctly loaded into the processor
    Go on to next processor
    Break;
}
Else If (Update.TotalSize > (Update.DataSize + 48))
{
    N ← 0
    While (N < Update.ExtendedSignatureCount)
    {
        If ((Update.ProcessorSignature[N] = Processor Signature) && (Update.ProcessorFlags[N] & Platform Bits))
        {
            Load Update.UpdateData into the Processor;
            Verify update correctly loaded into the processor
            Go on to next processor
            Break;
        }
        N ← N + 1
    }
    I ← I + (Update.TotalSize / 2048)
    If ((Update.TotalSize MOD 2048) = 0)
    {
        I ← I + 1
    }
}

NOTES
The platform Id bits in IA32_PLATFORM_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

10.11.8.2 Responsibilities of the Calling Program
This section of the document lists the responsibilities of a calling program using the interface specifications to load microcode update(s) into BIOS NVRAM.
• The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
• The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
• It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
• The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an
older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.

- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don’t exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 10-12 represents a calling program.

Example 10-12. INT 15 DO42 Calling Program Pseudo-code

```
// We must be in real mode
// If the system is not in Real mode exit
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
// If no Intel processors exist that can be updated exit
// Detect the presence of the Intel microcode update extensions
// If the BIOS fails the PresenceTest exit
// If the APIC is enabled, see if any other processors are out there
// Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e.,Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform Id Bits
    If current processor cannot be updated
        exit
}
// Determine the number of unique update blocks needed for this system
// NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
        (we have a unique update in the database for this processor))
    {
        Checksum the update from the database;
        If Checksum fails
            exit
        NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}
// Do we have enough update slots for all CPUs?
```
If there are more blocks required to support the unique processor steppings than update blocks provided by the BIOS exit

//
// Do we need any update blocks at all? If not, we are done
//
If (NumBlocks = 0)
exit

// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
//
// Load each Update
//
Issue the WriteUpdate function

If (STORAGE_FULL) returned
{
    Display Error -- BIOS is not managing NVRAM appropriately
    exit
}

If (INVALID_REVISION) returned
{
    Display Message: More recent update already loaded in NVRAM for this stepping
    continue
}

If any other error returned
{
    Display Diagnostic
    exit
}

//
// Verify the update was loaded correctly
//
Issue the ReadUpdate function

If an error occurred
{
    Display Diagnostic
    exit
}
//
// Compare the Update read to that written
//
If (Update read ≠ Update written)
{
    Display Diagnostic
    exit
}

I ← I + (size of microcode update / 2048)

//
// Enable Update Loading, and inform user
10.11.8.3 Microcode Update Functions

Table 10-13 defines the processor microcode update functions that implementations of INT 15H 0D042H must support.

Table 10-13. Microcode Update Functions

<table>
<thead>
<tr>
<th>Microcode Update Function</th>
<th>Function Number</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presence test</td>
<td>00H</td>
<td>Returns information about the supported functions.</td>
<td>Required</td>
</tr>
<tr>
<td>Write update data</td>
<td>01H</td>
<td>Writes one of the update data areas (slots).</td>
<td>Required</td>
</tr>
<tr>
<td>Update control</td>
<td>02H</td>
<td>Globally controls the loading of updates.</td>
<td>Required</td>
</tr>
<tr>
<td>Read update data</td>
<td>03H</td>
<td>Reads one of the update data areas (slots).</td>
<td>Required</td>
</tr>
</tbody>
</table>

10.11.8.4 INT 15H-based Interface

If an OEM-BIOS is implementing INT 15H 0D042H interface and offer to its client, the BIOS should allow additional microcode updates to be added to system flash.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX, and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 10.11.8.9, "Return Codes."

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

10.11.8.5 Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 10-14 lists the parameters and return codes for the function.

Table 10-14. Parameters for the Presence Test

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>Function Code 0D042H</td>
</tr>
<tr>
<td>BL</td>
<td>Sub-function 00H - Presence test</td>
</tr>
<tr>
<td>CF</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>AH</td>
<td>Return Code</td>
</tr>
<tr>
<td>AL</td>
<td>OEM Error</td>
</tr>
<tr>
<td>EBX</td>
<td>Signature Part 1 'INTE' - Part one of the signature</td>
</tr>
<tr>
<td>ECX</td>
<td>Signature Part 2 'LPEP' - Part two of the signature</td>
</tr>
<tr>
<td>EDX</td>
<td>Loader Version Version number of the microcode update loader</td>
</tr>
</tbody>
</table>

// Issue the Update Control function with Task = Enable.
In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

10.11.8.6 Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 10-15 lists the parameters and return codes for the function.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>Function Code 0D042H</td>
</tr>
<tr>
<td>BL</td>
<td>Sub-function 01H - Write update</td>
</tr>
<tr>
<td>ES:DI</td>
<td>Update Address</td>
</tr>
<tr>
<td></td>
<td>Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or...</td>
</tr>
<tr>
<td></td>
<td>Real Mode pointer to the Intel Update structure. This buffer is 64 KBytes in length if the processor supports a variable-size microcode update.</td>
</tr>
<tr>
<td>CX</td>
<td>Scratch Pad1</td>
</tr>
<tr>
<td>DX</td>
<td>Scratch Pad2</td>
</tr>
<tr>
<td>SI</td>
<td>Scratch Pad3</td>
</tr>
<tr>
<td>SS:SP</td>
<td>Stack pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag</td>
</tr>
<tr>
<td></td>
<td>Carry Set - Failure - AH Contains status</td>
</tr>
<tr>
<td></td>
<td>Carry Clear - All return values valid</td>
</tr>
<tr>
<td>AH</td>
<td>Return Code</td>
</tr>
<tr>
<td>AL</td>
<td>OEM Error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Codes (see Table 10-19 for code definitions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
</tr>
<tr>
<td>NOT_IMPLEMENTED</td>
</tr>
<tr>
<td>WRITE_FAILURE</td>
</tr>
<tr>
<td>ERASE_FAILURE</td>
</tr>
<tr>
<td>READ_FAILURE</td>
</tr>
</tbody>
</table>

Table 10-14. Parameters for the Presence Test (Contd.)

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>Update Count</td>
</tr>
<tr>
<td></td>
<td>Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates</td>
</tr>
</tbody>
</table>

Return Codes (see Table 10-19 for code definitions)

<table>
<thead>
<tr>
<th>Return Codes (see Table 10-19 for code definitions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
</tr>
<tr>
<td>NOT_IMPLEMENTED</td>
</tr>
<tr>
<td>WRITE_FAILURE</td>
</tr>
<tr>
<td>ERASE_FAILURE</td>
</tr>
<tr>
<td>READ_FAILURE</td>
</tr>
</tbody>
</table>

Table 10-15. Parameters for the Write Update Data Function
PROCESSOR MANAGEMENT AND INITIALIZATION

Table 10-15. Parameters for the Write Update Data Function (Contd.)

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE_FULL</td>
<td>The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.</td>
</tr>
<tr>
<td>CPU_NOT_PRESENT</td>
<td>The processor stepping does not currently exist in the system.</td>
</tr>
<tr>
<td>INVALID_HEADER</td>
<td>The update header contains a header or loader version that is not recognized by the BIOS.</td>
</tr>
<tr>
<td>INVALID_HEADER_CS</td>
<td>The update does not checksum correctly.</td>
</tr>
<tr>
<td>SECURITY_FAILURE</td>
<td>The processor rejected the update.</td>
</tr>
<tr>
<td>INVALID_REVISION</td>
<td>The same or more recent revision of the update exists in the storage device.</td>
</tr>
</tbody>
</table>

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 10.11.6, “Microcode Update Loader.” This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 10-8 and Figure 10-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.
Figure 10-8. Microcode Update Write Operation Flow [1]
10.11.8.7 Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 10-16 lists the parameters and return codes for the function.
This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 10-17 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

**Table 10-16. Parameters for the Control Update Sub-function**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>Function Code</td>
<td>0D042H</td>
</tr>
<tr>
<td>BL</td>
<td>Sub-function</td>
<td>02H - Control update</td>
</tr>
<tr>
<td>BH</td>
<td>Task</td>
<td>See the description below.</td>
</tr>
<tr>
<td>CX</td>
<td>Scratch Pad1</td>
<td>Real mode segment of 64 KBytes of RAM block</td>
</tr>
<tr>
<td>DX</td>
<td>Scratch Pad2</td>
<td>Real mode segment of 64 KBytes of RAM block</td>
</tr>
<tr>
<td>SI</td>
<td>Scratch Pad3</td>
<td>Real mode segment of 64 KBytes of RAM block</td>
</tr>
<tr>
<td>SS:SP</td>
<td>Stack pointer</td>
<td>32 kilobytes of stack minimum</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
<th>Carry Set - Failure - AH contains status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Carry Clear - All return values valid.</td>
</tr>
<tr>
<td>AH</td>
<td>Return Code</td>
<td>Status of the call</td>
</tr>
<tr>
<td>AL</td>
<td>OEM Error</td>
<td>Additional OEM Information.</td>
</tr>
<tr>
<td>BL</td>
<td>Update Status</td>
<td>Either enable or disable indicator</td>
</tr>
</tbody>
</table>

**Return Codes (see Table 10-19 for code definitions)**

- **SUCCESS**: Function completed successfully.
- **READ_FAILURE**: A failure occurred because of the inability to read the storage device.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

**10.11.8.8 Function 03H—Read Microcode Update Data**

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 10-18 lists the parameters and return codes.

**Table 10-17. Mnemonic Values**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>1</td>
<td>Enable the Update loading at initialization time.</td>
</tr>
<tr>
<td>Query</td>
<td>2</td>
<td>Determine the current state of the update control without changing its status.</td>
</tr>
</tbody>
</table>

**Table 10-18. Parameters for the Read Microcode Update Data Function**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>Function Code</td>
<td>0D042H</td>
</tr>
<tr>
<td>BL</td>
<td>Sub-function</td>
<td>03H - Read Update</td>
</tr>
<tr>
<td>ES:DI</td>
<td>Buffer Address</td>
<td>Real Mode pointer to the Intel Update structure that will be written with the binary data</td>
</tr>
</tbody>
</table>
The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple blocks, then the BIOS must return Failure with the NOT_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0xFFFFFFFF after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0xFFFFFFFF. The BIOS is responsible for translating this information into the header provided by this function.

### 10.11.8.9 Return Codes

After the call has been made, the return codes listed in Table 10-19 are available in the AH register.
### Table 10-19. Return Code Definitions

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>00H</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>NOT_IMPLEMENTED</td>
<td>86H</td>
<td>The function is not implemented.</td>
</tr>
<tr>
<td>ERASE_FAILURE</td>
<td>90H</td>
<td>A failure because of the inability to erase the storage device.</td>
</tr>
<tr>
<td>WRITE_FAILURE</td>
<td>91H</td>
<td>A failure because of the inability to write the storage device.</td>
</tr>
<tr>
<td>READ_FAILURE</td>
<td>92H</td>
<td>A failure because of the inability to read the storage device.</td>
</tr>
<tr>
<td>STORAGE_FULL</td>
<td>93H</td>
<td>The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.</td>
</tr>
<tr>
<td>CPU_NOT_PRESENT</td>
<td>94H</td>
<td>The processor stepping does not currently exist in the system.</td>
</tr>
<tr>
<td>INVALID_HEADER</td>
<td>95H</td>
<td>The update header contains a header or loader version that is not recognized by the BIOS.</td>
</tr>
<tr>
<td>INVALID_HEADER_CS</td>
<td>96H</td>
<td>The update does not checksum correctly.</td>
</tr>
<tr>
<td>SECURITY_FAILURE</td>
<td>97H</td>
<td>The update was rejected by the processor.</td>
</tr>
<tr>
<td>INVALID_REVISION</td>
<td>98H</td>
<td>The same or more recent revision of the update exists in the storage device.</td>
</tr>
<tr>
<td>UPDATE_NUM_INVALID</td>
<td>99H</td>
<td>The update number exceeds the maximum number of update blocks implemented by the BIOS.</td>
</tr>
<tr>
<td>NOT_EMPTY</td>
<td>9AH</td>
<td>The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The specified block is not a header block and is not empty.</td>
</tr>
</tbody>
</table>
CHAPTER 11
ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

The Advanced Programmable Interrupt Controller (APIC), referred to in the following sections as the local APIC, was introduced into the IA-32 processors with the Pentium processor (see Section 23.27, “Advanced Programmable Interrupt Controller (APIC)”) and is included in the P6 family, Pentium 4, Intel Xeon processors, and other more recent Intel 64 and IA-32 processor families (see Section 11.4.2, “Presence of the Local APIC”). The local APIC performs two primary functions for the processor:

- It receives interrupts from the processor’s interrupt pins, from internal sources and from an external I/O APIC (or other external interrupt controller). It sends these to the processor core for handling.
- In multiple processor (MP) systems, it sends and receives interprocessor interrupt (IPI) messages to and from other logical processors on the system bus. IPI messages can be used to distribute interrupts among the processors in the system or to execute system wide functions (such as, booting up processors or distributing work among a group of processors).

The external I/O APIC is part of Intel’s system chipset. Its primary function is to receive external interrupt events from the system and its associated I/O devices and relay them to the local APIC as interrupt messages. In MP systems, the I/O APIC also provides a mechanism for distributing external interrupts to the local APICs of selected processors or groups of processors on the system bus.

This chapter provides a description of the local APIC and its programming interface. It also provides an overview of the interface between the local APIC and the I/O APIC. Contact Intel for detailed information about the I/O APIC.

When a local APIC has sent an interrupt to its processor core for handling, the processor uses the interrupt and exception handling mechanism described in Chapter 6, “Interrupt and Exception Handling.” See Section 6.1, “Interrupt and Exception Overview,” for an introduction to interrupt and exception handling.

11.1 LOCAL AND I/O APIC OVERVIEW

Each local APIC consists of a set of APIC registers (see Table 11-1) and associated hardware that control the delivery of interrupts to the processor core and the generation of IPI messages. The APIC registers are memory mapped and can be read and written to using the MOV instruction.

Local APICs can receive interrupts from the following sources:

- **Locally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected directly to the processor’s local interrupt pins (LINT0 and LINT1). The I/O devices may also be connected to an 8259-type interrupt controller that is in turn connected to the processor through one of the local interrupt pins.

- **Externally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected to the interrupt input pins of an I/O APIC. Interrupts are sent as I/O interrupt messages from the I/O APIC to one or more of the processors in the system.

- **Inter-processor interrupts (IPIs)** — An Intel 64 or IA-32 processor can use the IPI mechanism to interrupt another processor or group of processors on the system bus. IPIs are used for software self-interrupts, interrupt forwarding, or preemptive scheduling.

- **APIC timer generated interrupts** — The local APIC timer can be programmed to send a local interrupt to its associated processor when a programmed count is reached (see Section 11.5.4, “APIC Timer”).

- **Performance monitoring counter interrupts** — P6 family, Pentium 4, and Intel Xeon processors provide the ability to send an interrupt to its associated processor when a performance-monitoring counter overflows (see Section 20.6.3.5.8, “Generating an Interrupt on Overflow”).

- **Thermal Sensor interrupts** — Pentium 4 and Intel Xeon processors provide the ability to send an interrupt to themselves when the internal thermal sensor has been tripped (see Section 15.8.2, “Thermal Monitor”).

- **APIC internal error interrupts** — When an error condition is recognized within the local APIC (such as an attempt to access an unimplemented register), the APIC can be programmed to send an interrupt to its associated processor (see Section 11.5.3, “Error Handling”).
Of these interrupt sources: the processor’s LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and the internal APIC error detector are referred to as **local interrupt sources**. Upon receiving a signal from a local interrupt source, the local APIC delivers the interrupt to the processor core using an interrupt delivery protocol that has been set up through a group of APIC registers called the **local vector table** or **LVT** (see Section 11.5.1, “Local Vector Table”). A separate entry is provided in the local vector table for each local interrupt source, which allows a specific interrupt delivery protocol to be set up for each source. For example, if the LINT1 pin is going to be used as an NMI pin, the LINT1 entry in the local vector table can be set up to deliver an interrupt with vector number 2 (NMI interrupt) to the processor core.

The local APIC handles interrupts from the other two interrupt sources (externally connected I/O devices and IPIs) through its IPI message handling facilities. A processor can generate IPIs by programming the interrupt command register (ICR) in its local APIC (see Section 11.6.1, “Interrupt Command Register (ICR)”)

IPIs can be sent to other processors in the system or to the originating processor (self-interrupts). When the target processor receives an IPI message, its local APIC handles the message automatically (using information included in the message such as vector number and trigger mode). See Section 11.6, “Issuing Interprocessor Interrupts,” for a detailed explanation of the local APIC’s IPI message delivery and acceptance mechanism.

The local APIC can also receive interrupts from externally connected devices through the I/O APIC (see Figure 11-1). The I/O APIC is responsible for receiving interrupts generated by system hardware and I/O devices and forwarding them to the local APIC as interrupt messages.

![Figure 11-1. Relationship of Local APIC and I/O APIC In Single-Processor Systems](image)

Individual pins on the I/O APIC can be programmed to generate a specific interrupt vector when asserted. The I/O APIC also has a “virtual wire mode” that allows it to communicate with a standard 8259A-style external interrupt controller. Note that the local APIC can be disabled (see Section 11.4.3, “Enabling or Disabling the Local APIC”). This allows an associated processor core to receive interrupts directly from an 8259A interrupt controller.

Both the local APIC and the I/O APIC are designed to operate in MP systems (see Figures 11-2 and 11-3). Each local APIC handles interrupts from the I/O APIC, IPIs from processors on the system bus, and self-generated interrupts. Interrupts can also be delivered to the individual processors through the local interrupt pins; however, this mechanism is commonly not used in MP systems.
The IPI mechanism is typically used in MP systems to send fixed interrupts (interrupts for a specific vector number) and special-purpose interrupts to processors on the system bus. For example, a local APIC can use an IPI to forward a fixed interrupt to another processor for servicing. Special-purpose IPIs (including NMI, INIT, SMI, and SIPI IPIs) allow one or more processors on the system bus to perform system-wide boot-up and control functions.

The following sections focus on the local APIC and its implementation in the Pentium 4, Intel Xeon, and P6 family processors. In these sections, the terms “local APIC” and “I/O APIC” refer to local and I/O APICs used with the P6 family processors and to local and I/O xAPICs used with the Pentium 4 and Intel Xeon processors (see Section 11.3, “The Intel® 82489DX External APIC, the APIC, the xAPIC, and the X2APIC”).

### 11.2 SYSTEM BUS VS. APIC BUS

For the P6 family and Pentium processors, the I/O APIC and local APICs communicate through the 3-wire inter-APIC bus (see Figure 11-3). Local APICs also use the APIC bus to send and receive IPIs. The APIC bus and its messages are invisible to software and are not classed as architectural.
Beginning with the Pentium 4 and Intel Xeon processors, the I/O APIC and local APICs (using the xAPIC architecture) communicate through the system bus (see Figure 11-2). The I/O APIC sends interrupt requests to the processors on the system bus through bridge hardware that is part of the Intel chipset. The bridge hardware generates the interrupt messages that go to the local APICs. IPIs between local APICs are transmitted directly on the system bus.


The local APIC in the P6 family and Pentium processors is an architectural subset of the Intel® 82489DX external APIC. See Section 23.27.1, "Software Visible Differences Between the Local APIC and the 82489DX."

The APIC architecture used in the Pentium 4 and Intel Xeon processors (called the xAPIC architecture) is an extension of the APIC architecture found in the P6 family processors. The primary difference between the APIC and xAPIC architectures is that with the xAPIC architecture, the local APICs and the I/O APIC communicate through the system bus. With the APIC architecture, they communicate through the APIC bus (see Section 11.2, "System Bus Vs. APIC Bus"). Also, some APIC architectural features have been extended and/or modified in the xAPIC architecture. These extensions and modifications are described in Section 11.4 through Section 11.10.

The basic operating mode of the xAPIC is xAPIC mode. The x2APIC architecture is an extension of the xAPIC architecture, primarily to increase processor addressability. The x2APIC architecture provides backward compatibility to the xAPIC architecture and forward extendability for future Intel platform innovations. These extensions and modifications are supported by a new mode of execution (x2APIC mode) are detailed in Section 11.12.

11.4 LOCAL APIC

The following sections describe the architecture of the local APIC and how to detect it, identify it, and determine its status. Descriptions of how to program the local APIC are given in Section 11.5.1, "Local Vector Table," and Section 11.6.1, "Interrupt Command Register (ICR)."

11.4.1 The Local APIC Block Diagram

Figure 11-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor’s physical address space with an initial starting address of FEE00000H. For correct APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 12.3, "Methods of Caching Available."

In MP system configurations, the APIC registers for Intel 64 or IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 11.4.5, "Relocating the Local APIC Registers," describes how to relocate the base address for APIC registers.

On processors supporting x2APIC architecture (indicated by CPUID.01H:ECX[21] = 1), the local APIC supports operation both in xAPIC mode and (if enabled by software) in x2APIC mode. x2APIC mode provides extended processor addressability (see Section 11.12).

NOTE

For P6 family, Pentium 4, and Intel Xeon processors, the APIC handles all memory accesses to addresses within the 4-KByte APIC register space internally and no external bus cycles are produced. For the Pentium processors with an on-chip APIC, bus cycles are produced for accesses to the APIC register space. Thus, for software intended to run on Pentium processors, system software should explicitly not map the APIC register space to regular system memory. Doing so can result in an invalid opcode exception (#UD) being generated or unpredictable execution.
Table 11-1 shows how the APIC registers are mapped into the 4-KByte APIC register space. Registers are 32 bits, 64 bits, or 256 bits in width; all are aligned on 128-bit boundaries. All 32-bit registers should be accessed using 128-bit aligned 32-bit loads or stores. Some processors may support loads and stores of less than 32 bits to some of the APIC registers. This is model specific behavior and is not guaranteed to work on all processors. Any FP/MMX/SSE access to an APIC register, or any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed. This undefined behavior could include hangs, incorrect results or unexpected exceptions, including machine checks, and may vary between implementations. Wider registers (64-bit or 256-bit) must be accessed using multiple 32-bit loads or stores, with all accesses being 128-bit aligned.

The local APIC registers listed in Table 11-1 are not MSRs. The only MSR associated with the programming of the local APIC is the IA32_APIC_BASE MSR (see Section 11.4.3, “Enabling or Disabling the Local APIC”).

**NOTE**

In processors based on Nehalem\(^1\) microarchitecture, the Local APIC ID Register is no longer Read/Write; it is Read Only.
### Table 11-1. Local APIC Register Address Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Register Name</th>
<th>Software Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEE0 0000H</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>FEE0 0010H</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>FEE0 0020H</td>
<td>Local APIC ID Register</td>
<td>Read/Write.</td>
</tr>
<tr>
<td>FEE0 0030H</td>
<td>Local APIC Version Register</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0040H</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>FEE0 0050H</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>FEE0 0060H</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>FEE0 0070H</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>FEE0 0080H</td>
<td>Task Priority Register (TPR)</td>
<td>Read/Write.</td>
</tr>
<tr>
<td>FEE0 0090H</td>
<td>Arbitration Priority Register(^1) (APR)</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 00A0H</td>
<td>Processor Priority Register (PPR)</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 00B0H</td>
<td>EOI Register</td>
<td>Write Only.</td>
</tr>
<tr>
<td>FEE0 00C0H</td>
<td>Remote Read Register(^1) (RRD)</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 00D0H</td>
<td>Logical Destination Register</td>
<td>Read/Write.</td>
</tr>
<tr>
<td>FEE0 00E0H</td>
<td>Destination Format Register</td>
<td>Read/Write (see Section 11.6.2.2).</td>
</tr>
<tr>
<td>FEE0 00F0H</td>
<td>Spurious Interrupt Vector Register</td>
<td>Read/Write (see Section 11.9).</td>
</tr>
<tr>
<td>FEE0 0100H</td>
<td>In-Service Register (ISR); bits 31:0</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0110H</td>
<td>In-Service Register (ISR); bits 63:32</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0120H</td>
<td>In-Service Register (ISR); bits 95:64</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0130H</td>
<td>In-Service Register (ISR); bits 127:96</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0140H</td>
<td>In-Service Register (ISR); bits 159:128</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0150H</td>
<td>In-Service Register (ISR); bits 191:160</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0160H</td>
<td>In-Service Register (ISR); bits 223:192</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0170H</td>
<td>In-Service Register (ISR); bits 255:224</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0180H</td>
<td>Trigger Mode Register (TMR); bits 31:0</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0190H</td>
<td>Trigger Mode Register (TMR); bits 63:32</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 01A0H</td>
<td>Trigger Mode Register (TMR); bits 95:64</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 01B0H</td>
<td>Trigger Mode Register (TMR); bits 127:96</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 01C0H</td>
<td>Trigger Mode Register (TMR); bits 159:128</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 01D0H</td>
<td>Trigger Mode Register (TMR); bits 191:160</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 01E0H</td>
<td>Trigger Mode Register (TMR); bits 223:192</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 01F0H</td>
<td>Trigger Mode Register (TMR); bits 255:224</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0200H</td>
<td>Interrupt Request Register (IRR); bits 31:0</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0210H</td>
<td>Interrupt Request Register (IRR); bits 63:32</td>
<td>Read Only.</td>
</tr>
<tr>
<td>FEE0 0220H</td>
<td>Interrupt Request Register (IRR); bits 95:64</td>
<td>Read Only.</td>
</tr>
</tbody>
</table>

1. See Table 2-1, “CPUID Signature Values of DisplayFamily_DisplayModel,” on page 1, and Section 2.8, “MSRs In Processors Based on Nehalem Microarchitecture,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, to determine which processors are based on Nehalem microarchitecture.
11.4.2 Presence of the Local APIC

Beginning with the P6 family processors, the presence or absence of an on-chip local APIC can be detected using the CPUID instruction. When the CPUID instruction is executed with a source operand of 1 in the EAX register, bit 9 of the CPUID feature flags returned in the EDX register indicates the presence (set) or absence (clear) of a local APIC.

11.4.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH; see Figure 11-5):
— When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 11.4.2, “Presence of the Local APIC”) is also set to 0.

— When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus loses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).

— For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality, if software guarantees no interrupt will be sent to the APIC as IA32_APIC_BASE[11] is cleared.

— When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 11.4.7.1, “Local APIC State After Power-Up or Reset.”

2. Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 11-23):

— If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 11-23). The state of the local APIC when in this software-disabled state is described in Section 11.4.7.2, “Local APIC State After It Has Been Software Disabled.”

— When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or reset to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, Intel® Processor Trace, the thermal sensor, and/or the internal APIC error detector).

11.4.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Figure 11-5). MSR bit functions are described below:

• **BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 9.4, “Multiple-Processor (MP) Initialization.” Following a power-up or reset, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).

• **APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 11.4.3, “Enabling or Disabling the Local APIC”). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future Intel 64 or IA-32 processors.

• **APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-KByte boundary. Following a power-up or reset, the field is set to FEE0 0000H.

• Bits 0 through 7, bits 9 and 10, and bits MAXPHYADDR2 through 63 in the IA32_APIC_BASE MSR are reserved.

---

2. The MAXPHYADDR is 36 bits for processors that do not support CPUID leaf 80000008H, or indicated by CPUID.80000008H:EAX[bits 7:0] for processors that support CPUID leaf 80000008H.
11.4.5 Relocating the Local APIC Registers

The Pentium 4, Intel Xeon, and P6 family processors permit the starting address of the APIC registers to be relocated from FEE00000H to another physical address by modifying the value in the base address field of the IA32_APIC_BASE MSR. This extension of the APIC architecture is provided to help resolve conflicts with memory maps of existing systems and to allow individual processors in an MP system to map their APIC registers to different locations in physical memory.

11.4.6 Local APIC ID

At power up, system hardware assigns a unique APIC ID to each local APIC on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for P6 family and Pentium processors). The hardware assigned APIC ID is based on system topology and includes encoding for socket position and cluster information (see Figure 9-2 and Section 9.9.1, "Hierarchical Mapping of Shared Resources").

In MP systems, the local APIC ID is also used as a processor ID by the BIOS and the operating system. Some processors permit software to modify the APIC ID. However, the ability of software to modify the APIC ID is processor model specific. Because of this, operating system software should avoid writing to the local APIC ID register. The value returned by bits 31-24 of the EBX register (when the CPUID instruction is executed with a source operand value of 1 in the EAX register) is always the Initial APIC ID (determined by the platform initialization). This is true even if software has changed the value in the Local APIC ID register.

The processor receives the hardware assigned APIC ID (or Initial APIC ID) by sampling pins A11# and A12# and pins BR0# through BR3# (for the Pentium 4, Intel Xeon, and P6 family processors) and pins BE0# through BE3# (for the Pentium processor). The APIC ID latched from these pins is stored in the APIC ID field of the local APIC ID register (see Figure 11-6), and is used as the Initial APIC ID for the processor.
For the P6 family and Pentium processors, the local APIC ID field in the local APIC ID register is 4 bits. Encodings 0H through EH can be used to uniquely identify 15 different processors connected to the APIC bus. For the Pentium 4 and Intel Xeon processors, the xAPIC specification extends the local APIC ID field to 8 bits. These can be used to identify up to 255 processors in the system.

11.4.7 Local APIC State

The following sections describe the state of the local APIC and its registers following a power-up or reset, after the local APIC has been software disabled, following an INIT reset, and following an INIT-deassert message. x2APIC will introduce 32-bit ID; see Section 11.12.

11.4.7.1 Local APIC State After Power-Up or Reset

Following a power-up or reset of the processor, the state of local APIC and its registers are as follows:

- The following registers are reset to all 0s.
  - IRR, ISR, TMR, ICR, LDR, and TPR.
  - Timer initial count and timer current count registers.
  - Divide configuration register.
- The DFR register is reset to all 1s.
- The LVT register is reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The local APIC ID register is set to a unique APIC ID. (Pentium and P6 family processors only). The Arb ID register is set to the value in the APIC ID register.
- The spurious-interrupt vector register is initialized to 000000FFH. By setting bit 8 to 0, software disables the local APIC.
- If the processor is the only processor in the system or it is the BSP in an MP system (see Section 9.4.1, “BSP and AP Processors”); the local APIC will respond normally to INIT and NMI messages, to INIT# signals and to STPCLK# signals. If the processor is in an MP system and has been designated as an AP; the local APIC will respond the same as for the BSP. In addition, it will respond to SIPI messages. For P6 family processors only, an AP will not respond to a STPCLK# signal.

11.4.7.2 Local APIC State After It Has Been Software Disabled

When the APIC software enable/disable flag in the spurious interrupt vector register has been explicitly cleared (as opposed to being cleared during a power up or reset), the local APIC is temporarily disabled (see Section 11.4.3, “Enabling or Disabling the Local APIC”). The operation and response of a local APIC while in this software-disabled state is as follows:

- The local APIC will respond normally to INIT, NMI, SMI, and SIPI messages.
- Pending interrupts in the IRR and ISR registers are held and require masking or handling by the CPU.
- The local APIC can still issue IPIs. It is software's responsibility to avoid issuing IPIs through the IPI mechanism and the ICR register if sending interrupts through this mechanism is not desired.
- The reception of any interrupt or transmission of any IPIs that are in progress when the local APIC is disabled are completed before the local APIC enters the software-disabled state.
- The mask bits for all the LVT entries are set. Attempts to reset these bits will be ignored.
- (For Pentium and P6 family processors) The local APIC continues to listen to all bus messages in order to keep its arbitration ID synchronized with the rest of the system.

11.4.7.3 Local APIC State After an INIT Reset (“Wait-for-SIPI” State)

An INIT reset of the processor can be initiated in either of two ways:
• By asserting the processor’s INIT# pin.
• By sending the processor an INIT IPI (an IPI with the delivery mode set to INIT).

Upon receiving an INIT through either of these mechanisms, the processor responds by beginning the initialization process of the processor core and the local APIC. The state of the local APIC following an INIT reset is the same as it is after a power-up or hardware reset, except that the APIC ID and arbitration ID registers are not affected. This state is also referred to at the “wait-for-SIPI” state (see also: Section 9.4.2, “MP Initialization Protocol Requirements and Restrictions”).

### 11.4.7.4 Local APIC State After It Receives an INIT-Deassert IPI

Only the Pentium and P6 family processors support the INIT-deassert IPI. An INIT-deassert IPI has no affect on the state of the APIC, other than to reload the arbitration ID register with the value in the APIC ID register.

### 11.4.8 Local APIC Version Register

The local APIC contains a hardwired version register. Software can use this register to identify the APIC version (see Figure 11-7). In addition, the register specifies the number of entries in the local vector table (LVT) for a specific implementation.

The fields in the local APIC version register are as follows:

**Version**

The version numbers of the local APIC:

- 0XH: 8248DX discrete APIC.
- 10H - 15H: Integrated APIC.
- Other values reserved.

**Max LVT Entry**

Shows the number of LVT entries minus 1. For the Pentium 4 and Intel Xeon processors (which have 6 LVT entries), the value returned in the Max LVT field is 5; for the P6 family processors (which have 5 LVT entries), the value returned is 4; for the Pentium processor (which has 4 LVT entries), the value returned is 3. For processors based on the Nehalem microarchitecture (which has 7 LVT entries) and onward, the value returned is 6.

**Suppress EOI-broadcasts**

Indicates whether software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register; see Section 11.8.5 and Section 11.9.

![Figure 11-7. Local APIC Version Register](image-url)
11.5 HANDLING LOCAL INTERRUPTS

The following sections describe facilities that are provided in the local APIC for handling local interrupts. These include: the processor’s LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, Intel Processor Trace, the thermal sensor, and the internal APIC error detector. Local interrupt handling facilities include: the LVT, the error status register (ESR), the divide configuration register (DCR), and the initial count and current count registers.

11.5.1 Local Vector Table

The local vector table (LVT) allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers (see Figure 11-8), one for each local interrupt:

- **LVT CMCI Register (FEE0 02F0H)** — Specifies interrupt delivery when an overflow condition of corrected machine check error count reaching a threshold value occurred in a machine check bank supporting CMCI (see Section 16.5.1, “CMCI Local APIC Interface”).

- **LVT Timer Register (FEE0 0320H)** — Specifies interrupt delivery when the APIC timer signals an interrupt (see Section 11.5.4, “APIC Timer”).

- **LVT Thermal Monitor Register (FEE0 0330H)** — Specifies interrupt delivery when the thermal sensor generates an interrupt (see Section 15.8.2, “Thermal Monitor”). This LVT entry is implementation specific, not architectural. If implemented, it will always be at base address FEE0 0330H.

- **LVT Performance Counter Register (FEE0 0340H)** — Specifies interrupt delivery when a performance counter generates an interrupt on overflow (see Section 20.6.3.5.8, “Generating an Interrupt on Overflow”) or when Intel PT signals a ToPA PMI (see Section 33.2.7.2). This LVT entry is implementation specific, not architectural. If implemented, it is not guaranteed to be at base address FEE0 0340H.

- **LVT LINT0 Register (FEE0 0350H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT0 pin.

- **LVT LINT1 Register (FEE0 0360H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT1 pin.

- **LVT Error Register (FEE0 0370H)** — Specifies interrupt delivery when the APIC detects an internal error (see Section 11.5.3, “Error Handling”).

The LVT performance counter register and its associated interrupt were introduced in the P6 processors and are also present in the Pentium 4 and Intel Xeon processors. The LVT thermal monitor register and its associated interrupt were introduced in the Pentium 4 and Intel Xeon processors. The LVT CMCI register and its associated interrupt were introduced in the Intel Xeon 5500 processors.

As shown in Figure 11-8, some of these fields and flags are not available (and reserved) for some entries.
The setup information that can be specified in the registers of the LVT table is as follows:

**Vector**
- Interrupt vector number.

**Delivery Mode**
- Specifies the type of interrupt to be sent to the processor. Some delivery modes will only operate as intended when used in conjunction with a specific trigger mode. The allowable delivery modes are as follows:
  - **000 (Fixed)** Delivers the interrupt specified in the vector field.
  - **010 (SMI)** Delivers an SMI interrupt to the processor core through the processor’s local SMI signal path. When using this delivery mode, the vector field should be set to 00H for future compatibility.

† (Pentium 4 and Intel Xeon processors.) When a performance monitoring counters interrupt is generated, the mask bit for its associated LVT entry is set.

Value After Reset: 0001 0000H
ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

100 (NMI) Delivers an NMI interrupt to the processor. The vector information is ignored.

101 (INIT) Delivers an INIT request to the processor core, which causes the processor to perform an INIT. When using this delivery mode, the vector field should be set to 00H for future compatibility. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

110 Reserved; not supported for any LVT register.

111 (ExtINT) Causes the processor to respond to the interrupt as if the interrupt originated in an externally connected (8259A-compatible) interrupt controller. A special INTA bus cycle corresponding to ExtINT, is routed to the external controller. The external controller is expected to supply the vector information. The APIC architecture supports only one ExtINT source in a system, usually contained in the compatibility bridge. Only one processor in the system should have an LVT entry configured to use the ExtINT delivery mode. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

Delivery Status (Read Only)
Indicates the interrupt delivery status, as follows:

0 (Idle) There is currently no activity for this interrupt source, or the previous interrupt from this source was delivered to the processor core and accepted.

1 (Send Pending) Indicates that an interrupt from this source has been delivered to the processor core but has not yet been accepted (see Section 11.5.5, “Local Interrupt Acceptance”).

Interrupt Input Pin Polarity
Specifies the polarity of the corresponding interrupt pin: (0) active high or (1) active low.

Remote IRR Flag (Read Only)
For fixed mode, level-triggered interrupts; this flag is set when the local APIC accepts the interrupt for servicing and is reset when an EOI command is received from the processor. The meaning of this flag is undefined for edge-triggered interrupts and other delivery modes.

Trigger Mode
Selects the trigger mode for the local LINT0 and LINT1 pins: (0) edge sensitive and (1) level sensitive. This flag is only used when the delivery mode is Fixed. When the delivery mode is NMI, SMI, or INIT, the trigger mode is always edge sensitive. When the delivery mode is ExtINT, the trigger mode is always level sensitive. The timer and error interrupts are always treated as edge sensitive.

If the local APIC is not used in conjunction with an I/O APIC and fixed delivery mode is selected; the Pentium 4, Intel Xeon, and P6 family processors will always use level-sensitive triggering, regardless if edge-sensitive triggering is selected.

Software should always set the trigger mode in the LVT LINT1 register to 0 (edge sensitive). Level-sensitive interrupts are not supported for LINT1.

Mask
Interrupt mask: (0) enables reception of the interrupt and (1) inhibits reception of the interrupt. When the local APIC handles a performance-monitoring counters interrupt, it automatically sets the mask flag in the LVT performance counter register. This flag is set to 1 on reset. It can be cleared only by software.

Timer Mode
Bits 18:17 selects the timer mode (see Section 11.5.4):
(00b) one-shot mode using a count-down value,
(01b) periodic mode reloading a count-down value,
(10b) TSC-Deadline mode using absolute target value in IA32_TSC_DEADLINE MSR (see Section 11.5.4.1),
(11b) is reserved.
11.5.2 Valid Interrupt Vectors

The Intel 64 and IA-32 architectures define 256 vector numbers, ranging from 0 through 255 (see Section 6.2, "Exception and Interrupt Vectors"). Local and I/O APICs support 240 of these vectors (in the range of 16 to 255) as valid interrupts.

When an interrupt vector in the range of 0 to 15 is sent or received through the local APIC, the APIC indicates an illegal vector in its Error Status Register (see Section 11.5.3, "Error Handling"). The Intel 64 and IA-32 architectures reserve vectors 16 through 31 for predefined interrupts, exceptions, and Intel-reserved encodings (see Table 6-1). However, the local APIC does not treat vectors in this range as illegal.

When an illegal vector value (0 to 15) is written to an LVT entry and the delivery mode is Fixed (bits 8-11 equal 0), the APIC may signal an illegal vector error, without regard to whether the mask bit is set or whether an interrupt is actually seen on the input.

11.5.3 Error Handling

The local APIC records errors detected during interrupt handling in the error status register (ESR). The format of the ESR is given in Figure 11-9; it contains the following flags:

- **Bit 0: Send Checksum Error.**
  Set when the local APIC detects a checksum error for a message that it sent on the APIC bus. Used only on P6 family and Pentium processors.

- **Bit 1: Receive Checksum Error.**
  Set when the local APIC detects a checksum error for a message that it received on the APIC bus. Used only on P6 family and Pentium processors.

- **Bit 2: Send Accept Error.**
  Set when the local APIC detects that a message it sent was not accepted by any APIC on the APIC bus. Used only on P6 family and Pentium processors.

- **Bit 3: Receive Accept Error.**
  Set when the local APIC detects that the message it received was not accepted by any APIC on the APIC bus, including itself. Used only on P6 family and Pentium processors.

**Figure 11-9. Error Status Register (ESR)**
• Bit 4: Redirectable IPI.
  Set when the local APIC detects an attempt to send an IPI with the lowest-priority delivery mode and the local
  APIC does not support the sending of such IPIs. This bit is used on some Intel Core and Intel Xeon processors.
  As noted in Section 11.6.2, the ability of a processor to send a lowest-priority IPI is model-specific and should
  be avoided.

• Bit 5: Send Illegal Vector.
  Set when the local APIC detects an illegal vector (one in the range 0 to 15) in the message that it is sending.
  This occurs as the result of a write to the ICR (in both xAPIC and x2APIC modes) or to SELF IPI register (x2APIC
  mode only) with an illegal vector.

  If the local APIC does not support the sending of lowest-priority IPIs and software writes the ICR to send a
  lowest-priority IPI with an illegal vector, the local APIC sets only the “redirectable IPI” error bit. The interrupt is
  not processed and hence the “Send Illegal Vector” bit is not set in the ESR.

• Bit 6: Receive Illegal Vector.
  Set when the local APIC detects an illegal vector (one in the range 0 to 15) in an interrupt message it receives
  or in an interrupt generated locally from the local vector table or via a self IPI. Such interrupts are not delivered
  to the processor; the local APIC will never set an IRR bit in the range 0 to 15.

• Bit 7: Illegal Register Address
  Set when the local APIC is in xAPIC mode and software attempts to access a register that is reserved in the
  processor’s local-APIC register-address space; see Table 10-1. (The local-APIC register-address space
  comprises the 4 KBytes at the physical address specified in the IA32_APIC_BASE MSR.) Used only on Intel
  Core, Intel Atom, Pentium 4, Intel Xeon, and P6 family processors.

  In x2APIC mode, software accesses the APIC registers using the RDMSR and WRMSR instructions. Use of one
  of these instructions to access a reserved register cause a general-protection exception (see Section
  10.12.1.3). They do not set the “Illegal Register Access” bit in the ESR.

The ESR is a write/read register. Before attempt to read from the ESR, software should first write to it. (The value
written does not affect the values read subsequently; only zero may be written in x2APIC mode.) This write clears
any previously logged errors and updates the ESR with any errors detected since the last write to the ESR. This
write also rearms the APIC error interrupt triggering mechanism.

The LVT Error Register (see Section 11.5.1) allows specification of the vector of the interrupt to be delivered to the
processor core when APIC error is detected. The register also provides a means of masking an APIC-error interrupt.
This masking only prevents delivery of APIC-error interrupts; the APIC continues to record errors in the ESR.

11.5.4   APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations.
This timer is set up by programming four registers: the divide configuration register (see Figure 11-10), the initial-
count and current-count registers (see Figure 11-11), and the LVT timer register (see Figure 11-8).

If CPUID.06H:EAX.ARAT[bit 2] = 1, the processor’s APIC timer runs at a constant rate regardless of P-state transi-
tions and it continues to run at the same rate in deep C-states.

If CPUID.06H:EAX.ARAT[bit 2] = 0 or if CPUID 06H is not supported, the APIC timer may temporarily stop while the
processor is in deep C-states or during transitions caused by Enhanced Intel SpeedStep® Technology.

The APIC timer frequency will be the processor’s bus clock or core crystal clock frequency (when TSC/core crystal
clock ratio is enumerated in CPUID leaf 0x15) divided by the value specified in the divide configuration register.

The timer can be configured through the timer LVT entry for one-shot or periodic operation. In one-shot mode, the
timer is started by programming its initial-count register. The initial count value is then copied into the current-
count register and count-down begins. After the timer reaches zero, a timer interrupt is generated and the timer
remains at its 0 value until reprogrammed.

In periodic mode, the timer is started by writing to the initial-count register (as in one-shot mode), and the value
written is copied into the current-count register, which counts down. The current-count register is automatically
reloaded from the initial-count register when the count reaches 0 and a timer interrupt is generated, and the count-
down is repeated. If during the count-down process the initial-count register is set, counting will restart, using the
new initial-count value. The initial-count register is a read-write register; the current-count register is read only.
A write of 0 to the initial-count register effectively stops the local APIC timer, in both one-shot and periodic mode. The LVT timer register determines the vector number that is delivered to the processor with the timer interrupt that is generated when the timer count reaches zero. The mask flag in the LVT timer register can be used to mask the timer interrupt.

**NOTE**

Changing the mode of the APIC timer (from one-shot to periodic or vice versa) by writing to the timer LVT entry does not start the timer. To start the timer, it is necessary to write to the initial-count register as described above.

### 11.5.4.1 TSC-Deadline Mode

The mode of operation of the local-APIC timer is determined by the LVT Timer Register. Specifically:

- If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, the mode is determined by bit 17 of the register.
- If CPUID.01H:ECX.TSC_Deadline[bit 24] = 1, the mode is determined by bits 18:17. See Figure 11-8. (If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, bit 18 of the register is reserved.)

The supported timer modes are given in Table 11-2. The three modes of the local APIC timer are mutually exclusive.

**Table 11-2. Local APIC Timer Modes**

<table>
<thead>
<tr>
<th>LVT Bits [18:17]</th>
<th>Timer Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>One-shot mode, program count-down value in an initial-count register. See Section 11.5.4</td>
</tr>
<tr>
<td>01b</td>
<td>Periodic mode, program interval value in an initial-count register. See Section 11.5.4</td>
</tr>
<tr>
<td>10b</td>
<td>TSC-Deadline mode, program target value in IA32_TSC_DEADLINE MSR.</td>
</tr>
<tr>
<td>11b</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
TSC-deadline mode allows software to use the local APIC timer to signal an interrupt at an absolute time. In TSC-deadline mode, writes to the initial-count register are ignored; and current-count register always reads 0. Instead, timer behavior is controlled using the IA32_TSC_DEADLINE MSR.

The IA32_TSC_DEADLINE MSR (MSR address 6E0H) is a per-logical processor MSR that specifies the time at which a timer interrupt should occur. Writing a non-zero 64-bit value into IA32_TSC_DEADLINE arms the timer. An interrupt is generated when the logical processor's time-stamp counter equals or exceeds the target value in the IA32_TSC_DEADLINE MSR. Then, when the timer generates an interrupt, it disarms itself and clears the IA32_TSC_DEADLINE MSR. Thus, each write to the IA32_TSC_DEADLINE MSR generates at most one timer interrupt.

In TSC-deadline mode, writing 0 to the IA32_TSC_DEADLINE MSR disarms the local-APIC timer. Transitioning between TSC-deadline mode and other timer modes also disarms the timer.

The hardware reset value of the IA32_TSC_DEADLINE MSR is 0. In other timer modes (LVT bit 18 = 0), the IA32_TSC_DEADLINE MSR reads zero and writes are ignored.

Software can configure the TSC-deadline timer to deliver a single interrupt using the following algorithm:

1. Detect support for TSC-deadline mode by verifying CPUID.1:ECX.24 = 1.
2. Select the TSC-deadline mode by programming bits 18:17 of the LVT Timer register with 10b.
3. Program the IA32_TSC_DEADLINE MSR with the target TSC value at which the timer interrupt is desired. This causes the processor to arm the timer.
4. The processor generates a timer interrupt when the value of time-stamp counter is greater than or equal to that of IA32_TSC_DEADLINE. It then disarms the timer and clears the IA32_TSC_DEADLINE MSR. (Both the time-stamp counter and the IA32_TSC_DEADLINE MSR are 64-bit unsigned integers.)
5. Software can re-arm the timer by repeating step 3.

The following are usage guidelines for TSC-deadline mode:

- Writes to the IA32_TSC_DEADLINE MSR are not serialized. Therefore, system software should not use WRMSR to the IA32_TSC_DEADLINE MSR as a serialization instruction. Read and write accesses to the IA32_TSC_DEADLINE and other MSR registers will occur in program order.
- Software can disarm the timer at any time by writing 0 to the IA32_TSC_DEADLINE MSR.
- If timer is armed, software can change the deadline (forward or backward) by writing a new value to the IA32_TSC_DEADLINE MSR.
- If software disarms the timer or postpones the deadline, race conditions may result in the delivery of a spurious timer interrupt. Software is expected to detect such spurious interrupts by checking the current value of the time-stamp counter to confirm that the interrupt was desired.
- In xAPIC mode (in which the local-APIC registers are memory-mapped), software must order the memory-mapped write to the LVT entry that enables TSC-deadline mode and any subsequent WRMSR to the IA32_TSC_DEADLINE MSR. Software can assure proper ordering by executing the MFENCE instruction after the memory-mapped write and before any WRMSR. (In x2APIC mode, the WRMSR instruction is used to write to the LVT entry. The processor ensures the ordering of this write and any subsequent WRMSR to the deadline; no fencing is required.)

11.5.5 Local Interrupt Acceptance

When a local interrupt is sent to the processor core, it is subject to the acceptance criteria specified in the interrupt acceptance flow chart in Figure 11-17. If the interrupt is accepted, it is logged into the IRR register and handled by the processor according to its priority (see Section 11.8.4, “Interrupt Acceptance for Fixed Interrupts”). If the interrupt is not accepted, it is sent back to the local APIC and retried.

---

3. If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 26 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.
11.6 ISSUING INTERPROCESSOR INTERRUPTS

The following sections describe the local APIC facilities that are provided for issuing interprocessor interrupts (IPIs) from software. The primary local APIC facility for issuing IPIs is the interrupt command register (ICR). The ICR can be used for the following functions:

- To send an interrupt to another processor.
- To allow a processor to forward an interrupt that it received but did not service to another processor for servicing.
- To direct the processor to interrupt itself (perform a self interrupt).
- To deliver special IPIs, such as the start-up IPI (SIPI) message, to other processors.

Interrupts generated with this facility are delivered to the other processors in the system through the system bus (for Pentium 4 and Intel Xeon processors) or the APIC bus (for P6 family and Pentium processors). The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.

11.6.1 Interrupt Command Register (ICR)

The interrupt command register (ICR) is a 64-bit\(^4\) local APIC register (see Figure 11-12) that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other processors in the system.

Address: FEE0 0300H (0 - 31)
FEE0 0310H (32 - 63)
Value after Reset: 0H

NOTE:
1. The ability of a processor to send Lowest Priority IPI is model specific.

\(^4\) In XAPIC mode the ICR is addressed as two 32-bit registers, ICR_LOW (FFE0 0300H) and ICR_HIGH (FFE0 0310H). In x2APIC mode, the ICR uses MSR 830H.
To send an IPI, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. (All fields of the ICR are read-write by software with the exception of the delivery status field, which is read-only.) The act of writing to the low doubleword of the ICR causes the IPI to be sent.

The ICR consists of the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>The vector number of the interrupt being sent.</td>
</tr>
<tr>
<td>Delivery Mode</td>
<td>Specifies the type of IPI to be sent. This field is also known as the IPI message type field.</td>
</tr>
<tr>
<td>000 (Fixed)</td>
<td>Delivers the interrupt specified in the vector field to the target processor or processors.</td>
</tr>
<tr>
<td>001 (Lowest Priority)</td>
<td>Same as fixed mode, except that the interrupt is delivered to the processor executing at the lowest priority among the set of processors specified in the destination field. The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.</td>
</tr>
<tr>
<td>010 (SMI)</td>
<td>Delivers an SMI interrupt to the target processor or processors. The vector field must be programmed to 00H for future compatibility.</td>
</tr>
<tr>
<td>011 (Reserved)</td>
<td></td>
</tr>
<tr>
<td>100 (NMI)</td>
<td>Delivers an NMI interrupt to the target processor or processors. The vector information is ignored.</td>
</tr>
<tr>
<td>101 (INIT)</td>
<td>Delivers an INIT request to the target processor or processors, which causes them to perform an INIT. As a result of this IPI message, all the target processors perform an INIT. The vector field must be programmed to 00H for future compatibility.</td>
</tr>
<tr>
<td>101 (INIT Level De-assert)</td>
<td>(Not supported in the Pentium 4 and Intel Xeon processors.) Sends a synchronization message to all the local APICs in the system to set their arbitration IDs (stored in their Arb ID registers) to the values of their APIC IDs (see Section 11.7, “System and APIC Bus Arbitration”). For this delivery mode, the level flag must be set to 0 and trigger mode flag to 1. This IPI is sent to all processors, regardless of the value in the destination field or the destination shorthand field; however, software should specify the “all including self” shorthand.</td>
</tr>
<tr>
<td>110 (Start-Up)</td>
<td>Sends a special “start-up” IPI (called a SIPI) to the target processor or processors. The vector typically points to a start-up routine that is part of the BIOS boot-strap code (see Section 9.4, “Multiple-Processor (MP) Initialization”). IPIs sent with this delivery mode are not automatically retried if the source APIC is unable to deliver it. It is up to the software to determine if the SIPI was not successfully delivered and to reissue the SIPI if necessary.</td>
</tr>
</tbody>
</table>

**Destination Mode** Selects either physical (0) or logical (1) destination mode (see Section 11.6.2, “Determining IPI Destination”).

**Delivery Status (Read Only)** Indicates the IPI delivery status, as follows:

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Idle)</td>
<td>Indicates that this local APIC has completed sending any previous IPIs.</td>
</tr>
<tr>
<td>1 (Send Pending)</td>
<td>Indicates that this local APIC has not completed sending the last IPI.</td>
</tr>
</tbody>
</table>

**Level** For the INIT level de-assert delivery mode this flag must be set to 0; for all other delivery modes it must be set to 1. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 1.)
**Trigger Mode**

Selects the trigger mode when using the INIT level de-assert delivery mode: edge (0) or level (1). It is ignored for all other delivery modes. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 0.)

**Destination Shorthand**

Indicates whether a shorthand notation is used to specify the destination of the interrupt and, if so, which shorthand is used. Destination shorthands are used in place of the 8-bit destination field, and can be sent by software using a single write to the low doubleword of the ICR. Shorthands are defined for the following cases: software self interrupt, IPIs to all processors in the system including the sender, IPIs to all processors in the system excluding the sender.

00: (No Shorthand)

The destination is specified in the destination field.

01: (Self)

The issuing APIC is the one and only destination of the IPI. This destination shorthand allows software to interrupt the processor on which it is executing. An APIC implementation is free to deliver the self-interrupt message internally or to issue the message to the bus and “snoop” it as with any other IPI message.

10: (All Including Self)

The IPI is sent to all processors in the system including the processor sending the IPI. The APIC will broadcast an IPI message with the destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors.

11: (All Excluding Self)

The IPI is sent to all processors in a system with the exception of the processor sending the IPI. The APIC broadcasts a message with the physical destination mode and destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors. Support for this destination shorthand in conjunction with the lowest-priority delivery mode is model specific. For Pentium 4 and Intel Xeon processors, when this shorthand is used together with lowest priority delivery mode, the IPI may be redirected back to the issuing processor.

**Destination**

Specifies the target processor or processors. This field is only used when the destination shorthand field is set to 00B. If the destination mode is set to physical, then bits 56 through 59 contain the APIC ID of the target processor for Pentium and P6 family processors and bits 56 through 63 contain the APIC ID of the target processor for Pentium 4 and Intel Xeon processors. If the destination mode is set to logical, the interpretation of the 8-bit destination field depends on the settings of the DFR and LDR registers of the local APICs in all the processors in the system (see Section 11.6.2, "Determining IPI Destination").

Not all combinations of options for the ICR are valid. Table 11-3 shows the valid combinations for the fields in the ICR for the Pentium 4 and Intel Xeon processors; Table 11-4 shows the valid combinations for the fields in the ICR for the P6 family processors. Also note that the lower half of the ICR may not be preserved over transitions to the deepest C-States.

ICR operation in x2APIC mode is discussed in Section 11.12.9.

### Table 11-3. Valid Combinations for Pentium 4 and Intel Xeon Processors Local xAPIC Interrupt Command Register

<table>
<thead>
<tr>
<th>Destination Shorthand</th>
<th>Valid/Invalid</th>
<th>Trigger Mode</th>
<th>Delivery Mode</th>
<th>Destination Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Shorthand</td>
<td>Valid</td>
<td>Edge</td>
<td>All Modes¹</td>
<td>Physical or Logical</td>
</tr>
<tr>
<td>No Shorthand</td>
<td>Invalid²</td>
<td>Level</td>
<td>All Modes</td>
<td>Physical or Logical</td>
</tr>
<tr>
<td>Self</td>
<td>Valid</td>
<td>Edge</td>
<td>Fixed</td>
<td>X³</td>
</tr>
<tr>
<td>Self</td>
<td>Invalid²</td>
<td>Level</td>
<td>Fixed</td>
<td>X</td>
</tr>
<tr>
<td>Self</td>
<td>Invalid</td>
<td>X</td>
<td>Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All Including Self</td>
<td>Valid</td>
<td>Edge</td>
<td>Fixed</td>
<td>X</td>
</tr>
<tr>
<td>All Including Self</td>
<td>Invalid²</td>
<td>Level</td>
<td>Fixed</td>
<td>X</td>
</tr>
</tbody>
</table>
### Table 11-3. Valid Combinations for Pentium 4 and Intel Xeon Processors Local xAPIC Interrupt Command Register

<table>
<thead>
<tr>
<th>Destination Shorthand</th>
<th>Valid/Invalid</th>
<th>Trigger Mode</th>
<th>Delivery Mode</th>
<th>Destination Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Including Self</td>
<td>Invalid</td>
<td>X</td>
<td>Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All Excluding Self</td>
<td>Valid</td>
<td>Edge</td>
<td>Fixed, Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All Excluding Self</td>
<td>Invalid²</td>
<td>Level</td>
<td>Fixed, Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The ability of a processor to send a lowest priority IPI is model specific.
2. Treated as edge triggered if level bit is set to 1, otherwise ignored.
3. Treated as edge triggered when Level bit is set to 1; treated as “INIT Level Deassert” message when level bit is set to 0 (deassert).
4. X means the setting is ignored.
5. The behavior of the APIC is undefined.

### Table 11-4. Valid Combinations for the P6 Family Processor Local APIC Interrupt Command Register

<table>
<thead>
<tr>
<th>Destination Shorthand</th>
<th>Valid/Invalid</th>
<th>Trigger Mode</th>
<th>Delivery Mode</th>
<th>Destination Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Shorthand</td>
<td>Valid</td>
<td>Edge</td>
<td>All Modes</td>
<td>Physical or Logical</td>
</tr>
<tr>
<td>No Shorthand</td>
<td>Valid²</td>
<td>Level</td>
<td>Fixed, Lowest Priority¹, NMI</td>
<td>Physical or Logical</td>
</tr>
<tr>
<td>No Shorthand</td>
<td>Valid³</td>
<td>Level</td>
<td>INIT</td>
<td>Physical or Logical</td>
</tr>
<tr>
<td>Self</td>
<td>Valid</td>
<td>Edge</td>
<td>Fixed</td>
<td>X⁴</td>
</tr>
<tr>
<td>Self</td>
<td>Valid²</td>
<td>Level</td>
<td>Fixed</td>
<td>X</td>
</tr>
<tr>
<td>Self</td>
<td>Invalidate⁵</td>
<td>X</td>
<td>Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All including Self</td>
<td>Valid</td>
<td>Edge</td>
<td>Fixed</td>
<td>X</td>
</tr>
<tr>
<td>All including Self</td>
<td>Invalidate⁵</td>
<td>X</td>
<td>Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All excluding Self</td>
<td>Valid</td>
<td>Edge</td>
<td>All Modes¹</td>
<td>X</td>
</tr>
<tr>
<td>All excluding Self</td>
<td>Invalidate⁵</td>
<td>X</td>
<td>Lowest Priority, NMI, INIT, SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All excluding Self</td>
<td>Valid²</td>
<td>Level</td>
<td>Fixed, Lowest Priority¹, NMI</td>
<td>X</td>
</tr>
<tr>
<td>All excluding Self</td>
<td>Invalidate⁵</td>
<td>Level</td>
<td>SMI, Start-Up</td>
<td>X</td>
</tr>
<tr>
<td>All excluding Self</td>
<td>Invalid³</td>
<td>Level</td>
<td>INIT</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>Invalidate⁵</td>
<td>Level</td>
<td>SMI, Start-Up</td>
<td>X</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The ability of a processor to send a lowest priority IPI is model specific.
2. Treated as edge triggered if level bit is set to 1, otherwise ignored.
3. X means the setting is ignored.
4. The behavior of the APIC is undefined.

### 11.6.2 Determining IPI Destination

The destination of an IPI³ can be one, all, or a subset (group) of the processors on the system bus. The sender of the IPI specifies the destination of an IPI with the following APIC registers and fields within the registers:

- **ICR Register** — The following fields in the ICR register are used to specify the destination of an IPI.

5. Determination of IPI destinations in x2APIC mode is discussed in Section 10.12.10.
— **Destination Mode** — Selects one of two destination modes (physical or logical).

— **Destination Field** — In physical destination mode, used to specify the APIC ID of the destination processor; in logical destination mode, used to specify a message destination address (MDA) that can be used to select specific processors in clusters.

— **Destination Shorthand** — A quick method of specifying all processors, all excluding self, or self as the destination.

— **Delivery mode, Lowest Priority** — Architecturally specifies that a lowest-priority arbitration mechanism be used to select a destination processor from a specified group of processors. The ability of a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.

**Local destination register (LDR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.

**Destination format register (DFR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.

How the ICR, LDR, and DFR are used to select an IPI destination depends on the destination mode used: physical, logical, broadcast/self, or lowest-priority delivery mode. These destination modes are described in the following sections.

### 11.6.2.1 Physical Destination Mode

In physical destination mode, the destination processor is specified by its local APIC ID (see Section 11.4.6, "Local APIC ID"). For Pentium 4 and Intel Xeon processors, either a single destination (local APIC IDs 00H through FEH) or a broadcast to all APICs (the APIC ID is FFH) may be specified in physical destination mode.

A broadcast IPI (bits 28-31 of the MDA are 1's) or I/O subsystem initiated interrupt with lowest priority delivery mode is not supported in physical destination mode and must not be configured by software. Also, for any non-broadcast IPI or I/O subsystem initiated interrupt with lowest priority delivery mode, software must ensure that APICs defined in the interrupt address are present and enabled to receive interrupts.

For the P6 family and Pentium processors, a single destination is specified in physical destination mode with a local APIC ID of 0H through 0EH, allowing up to 15 local APICs to be addressed on the APIC bus. A broadcast to all local APICs is specified with 0FH.

**NOTE**

The number of local APICs that can be addressed on the system bus may be restricted by hardware.

### 11.6.2.2 Logical Destination Mode

In logical destination mode, IPI destination is specified using an 8-bit message destination address (MDA), which is entered in the destination field of the ICR. Upon receiving an IPI message that was sent using logical destination mode, a local APIC compares the MDA in the message with the values in its LDR and DFR to determine if it should accept and handle the IPI. For both configurations of logical destination mode, when combined with lowest priority delivery mode, software is responsible for ensuring that all of the local APICs included in or addressed by the IPI or I/O subsystem interrupt are present and enabled to receive the interrupt.

Figure 11-13 shows the layout of the logical destination register (LDR). The 8-bit logical APIC ID field in this register is used to create an identifier that can be compared with the MDA.

**NOTE**

The logical APIC ID should not be confused with the local APIC ID that is contained in the local APIC ID register.
Figure 11-14 shows the layout of the destination format register (DFR). The 4-bit model field in this register selects one of two models (flat or cluster) that can be used to interpret the MDA when using logical destination mode.

1. **Flat Model** — This model is selected by programming DFR bits 28 through 31 to 1111. Here, a unique logical APIC ID can be established for up to 8 local APICs by setting a different bit in the logical APIC ID field of the LDR for each local APIC. A group of local APICs can then be selected by setting one or more bits in the MDA.

   Each local APIC performs a bit-wise AND of the MDA and its logical APIC ID. If a true condition (non-zero) is detected, the local APIC accepts the IPI message. A broadcast to all APICs is achieved by setting the MDA to 1s.

2. **Cluster Model** — This model is selected by programming DFR bits 28 through 31 to 0000. This model supports two basic destination schemes: flat cluster and hierarchical cluster.

   The flat cluster destination model is only supported for P6 family and Pentium processors. Using this model, all APICs are assumed to be connected through the APIC bus. Bits 60 through 63 of the MDA contains the encoded address of the destination cluster and bits 56 through 59 identify up to four local APICs within the cluster (each bit is assigned to one local APIC in the cluster, as in the flat connection model). To identify one or more local APICs, bits 60 through 63 of the MDA are compared with bits 28 through 31 of the LDR to determine if a local APIC is part of the cluster. Bits 56 through 59 of the MDA are compared with Bits 24 through 27 of the LDR to identify a local APICs within the cluster.

   Sets of processors within a cluster can be specified by writing the target cluster address in bits 60 through 63 of the MDA and setting selected bits in bits 56 through 59 of the MDA, corresponding to the chosen members of the cluster. In this mode, 15 clusters (with cluster addresses of 0 through 14) each having 4 local APICs can be specified in the message. For the P6 and Pentium processor’s local APICs, however, the APIC arbitration ID supports only 15 APIC agents. Therefore, the total number of processors and their local APICs supported in this mode is limited to 15. Broadcast to all local APICs is achieved by setting all destination bits to one. This guarantees a match on all clusters and selects all APICs in each cluster. A broadcast IPI or I/O subsystem broadcast interrupt with lowest priority delivery mode is not supported in cluster mode and must not be configured by software.

   The hierarchical cluster destination model can be used with Pentium 4, Intel Xeon, P6 family, or Pentium processors. With this model, a hierarchical network can be created by connecting different flat clusters via independent system or APIC buses. This scheme requires a cluster manager within each cluster, which is responsible for handling message passing between system or APIC buses. One cluster contains up to 4 agents. Thus 15 cluster managers, each with 4 agents, can form a network of up to 60 APIC agents. Note that hierar-
Advanced Programmable Interrupt Controller (APIC)

chical APIC networks requires a special cluster manager device, which is not part of the local or the I/O APIC units.

**NOTES**

All processors that have their APIC software enabled (using the spurious vector enable/disable bit) must have their DFRs (Destination Format Registers) programmed identically.

The default mode for DFR is flat mode. If you are using cluster mode, DFRs must be programmed before the APIC is software enabled. Since some chipsets do not accurately track a system view of the logical mode, program DFRs as soon as possible after starting the processor.

11.6.2.3 Broadcast/Self Delivery Mode

The destination shorthand field of the ICR allows the delivery mode to be by-passed in favor of broadcasting the IPI to all the processors on the system bus and/or back to itself (see Section 11.6.1, "Interrupt Command Register (ICR)"). Three destination shorthands are supported: self, all excluding self, and all including self. The destination mode is ignored when a destination shorthand is used.

11.6.2.4 Lowest Priority Delivery Mode

With lowest priority delivery mode, the ICR is programmed to send an IPI to several processors on the system bus, using the logical or shorthand destination mechanism for selecting the processor. The selected processors then arbitrate with one another over the system bus or the APIC bus, with the lowest-priority processor accepting the IPI.

For systems based on the Intel Xeon processor, the chipset bus controller accepts messages from the I/O APIC agents in the system and directs interrupts to the processors on the system bus. When using the lowest priority delivery mode, the chipset chooses a target processor to receive the interrupt out of the set of possible targets. The Pentium 4 processor provides a special bus cycle on the system bus that informs the chipset of the current task priority for each logical processor in the system. The chipset saves this information and uses it to choose the lowest priority processor when an interrupt is received.

For systems based on P6 family processors, the processor priority used in lowest-priority arbitration is contained in the arbitration priority register (APR) in each local APIC. Figure 11-15 shows the layout of the APR.

![Arbitration Priority Register (APR)](image)

The APR value is computed as follows:

\[
\text{IF } (\text{TPR}[7:4] \geq \text{IRRV}[7:4]) \text{ AND } (\text{TPR}[7:4] > \text{ISRV}[7:4]) \\
\text{THEN} \\
\hspace{1cm} \text{APR}[7:0] \leftarrow \text{TPR}[7:0] \\
\text{ELSE} \\
\hspace{1cm} \text{APR}[7:4] \leftarrow \max(\text{TPR}[7:4] \text{ AND ISRV}[7:4], \text{IRRV}[7:4]) \\
\hspace{1cm} \text{APR}[3:0] \leftarrow 0.
\]

Here, the TPR value is the task priority value in the TPR (see Figure 11-18), the IRRV value is the vector number for the highest priority bit that is set in the IRR (see Figure 11-20) or 00H (if no IRR bit is set), and the ISRV value is the vector number for the highest priority bit that is set in the ISR (see Figure 11-20). Following arbitration
among the destination processors, the processor with the lowest value in its APR handles the IPI and the other processors ignore it. (P6 family and Pentium processors.) For these processors, if a focus processor exists, it may accept the interrupt, regardless of its priority. A processor is said to be the focus of an interrupt if it is currently servicing that interrupt or if it has a pending request for that interrupt. For Intel Xeon processors, the concept of a focus processor is not supported.

In operating systems that use the lowest priority delivery mode but do not update the TPR, the TPR information saved in the chipset will potentially cause the interrupt to be always delivered to the same processor from the logical set. This behavior is functionally backward compatible with the P6 family processor but may result in unexpected performance implications.

11.6.3 IPI Delivery and Acceptance

When the low double-word of the ICR is written to, the local APIC creates an IPI message from the information contained in the ICR and sends the message out on the system bus (Pentium 4 and Intel Xeon processors) or the APIC bus (P6 family and Pentium processors). The manner in which these IPIs are handled after being issued is described in Section 11.8, "Handling Interrupts."

11.7 SYSTEM AND APIC BUS ARBITRATION

When several local APICs and the I/O APIC are sending IPI and interrupt messages on the system bus (or APIC bus), the order in which the messages are sent and handled is determined through bus arbitration.

For the Pentium 4 and Intel Xeon processors, the local and I/O APICs use the arbitration mechanism defined for the system bus to determine the order in which IPIs are handled. This mechanism is non-architectural and cannot be controlled by software.

For the P6 family and Pentium processors, the local and I/O APICs use an APIC-based arbitration mechanism to determine the order in which IPIs are handled. Here, each local APIC is given an arbitration priority of from 0 to 15, which the I/O APIC uses during arbitration to determine which local APIC should be given access to the APIC bus. The local APIC with the highest arbitration priority always wins bus access. Upon completion of an arbitration round, the winning local APIC lowers its arbitration priority to 0 and the losing local APICs each raise theirs by 1.

The current arbitration priority for a local APIC is stored in a 4-bit, software-transparent arbitration ID (Arb ID) register. During reset, this register is initialized to the APIC ID number (stored in the local APIC ID register). The INIT level-deassert IPI, which is issued with an ICR command, can be used to resynchronize the arbitration priorities of the local APICs by resetting Arb ID register of each agent to its current APIC ID value. (The Pentium 4 and Intel Xeon processors do not implement the Arb ID register.)

Section 11.10, "APIC Bus Message Passing Mechanism and Protocol (P6 Family, Pentium Processors)," describes the APIC bus arbitration protocols and bus message formats, while Section 11.6.1, "Interrupt Command Register (ICR)," describes the INIT level de-assert IPI message.

Note that except for the SIPI IPI (see Section 11.6.1, "Interrupt Command Register (ICR)"), all bus messages that fail to be delivered to their specified destination or destinations are automatically retried. Software should avoid situations in which IPIs are sent to disabled or nonexistent local APICs, causing the messages to be resent repeatedly. Additionally, interrupt sources that target the APIC should be masked or changed to no longer target the APIC.

11.8 HANDLING INTERRUPTS

When a local APIC receives an interrupt from a local source, an interrupt message from an I/O APIC, or an IPI, the manner in which it handles the message depends on processor implementation, as described in the following sections.
11.8.1  Interrupt Handling with the Pentium 4 and Intel Xeon Processors

With the Pentium 4 and Intel Xeon processors, the local APIC handles the local interrupts, interrupt messages, and IPIs it receives as follows:

1. It determines if it is the specified destination or not (see Figure 11-16). If it is the specified destination, it accepts the message; if it is not, it discards the message.

![Figure 11-16. Interrupt Acceptance Flow Chart for the Local APIC (Pentium 4 and Intel Xeon Processors)](image)

2. If the local APIC determines that it is the designated destination for the interrupt and if the interrupt request is an NMI, SMI, INIT, ExtINT, or SIPI, the interrupt is sent directly to the processor core for handling.

3. If the local APIC determines that it is the designated destination for the interrupt but the interrupt request is not one of the interrupts given in step 2, the local APIC sets the appropriate bit in the IRR.

4. When interrupts are pending in the IRR register, the local APIC dispatches them to the processor one at a time, based on their priority and the current processor priority in the PPR (see Section 11.8.3.1, “Task and Processor Priorities”).

5. When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the end-of-interrupt (EOI) register in the local APIC (see Section 11.8.5, “Signaling Interrupt Servicing Completion”). The act of writing to the EOI register causes the local APIC to delete the interrupt from its ISR queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed. (A write to the EOI register must not be included in the handler routine for an NMI, SMI, INIT, ExtINT, or SIPI.)

11.8.2  Interrupt Handling with the P6 Family and Pentium Processors

With the P6 family and Pentium processors, the local APIC handles the local interrupts, interrupt messages, and IPIs it receives as follows (see Figure 11-17).

1. (IPIs only) The local APIC examines the IPI message to determine if it is the specified destination for the IPI as described in Section 11.6.2, "Determining IPI Destination.” If it is the specified destination, it continues its acceptance procedure; if it is not the destination, it discards the IPI message. When the message specifies lowest-priority delivery mode, the local APIC will arbitrate with the other processors that were designated as recipients of the IPI message (see Section 11.6.2.4, "Lowest Priority Delivery Mode").

2. If the local APIC determines that it is the designated destination for the interrupt and if the interrupt request is an NMI, SMI, INIT, ExtINT, or INIT-deassert interrupt, or one of the MP protocol IPI messages (BIPI, FIPI, and SIPI), the interrupt is sent directly to the processor core for handling.

3. If the local APIC determines that it is the designated destination for the interrupt but the interrupt request is not one of the interrupts given in step 2, the local APIC looks for an open slot in one of its two pending interrupt queues contained in the IRR and ISR registers (see Figure 11-20). If a slot is available (see Section 11.8.4, "Interrupt Acceptance for Fixed Interrupts"), places the interrupt in the slot. If a slot is not available, it rejects the interrupt request and sends it back to the sender with a retry message.

4. When interrupts are pending in the IRR register, the local APIC dispatches them to the processor one at a time, based on their priority and the current processor priority in the PPR (see Section 11.8.3.1, “Task and Processor Priorities”).

5. When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the end-of-interrupt (EOI)
register in the local APIC (see Section 11.8.5, "Signaling Interrupt Servicing Completion"). The act of writing to the EOI register causes the local APIC to delete the interrupt from its queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed. (A write to the EOI register must not be included in the handler routine for an NMI, SMI, INIT, ExtINT, or SIPI.)

The following sections describe the acceptance of interrupts and their handling by the local APIC and processor in greater detail.

11.8.3 Interrupt, Task, and Processor Priority

Each interrupt delivered to the processor through the local APIC has a priority based on its vector number. The local APIC uses this priority to determine when to service the interrupt relative to the other activities of the processor, including the servicing of other interrupts.

Each interrupt vector is an 8-bit value. The **interrupt-priority class** is the value of bits 7:4 of the interrupt vector. The lowest interrupt-priority class is 1 and the highest is 15; interrupts with vectors in the range 0–15 (with interrupt-priority class 0) are illegal and are never delivered. Because vectors 0–31 are reserved for dedicated uses by the Intel 64 and IA-32 architectures, software should configure interrupt vectors to use interrupt-priority classes in the range 2–15.

Each interrupt-priority class encompasses 16 vectors. The relative priority of interrupts within an interrupt-priority class is determined by the value of bits 3:0 of the vector number. The higher the value of those bits, the higher the
priority within that interrupt-priority class. Thus, each interrupt vector comprises two parts, with the high 4 bits indicating its interrupt-priority class and the low 4 bits indicating its ranking within the interrupt-priority class.

11.8.3.1 Task and Processor Priorities

The local APIC also defines a task priority and a processor priority that determine the order in which interrupts are handled. The task-priority class is the value of bits 7:4 of the task-priority register (TPR), which can be written by software (TPR is a read/write register); see Figure 11-18.

NOTE

In this discussion, the term "task" refers to a software defined task, process, thread, program, or routine that is dispatched to run on the processor by the operating system. It does not refer to an IA-32 architecture defined task as described in Chapter 8, "Task Management."

The task priority allows software to set a priority threshold for interrupting the processor. This mechanism enables the operating system to temporarily block low priority interrupts from disturbing high-priority work that the processor is doing. The ability to block such interrupts using task priority results from the way that the TPR controls the value of the processor-priority register (PPR).6

The processor-priority class is a value in the range 0–15 that is maintained in bits 7:4 of the processor-priority register (PPR); see Figure 11-19. The PPR is a read-only register. The processor-priority class represents the current priority at which the processor is executing.

The value of the PPR is based on the value of TPR and the value ISRV; ISRV is the vector number of the highest priority bit that is set in the ISR or 00H if no bit is set in the ISR. (See Section 11.8.4 for more details on the ISR.) The value of PPR is determined as follows:

- PPR[7:4] (the processor-priority class) the maximum of TPR[7:4] (the task-priority class) and ISRV[7:4] (the priority of the highest priority interrupt in service).
- PPR[3:0] (the processor-priority sub-class) is determined as follows:
  - If TPR[7:4] > ISRV[7:4], PPR[3:0] is TPR[3:0] (the task-priority sub-class).
  - If TPR[7:4] < ISRV[7:4], PPR[3:0] is 0.
  - If TPR[7:4] = ISRV[7:4], PPR[3:0] may be either TPR[3:0] or 0. The actual behavior is model-specific.

6. The TPR also determines the arbitration priority of the local processor; see Section 11.6.2.4, "Lowest Priority Delivery Mode."
The processor-priority class determines the priority threshold for interrupting the processor. The processor will deliver only those interrupts that have an interrupt-priority class higher than the processor-priority class in the PPR. If the processor-priority class is 0, the PPR does not inhibit the delivery any interrupt; if it is 15, the processor inhibits the delivery of all interrupts. (The processor-priority mechanism does not affect the delivery of interrupts with the NMI, SMI, INIT, ExtINT, INIT-deassert, and start-up delivery modes.)

The processor does not use the processor-priority sub-class to determine which interrupts to deliver and which to inhibit. (The processor uses the processor-priority sub-class only to satisfy reads of the PPR.)

### 11.8.4 Interrupt Acceptance for Fixed Interrupts

The local APIC queues the fixed interrupts that it accepts in one of two interrupt pending registers: the interrupt request register (IRR) or in-service register (ISR). These two 256-bit read-only registers are shown in Figure 11-20. The 256 bits in these registers represent the 256 possible vectors; vectors 0 through 15 are reserved by the APIC (see also: Section 11.5.2, “Valid Interrupt Vectors”).

**NOTE**

All interrupts with an NMI, SMI, INIT, ExtINT, start-up, or INIT-deassert delivery mode bypass the IRR and ISR registers and are sent directly to the processor core for servicing.

<table>
<thead>
<tr>
<th>255</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>IRR</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>ISR</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>TMR</td>
</tr>
</tbody>
</table>

Addresses: IRR FEE0 0200H - FEE0 0270H  
ISR FEE0 0100H - FEE0 0170H  
TMR FEE0 0180H - FEE0 01F0H

Value after reset: 0H

Figure 11-20. IRR, ISR, and TMR Registers

The IRR contains the active interrupt requests that have been accepted, but not yet dispatched to the processor for servicing. When the local APIC accepts an interrupt, it sets the bit in the IRR that corresponds the vector of the accepted interrupt. When the processor core is ready to handle the next interrupt, the local APIC clears the highest priority IRR bit that is set and sets the corresponding ISR bit. The vector for the highest priority bit set in the ISR is then dispatched to the processor core for servicing.

While the processor is servicing the highest priority interrupt, the local APIC can send additional fixed interrupts by setting bits in the IRR. When the interrupt service routine issues a write to the EOI register (see Section 11.8.5, "Signaling Interrupt Servicing Completion"), the local APIC responds by clearing the highest priority ISR bit that is set. It then repeats the process of clearing the highest priority bit in the IRR and setting the corresponding bit in the ISR. The processor core then begins executing the service routing for the highest priority bit set in the ISR.

If more than one interrupt is generated with the same vector number, the local APIC can set the bit for the vector both in the IRR and the ISR. This means that for the Pentium 4 and Intel Xeon processors, the IRR and ISR can queue two interrupts for each interrupt vector: one in the IRR and one in the ISR. Any additional interrupts issued for the same interrupt vector are collapsed into the single bit in the IRR.

For the P6 family and Pentium processors, the IRR and ISR registers can queue no more than two interrupts per interrupt vector and will reject other interrupts that are received within the same vector.

If the local APIC receives an interrupt with an interrupt-priority class higher than that of the interrupt currently in service, and interrupts are enabled in the processor core, the local APIC dispatches the higher priority interrupt to the processor immediately (without waiting for a write to the EOI register). The currently executing interrupt handler is then interrupted so the higher-priority interrupt can be handled. When the handling of the higher-priority interrupt has been completed, the servicing of the interrupted interrupt is resumed.
The trigger mode register (TMR) indicates the trigger mode of the interrupt (see Figure 11-20). Upon acceptance of an interrupt into the IRR, the corresponding TMR bit is cleared for edge-triggered interrupts and set for level-triggered interrupts. If a TMR bit is set when an EOI cycle for its corresponding interrupt vector is generated, an EOI message is sent to all I/O APICs.

11.8.5 Signaling Interrupt Servicing Completion

For all interrupts except those delivered with the NMI, SMI, INIT, ExtINT, the start-up, or INIT-Deassert delivery mode, the interrupt handler must include a write to the end-of-interrupt (EOI) register (see Figure 11-21). This write must occur at the end of the handler routine, sometime before the IRET instruction. This action indicates that the servicing of the current interrupt is complete and the local APIC can issue the next interrupt from the ISR.

Upon receiving an EOI, the APIC clears the highest priority bit in the ISR and dispatches the next highest priority interrupt to the processor. If the terminated interrupt was a level-triggered interrupt, the local APIC also sends an end-of-interrupt message to all I/O APICs.

System software may prefer to direct EOIs to specific I/O APICs rather than having the local APIC send end-of-interrupt messages to all I/O APICs.

Software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register (see Section 11.9). If this bit is set, a broadcast EOI is not generated on an EOI cycle even if the associated TMR bit indicates that the current interrupt was level-triggered. The default value for the bit is 0, indicating that EOI broadcasts are performed.

Bit 12 of the Spurious Interrupt Vector Register is reserved to 0 if the processor does not support suppression of EOI broadcasts. Support for EOI-broadcast suppression is reported in bit 24 in the Local APIC Version Register (see Section 11.4.8); the feature is supported if that bit is set to 1. When supported, the feature is available in both xAPIC mode and x2APIC mode.

System software desiring to perform directed EOIs for level-triggered interrupts should set bit 12 of the Spurious Interrupt Vector Register and follow each the EOI to the local xAPIC for a level triggered interrupt with a directed EOI to the I/O APIC generating the interrupt (this is done by writing to the I/O APIC’s EOI register). System software performing directed EOIs must retain a mapping associating level-triggered interrupts with the I/O APICs in the system.

11.8.6 Task Priority in IA-32e Mode

In IA-32e mode, operating systems can manage the 16 interrupt-priority classes (see Section 11.8.3, “Interrupt, Task, and Processor Priority”) explicitly using the task priority register (TPR). Operating systems can use the TPR to temporarily block specific (low-priority) interrupts from interrupting a high-priority task. This is done by loading TPR with a value in which the task-priority class corresponds to the highest interrupt-priority class that is to be blocked. For example:

- Loading the TPR with a task-priority class of 8 (01000B) blocks all interrupts with an interrupt-priority class of 8 or less while allowing all interrupts with an interrupt-priority class of 9 or more to be recognized.
- Loading the TPR with a task-priority class of 0 enables all external interrupts.
- Loading the TPR with a task-priority class of 0FH (01111B) disables all external interrupts.

The TPR (shown in Figure 11-18) is cleared to 0 on reset. In 64-bit mode, software can read and write the TPR using an alternate interface, MOV CR8 instruction. The new task-priority class is established when the MOV CR8
instruction completes execution. Software does not need to force serialization after loading the TPR using MOV CR8.

Use of the MOV CRn instruction requires a privilege level of 0. Programs running at privilege level greater than 0 cannot read or write the TPR. An attempt to do so causes a general-protection exception. The TPR is abstracted from the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external device, such as an APIC or 8259. Typically, the IC provides a priority mechanism similar or identical to the TPR. The IC, however, is considered implementation-dependent with the underlying priority mechanisms subject to change. CR8, by contrast, is part of the Intel 64 architecture. Software can depend on this definition remaining unchanged.

Figure 11-22 shows the layout of CR8; only the low four bits are used. The remaining 60 bits are reserved and must be written with zeros. Failure to do this causes a general-protection exception.

![Figure 11-22. CR8 Register](image)

### 11.8.6.1 Interaction of Task Priorities between CR8 and APIC

The first implementation of Intel 64 architecture includes a local advanced programmable interrupt controller (APIC) that is similar to the APIC used with previous IA-32 processors. Some aspects of the local APIC affect the operation of the architecturally defined task priority register and the programming interface using CR8.

Notable CR8 and APIC interactions are:

- The processor powers up with the local APIC enabled.
- The APIC must be enabled for CR8 to function as the TPR. Writes to CR8 are reflected into the APIC Task Priority Register.
- APIC.TPR[bits 7:4] = CR8[bits 3:0], APIC.TPR[bits 3:0] = 0. A read of CR8 returns a 64-bit value which is the value of TPR[bits 7:4], zero extended to 64 bits.

There are no ordering mechanisms between direct updates of the APIC.TPR and CR8. Operating software should implement either direct APIC TPR updates or CR8 style TPR updates but not mix them. Software can use a serializing instruction (for example, CPUID) to serialize updates between MOV CR8 and stores to the APIC.

### 11.9 SPURIOUS INTERRUPT

A special situation may occur when a processor raises its task priority to be greater than or equal to the level of the interrupt for which the processor INTR signal is currently being asserted. If at the time the INTA cycle is issued, the interrupt that was to be dispensed has become masked (programmed by software), the local APIC will deliver a spurious-interrupt vector. Dispensing the spurious-interrupt vector does not affect the ISR, so the handler for this vector should return without an EOI.

The vector number for the spurious-interrupt vector is specified in the spurious-interrupt vector register (see Figure 11-23). The functions of the fields in this register are as follows:

**Spurious Vector**

Determines the vector number to be delivered to the processor when the local APIC generates a spurious vector.

(Pentium 4 and Intel Xeon processors.) Bits 0 through 7 of the this field are programmable by software.

(P6 family and Pentium processors). Bits 4 through 7 of this field are programmable by software, and bits 0 through 3 are hardwired to logical ones. Software writes to bits 0 through 3 have no effect.
APIC Software Enable/Disable

Allows software to temporarily enable (1) or disable (0) the local APIC (see Section 11.4.3, “Enabling or Disabling the Local APIC”).

Focus Processor Checking

Determines if focus processor checking is enabled (0) or disabled (1) when using the lowest-priority delivery mode. In Pentium 4 and Intel Xeon processors, this bit is reserved and should be cleared to 0.

Suppress EOI Broadcasts

Determines whether an EOI for a level-triggered interrupt causes EOI messages to be broadcast to the I/O APICs (0) or not (1). See Section 11.8.5. The default value for this bit is 0, indicating that EOI broadcasts are performed. This bit is reserved to 0 if the processor does not support EOI-broadcast suppression.

NOTE

Do not program an LVT or IOAPIC RTE with a spurious vector even if you set the mask bit. A spurious vector ISR does not do an EOI. If for some reason an interrupt is generated by an LVT or RTE entry, the bit in the in-service register will be left set for the spurious vector. This will mask all interrupts at the same or lower priority

Figure 11-23. Spurious-Interrupt Vector Register (SVR)

11.10 APIC BUS MESSAGE PASSING MECHANISM AND PROTOCOL (P6 FAMILY, PENTIUM PROCESSORS)

The Pentium 4 and Intel Xeon processors pass messages among the local and I/O APICs on the system bus, using the system bus message passing mechanism and protocol.

The P6 family and Pentium processors, pass messages among the local and I/O APICs on the serial APIC bus, as follows. Because only one message can be sent at a time on the APIC bus, the I/O APIC and local APICs employ a “rotating priority” arbitration protocol to gain permission to send a message on the APIC bus. One or more APICs may start sending their messages simultaneously. At the beginning of every message, each APIC presents the type of the message it is sending and its current arbitration priority on the APIC bus. This information is used for arbitration. After each arbitration cycle (within an arbitration round), only the potential winners keep driving the bus.
By the time all arbitration cycles are completed, there will be only one APIC left driving the bus. Once a winner is selected, it is granted exclusive use of the bus, and will continue driving the bus to send its actual message.

After each successfully transmitted message, all APICs increase their arbitration priority by 1. The previous winner (that is, the one that has just successfully transmitted its message) assumes a priority of 0 (lowest). An agent whose arbitration priority was 15 (highest) during arbitration, but did not send a message, adopts the previous winner’s arbitration priority, incremented by 1.

Note that the arbitration protocol described above is slightly different if one of the APICs issues a special End-Of-Interrupt (EOI). This high-priority message is granted the bus regardless of its sender’s arbitration priority, unless more than one APIC issues an EOI message simultaneously. In the latter case, the APICs sending the EOI messages arbitrate using their arbitration priorities.

If the APICs are set up to use “lowest priority” arbitration (see Section 11.6.2.4, “Lowest Priority Delivery Mode”) and multiple APICs are currently executing at the lowest priority (the value in the APR register), the arbitration priorities (unique values in the Arb ID register) are used to break ties. All 8 bits of the APR are used for the lowest priority arbitration.

11.10.1 Bus Message Formats
See Section 11.13, “APIC Bus Message Formats,” for a description of bus message formats used to transmit messages on the serial APIC bus.

11.11 MESSAGE SIGNALLED INTERRUPTS
The PCI Local Bus Specification, Rev 2.2 (www.pcisig.com) introduces the concept of message signalled interrupts. As the specification indicates:

“Message signalled interrupts (MSI) is an optional feature that enables PCI devices to request service by writing a system-specified message to a system-specified address (PCI DWORD memory write transaction). The transaction address specifies the message destination while the transaction data specifies the message. System software is expected to initialize the message destination and message during device configuration, allocating one or more non-shared messages to each MSI capable function.”

The capabilities mechanism provided by the PCI Local Bus Specification is used to identify and configure MSI capable PCI devices. Among other fields, this structure contains a Message Data Register and a Message Address Register. To request service, the PCI device function writes the contents of the Message Data Register to the address contained in the Message Address Register (and the Message Upper Address register for 64-bit message addresses).

Section 11.11.1 and Section 11.11.2 provide layout details for the Message Address Register and the Message Data Register. The operation issued by the device is a PCI write command to the Message Address Register with the Message Data Register contents. The operation follows semantic rules as defined for PCI write operations and is a DWORD operation.

11.11.1 Message Address Register Format
The format of the Message Address Register (lower 32-bits) is shown in Figure 11-24.
Fields in the Message Address Register are as follows:

1. **Bits 31-20** — These bits contain a fixed value for interrupt messages (0FEEH). This value locates interrupts at the 1-MByte area with a base address of 4G – 18M. All accesses to this region are directed as interrupt messages. Care must be taken to ensure that no other device claims the region as I/O space.

2. **Destination ID** — This field contains an 8-bit destination ID. It identifies the message’s target processor(s). The destination ID corresponds to bits 63:56 of the I/O APIC Redirection Table Entry if the IOAPIC is used to dispatch the interrupt to the processor(s).

3. **Redirection hint indication** (RH) — When this bit is set, the message is directed to the processor with the lowest interrupt priority among processors that can receive the interrupt.
   - When RH is 0, the interrupt is directed to the processor listed in the Destination ID field.
   - When RH is 1 and the physical destination mode is used, the Destination ID field must not be set to FFH; it must point to a processor that is present and enabled to receive the interrupt.
   - When RH is 1 and the logical destination mode is active in a system using a flat addressing model, the Destination ID field must be set so that bits set to 1 identify processors that are present and enabled to receive the interrupt.
   - If RH is set to 1 and the logical destination mode is active in a system using cluster addressing model, then Destination ID field must not be set to FFH; the processors identified with this field must be present and enabled to receive the interrupt.

4. **Destination mode (DM)** — This bit indicates whether the Destination ID field should be interpreted as logical or physical APIC ID for delivery of the lowest priority interrupt.
   - If RH is 1 and DM is 0, the Destination ID field is in physical destination mode and only the processor in the system that has the matching APIC ID is considered for delivery of that interrupt (this means no redirection).
   - If RH is 1 and DM is 1, the Destination ID Field is interpreted as in logical destination mode and the redirection is limited to only those processors that are part of the logical group of processors based on the processor’s logical APIC ID and the Destination ID field in the message. The logical group of processors consists of those identified by matching the 8-bit Destination ID with the logical destination identified by the Destination Format Register and the Logical Destination Register in each local APIC. The details are similar to those described in Section 11.6.2, “Determining IPI Destination.”
   - If RH is 0, then the DM bit is ignored and the message is sent ahead independent of whether the physical or logical destination mode is used.

### 11.11.2 Message Data Register Format

The layout of the Message Data Register is shown in Figure 11-25.

Reserved fields are not assumed to be any value. Software must preserve their contents on writes. Other fields in the Message Data Register are described below.

1. **Vector** — This 8-bit field contains the interrupt vector associated with the message. Values range from 010H to 0FEH. Software must guarantee that the field is not programmed with vector 00H to 0FH.

2. **Delivery Mode** — This 3-bit field specifies how the interrupt receipt is handled. Delivery Modes operate only in conjunction with specified Trigger Modes. Correct Trigger Modes must be guaranteed by software. Restrictions are indicated below:
   a. **000B (Fixed Mode)** — Deliver the signal to all the agents listed in the destination. The Trigger Mode for fixed delivery mode can be edge or level.
   b. **001B (Lowest Priority)** — Deliver the signal to the agent that is executing at the lowest priority of all agents listed in the destination field. The trigger mode can be edge or level.
   c. **010B (System Management Interrupt or SMI)** — The delivery mode is edge only. For systems that rely on SMI semantics, the vector field is ignored but must be programmed to all zeroes for future compatibility.
d. **100B (NMI)** — Deliver the signal to all the agents listed in the destination field. The vector information is ignored. NMI is an edge triggered interrupt regardless of the Trigger Mode Setting.

e. **101B (INIT)** — Deliver this signal to all the agents listed in the destination field. The vector information is ignored. INIT is an edge triggered interrupt regardless of the Trigger Mode Setting.

f. **111B (ExtINT)** — Deliver the signal to the INTR signal of all agents in the destination field (as an interrupt that originated from an 8259A compatible interrupt controller). The vector is supplied by the INTA cycle issued by the activation of the ExtINT. ExtINT is an edge triggered interrupt.

3. **Level** — Edge triggered interrupt messages are always interpreted as assert messages. For edge triggered interrupts this field is not used. For level triggered interrupts, this bit reflects the state of the interrupt input.

4. **Trigger Mode** — This field indicates the signal type that will trigger a message.
   a. **0** — Indicates edge sensitive.
   b. **1** — Indicates level sensitive.

### 11.12 EXTENDED XAPIC (X2APIC)

The x2APIC architecture extends the xAPIC architecture (described in Section 11.4) in a backward compatible manner and provides forward extendability for future Intel platform innovations. Specifically, the x2APIC architecture does the following.

- Retains all key elements of compatibility to the xAPIC architecture.
  - Delivery modes.
  - Interrupt and processor priorities.
  - Interrupt sources.
  - Interrupt destination types.
- Provides extensions to scale processor addressability for both the logical and physical destination modes.
ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

• Adds new features to enhance performance of interrupt delivery.
• Reduces complexity of logical destination mode interrupt delivery on link based platform architectures.
• Uses MSR programming interface to access APIC registers in x2APIC mode instead of memory-mapped interfaces. Memory-mapped interface is supported when operating in xAPIC mode.

11.12.1 Detecting and Enabling x2APIC Mode
Processor support for x2APIC mode can be detected by executing CPUID with EAX=1 and then checking ECX, bit 21 ECX. If CPUID.(EAX=1):ECX.21 is set, the processor supports the x2APIC capability and can be placed into the x2APIC mode.

System software can place the local APIC in the x2APIC mode by setting the x2APIC mode enable bit (bit 10) in the IA32_APIC_BASE MSR at MSR address 01BH. The layout for the IA32_APIC_BASE MSR is shown in Figure 11-26.

Table 11-5, “x2APIC operating mode configurations” describe the possible combinations of the enable bit (EN - bit 11) and the extended mode bit (EXTD - bit 10) in the IA32_APIC_BASE MSR.

Table 11-5. x2APIC Operating Mode Configurations

<table>
<thead>
<tr>
<th>xAPIC global enable (IA32_APIC_BASE[11])</th>
<th>x2APIC enable (IA32_APIC_BASE[10])</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>local APIC is disabled</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>invalid</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>local APIC is enabled in xAPIC mode</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>local APIC is enabled in x2APIC mode</td>
</tr>
</tbody>
</table>

Once the local APIC has been switched to x2APIC mode (EN = 1, EXTD = 1), switching back to xAPIC mode would require system software to disable the local APIC unit. Specifically, attempting to write a value to the IA32_APIC_BASE MSR that has (EN= 1, EXTD = 0) when the local APIC is enabled and in x2APIC mode causes a general-protection exception. Once bit 10 in IA32_APIC_BASE MSR is set, the only way to leave x2APIC mode using IA32_APIC_BASE would require a WRMSR to set both bit 11 and bit 10 to zero. Section 11.12.5, “x2APIC State Transitions,” provides a detailed state diagram for the state transitions allowed for the local APIC.

11.12.1.1 Instructions to Access APIC Registers
In x2APIC mode, system software uses RDMSR and WRMSR to access the APIC registers. The MSR addresses for accessing the x2APIC registers are architecturally defined and specified in Section 11.12.1.2, “x2APIC Register Address Space.” Executing the RDMSR instruction with the APIC register address specified in ECX returns the content of bits 0 through 31 of the APIC registers in EAX. Bits 32 through 63 are returned in register EDX - these bits are reserved if the APIC register being read is a 32-bit register. Similarly executing the WRMSR instruction with the APIC register address in ECX, writes bits 0 to 31 of register EAX to bits 0 to 31 of the specified APIC register. If the register is a 64-bit register then bits 0 to 31 of register EDX are written to bits 32 to 63 of the APIC register.
Interrupt Command Register is the only APIC register that is implemented as a 64-bit MSR. The semantics of handling reserved bits are defined in Section 11.12.1.3, “Reserved Bit Checking.”

11.12.1.2 x2APIC Register Address Space

The MSR address range 800H through 8FFH is architecturally reserved and dedicated for accessing APIC registers in x2APIC mode. Table 11-6 lists the APIC registers that are available in x2APIC mode. When appropriate, the table also gives the offset at which each register is available on the page referenced by IA32_APIC_BASE[35:12] in xAPIC mode.

There is a one-to-one mapping between the x2APIC MSRs and the legacy xAPIC register offsets with the following exceptions:

- The Destination Format Register (DFR): The DFR, supported at offset 0E0H in xAPIC mode, is not supported in x2APIC mode. There is no MSR with address 80EH.
- The Interrupt Command Register (ICR): The two 32-bit registers in xAPIC mode (at offsets 300H and 310H) are merged into a single 64-bit MSR in x2APIC mode (with MSR address 830H). There is no MSR with address 831H.
- The SELF IPI register. This register is available only in x2APIC mode at address 83FH. In xAPIC mode, there is no register defined at offset 3F0H.

MSR addresses in the range 800H–8FFH that are not listed in Table 11-6 (including 80EH and 831H) are reserved. Executions of RDMSR and WRMSR that attempt to access such addresses cause general-protection exceptions.

The MSR address space is compressed to allow for future growth. Every 32-bit register on a 128-bit boundary in the legacy MMIO space is mapped to a single MSR in the local x2APIC MSR address space. The upper 32-bits of all x2APIC MSRs (except for the ICR) are reserved.

<table>
<thead>
<tr>
<th>MSR Address (x2APIC mode)</th>
<th>MMIO Offset (xAPIC mode)</th>
<th>Register Name</th>
<th>MSR R/W Semantics</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>802H</td>
<td>020H</td>
<td>Local APIC ID register</td>
<td>Read-only</td>
<td>See Section 11.12.5.1 for initial values.</td>
</tr>
<tr>
<td>803H</td>
<td>030H</td>
<td>Local APIC Version register</td>
<td>Read-only</td>
<td>Same version used in xAPIC mode and x2APIC mode.</td>
</tr>
<tr>
<td>808H</td>
<td>080H</td>
<td>Task Priority Register (TPR)</td>
<td>Read/write</td>
<td>Bits 31:8 are reserved.</td>
</tr>
<tr>
<td>80AH</td>
<td>0A0H</td>
<td>Processor Priority Register (PPR)</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>80BH</td>
<td>0B0H</td>
<td>EOI register</td>
<td>Write-only</td>
<td>WRMSR of a non-zero value causes #GP(0).</td>
</tr>
<tr>
<td>80DH</td>
<td>0D0H</td>
<td>Logical Destination Register (LDR)</td>
<td>Read-only</td>
<td>Read/write in xAPIC mode.</td>
</tr>
<tr>
<td>80FH</td>
<td>0F0H</td>
<td>Spurious Interrupt Vector Register (SVR)</td>
<td>Read/write</td>
<td>See Section 11.9 for reserved bits.</td>
</tr>
<tr>
<td>810H</td>
<td>100H</td>
<td>In-Service Register (ISR); bits 31:0</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>811H</td>
<td>110H</td>
<td>ISR bits 63:32</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>812H</td>
<td>120H</td>
<td>ISR bits 95:64</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>813H</td>
<td>130H</td>
<td>ISR bits 127:96</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>814H</td>
<td>140H</td>
<td>ISR bits 159:128</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>815H</td>
<td>150H</td>
<td>ISR bits 191:160</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>816H</td>
<td>160H</td>
<td>ISR bits 223:192</td>
<td>Read-only</td>
<td></td>
</tr>
</tbody>
</table>
### Table 11-6. Local APIC Register Address Map Supported by x2APIC (Contd.)

<table>
<thead>
<tr>
<th>MSR Address (x2APIC mode)</th>
<th>MMIO Offset (xAPIC mode)</th>
<th>Register Name</th>
<th>MSR R/W Semantics</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>817H</td>
<td>170H</td>
<td>ISR bits 255:224</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>818H</td>
<td>180H</td>
<td>Trigger Mode Register (TMR); bits 31:0</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>819H</td>
<td>190H</td>
<td>TMR bits 63:32</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>81AH</td>
<td>1A0H</td>
<td>TMR bits 95:64</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>81BH</td>
<td>1B0H</td>
<td>TMR bits 127:96</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>81CH</td>
<td>1C0H</td>
<td>TMR bits 159:128</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>81DH</td>
<td>1D0H</td>
<td>TMR bits 191:160</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>81EH</td>
<td>1E0H</td>
<td>TMR bits 223:192</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>81FH</td>
<td>1F0H</td>
<td>TMR bits 255:224</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>820H</td>
<td>200H</td>
<td>Interrupt Request Register (IRR); bits 31:0</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>821H</td>
<td>210H</td>
<td>IRR bits 63:32</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>822H</td>
<td>220H</td>
<td>IRR bits 95:64</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>823H</td>
<td>230H</td>
<td>IRR bits 127:96</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>824H</td>
<td>240H</td>
<td>IRR bits 159:128</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>825H</td>
<td>250H</td>
<td>IRR bits 191:160</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>826H</td>
<td>260H</td>
<td>IRR bits 223:192</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>827H</td>
<td>270H</td>
<td>IRR bits 255:224</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>828H</td>
<td>280H</td>
<td>Error Status Register (ESR)</td>
<td>Read/write</td>
<td>WRMSR of a non-zero value causes #GP(0). See Section 11.5.3.</td>
</tr>
<tr>
<td>82FH</td>
<td>2F0H</td>
<td>LVT CMCI register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>830H^4</td>
<td>300H and 310H</td>
<td>Interrupt Command Register (ICR)</td>
<td>Read/write</td>
<td>See Figure 11-28 for reserved bits</td>
</tr>
<tr>
<td>832H</td>
<td>320H</td>
<td>LVT Timer register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>833H</td>
<td>330H</td>
<td>LVT Thermal Sensor register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>834H</td>
<td>340H</td>
<td>LVT Performance Monitoring register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>835H</td>
<td>350H</td>
<td>LVT LINT0 register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>836H</td>
<td>360H</td>
<td>LVT LINT1 register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>837H</td>
<td>370H</td>
<td>LVT Error register</td>
<td>Read/write</td>
<td>See Figure 11-8 for reserved bits.</td>
</tr>
<tr>
<td>838H</td>
<td>380H</td>
<td>Initial Count register (for Timer)</td>
<td>Read/write</td>
<td></td>
</tr>
<tr>
<td>839H</td>
<td>390H</td>
<td>Current Count register (for Timer)</td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>83EH</td>
<td>3E0H</td>
<td>Divide Configuration Register (DCR; for Timer)</td>
<td>Read/write</td>
<td>See Figure 11-10 for reserved bits.</td>
</tr>
<tr>
<td>83FH</td>
<td>Not available</td>
<td>SELF IPI^5</td>
<td>Write-only</td>
<td>Available only in x2APIC mode.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. WRMSR causes #GP(0) for read-only registers.
11.12.1.3 Reserved Bit Checking

Section 11.12.1.2 and Table 11-6 specifies the reserved bit definitions for the APIC registers in x2APIC mode. Non-zero writes (by WRMSR instruction) to reserved bits to these registers will raise a general protection fault exception while reads return zeros (RsvdZ semantics).

In x2APIC mode, the local APIC ID register is increased to 32 bits wide. This enables $2^{32}-1$ processors to be addressable in physical destination mode. This 32-bit value is referred to as “x2APIC ID”. A processor implementation may choose to support less than 32 bits in its hardware. System software should be agnostic to the actual number of bits that are implemented. All non-implemented bits will return zeros on reads by software.

The APIC ID value of $\text{FFFF_FFFF}_{16}$ and the highest value corresponding to the implemented bit-width of the local APIC ID register in the system are reserved and cannot be assigned to any logical processor.

In x2APIC mode, the local APIC ID register is a read-only register to system software and will be initialized by hardware. It is accessed via the RDMSR instruction reading the MSR at address $\text{0802H}_{16}$.

Each logical processor in the system (including clusters with a communication fabric) must be configured with an unique x2APIC ID to avoid collisions of x2APIC IDs. On DP and high-end MP processors targeted to specific market segments and depending on the system configuration, it is possible that logical processors in different and “un-connected” clusters power up initialized with overlapping x2APIC IDs. In these configurations, a model-specific means may be provided in those product segments to enable BIOS and/or platform firmware to re-configure the x2APIC IDs in some clusters to provide for unique and non-overlapping system wide IDs before configuring the disconnected components into a single system.

11.12.2 x2APIC Register Availability

The local APIC registers can be accessed via the MSR interface only when the local APIC has been switched to the x2APIC mode as described in Section 11.12.1. Accessing any APIC register in the MSR address range $\text{0800H}_{16}$ through $\text{08FFH}_{16}$ via RDMSR or WRMSR when the local APIC is not in x2APIC mode causes a general-protection exception. In x2APIC mode, the memory mapped interface is not available and any access to the MMIO interface will behave similar to that of a legacy xAPIC in globally disabled state. Table 11-7 provides the interactions between the legacy & extended modes and the legacy and register interfaces.

<table>
<thead>
<tr>
<th>MMIO Interface</th>
<th>MSR Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>xAPIC mode</td>
<td>Available</td>
</tr>
<tr>
<td>x2APIC mode</td>
<td>Behavior identical to xAPIC in globally disabled state</td>
</tr>
</tbody>
</table>

11.12.3 MSR Access in x2APIC Mode

To allow for efficient access to the APIC registers in x2APIC mode, the serializing semantics of WRMSR are relaxed when writing to the APIC registers. Thus, system software should not use “WRMSR to APIC registers in x2APIC mode” as a serializing instruction. Read and write accesses to the APIC registers will occur in program order. A WRMSR to an APIC register may complete before all preceding stores are globally visible; software can prevent this by inserting a serializing instruction or the sequence MFENCE;LFENCE before the WRMSR.

The RDMSR instruction is not serializing and this behavior is unchanged when reading APIC registers in x2APIC mode. System software accessing the APIC registers using the RDMSR instruction should not expect a serializing behavior. (Note: The MMIO-based xAPIC interface is mapped by system software as an un-cached region. Consequently, read/writes to the xAPIC-MMIO interface have serializing semantics in the xAPIC mode.)
11.12.4 VM-Exit Controls for MSRs and x2APIC Registers

The VMX architecture allows a VMM to specify lists of MSRs to be loaded or stored on VMX transitions using the VMX-transition MSR areas (see VM-exit MSR-store address field, VM-exit MSR-load address field, and VM-entry MSR-load address field in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C).

The x2APIC MSRs cannot be loaded and stored on VMX transitions. A VMX transition fails if the VMM has specified that the transition should access any MSRs in the address range from 0000_0800H to 0000_08FFH (the range used for accessing the x2APIC registers). Specifically, processing of an 128-bit entry in any of the VMX-transition MSR areas fails if bits 31:0 of that entry (represented as ENTRY_LOW_DW) satisfies the expression: "ENTRY_LOW_DW & FFFFF800H = 00000800H". Such a failure causes an associated VM entry to fail (by reloading host state) and causes an associated VM exit to lead to VMX abort.

11.12.5 x2APIC State Transitions

This section provides a detailed description of the x2APIC states of a local x2APIC unit, transitions between these states as well as interactions of these states with INIT and reset.

11.12.5.1 x2APIC States

The valid states for a local x2APIC unit are listed in Table 11-5.

- APIC disabled: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=0.
- xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0.

The state corresponding to EXTD=1 and EN=0 is not valid and it is not possible to get into this state. An execution of WRMSR to the IA32_APIC_BASE_MSR that attempts a transition from a valid state to this invalid state causes a general-protection exception. Figure 11-27 shows the comprehensive state transition diagram for a local x2APIC unit.

On coming out of reset, the local APIC unit is enabled and is in the xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0. The APIC registers are initialized as follows.

- The local APIC ID is initialized by hardware with a 32 bit ID (x2APIC ID). The lowest 8 bits of the x2APIC ID are the legacy local xAPIC ID, and are stored in the upper 8 bits of the APIC register for access in xAPIC mode.
- The following APIC registers are reset to all zeros for those fields that are defined in the xAPIC mode.
  - IRR, ISR, TMR, ICR, LDR, TPR, Divide Configuration Register (See Section 11.4 through Section 11.6 for details of individual APIC registers).
  - Timer initial count and timer current count registers.
- The LVT registers are reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The Spurious Interrupt Vector Register is initialized to 000000FFH.
- The DFR (available only in xAPIC mode) is reset to all 1s.
- SELF IPI register is reset to zero.
x2APIC After Reset

The valid transitions from the xAPIC mode state are:

- to the x2APIC mode by setting EXT to 1 (resulting EN=1, EXTD=1). The physical x2APIC ID (see Figure 11-6) is preserved across this transition and the logical x2APIC ID (see Figure 11-29) is initialized by hardware during this transition as documented in Section 11.12.10.2. The state of the extended fields in other APIC registers, which was not initialized at reset, is not architecturally defined across this transition and system software should explicitly initialize those programmable APIC registers.

- to the disabled state by setting EN to 0 (resulting EN=0, EXTD=0).

The result of an INIT in the xAPIC state places the APIC in the state with EN=1, EXTD=0. The state of the local APIC ID register is preserved (the 8-bit xAPIC ID is in the upper 8 bits of the APIC ID register). All the other APIC registers are initialized as a result of INIT.

A reset in this state places the APIC in the state with EN=1, EXTD=0. The state of the local APIC ID register is initialized as described in Section 11.12.5.1. All the other APIC registers are initialized described in Section 11.12.5.1.

x2APIC Transitions From x2APIC Mode

From the x2APIC mode, the only valid x2APIC transition using IA32_APIC_BASE is to the state where the x2APIC is disabled by setting EN to 0 and EXTD to 0. The x2APIC ID (32 bits) and the legacy local xAPIC ID (8 bits) are preserved across this transition. A transition from the x2APIC mode to xAPIC mode is not valid, and the corresponding WRMSR to the IA32_APIC_BASE MSR causes a general-protection exception.

A reset in this state places the x2APIC in xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 11.12.5.1.

An INIT in this state keeps the x2APIC in the x2APIC mode. The state of the local APIC ID register is preserved (all 32 bits). However, all the other APIC registers are initialized as a result of the INIT transition.
x2APIC Transitions From Disabled Mode

From the disabled state, the only valid x2APIC transition using IA32_APIC_BASE is to the xAPIC mode (EN= 1, EXTD = 0). Thus the only means to transition from x2APIC mode to xAPIC mode is a two-step process:

• first transition from x2APIC mode to local APIC disabled mode (EN= 0, EXTD = 0),
• followed by another transition from disabled mode to xAPIC mode (EN= 1, EXTD= 0).

Consequently, all the APIC register states in the x2APIC, except for the x2APIC ID (32 bits), are not preserved across mode transitions.

A reset in the disabled state places the x2APIC in the xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 11.12.5.1.

An INIT in the disabled state keeps the x2APIC in the disabled state.

State Changes From xAPIC Mode to x2APIC Mode

After APIC register states have been initialized by software in xAPIC mode, a transition from xAPIC mode to x2APIC mode does not affect most of the APIC register states, except the following:

• The Logical Destination Register is not preserved.
• Any APIC ID value written to the memory-mapped local APIC ID register is not preserved.
• The high half of the Interrupt Command Register is not preserved.

11.12.6 Routing of Device Interrupts in x2APIC Mode

The x2APIC architecture is intended to work with all existing IOxAPIC units as well as all PCI and PCI Express (PCIe) devices that support the capability for message-signaled interrupts (MSI). Support for x2APIC modifies only the following:

• the local APIC units;
• the interconnects joining IOxAPIC units to the local APIC units; and
• the interconnects joining MSI-capable PCI and PCIe devices to the local APIC units.

No modifications are required to MSI-capable PCI and PCIe devices. Similarly, no modifications are required to IOxAPIC units. This is made possible through use of the interrupt-remapping architecture specified in the Intel® Virtualization Technology for Directed I/O Specification, Revision 1.3 and/or later versions, for the routing of interrupts from MSI-capable devices to local APIC units operating in x2APIC mode.

11.12.7 Initialization by System Software

Routing of device interrupts to local APIC units operating in x2APIC mode requires use of the interrupt-remapping architecture specified in the Intel® Virtualization Technology for Directed I/O Specification (Revision 1.3 and/or later versions). Because of this, BIOS must enumerate support for and software must enable this interrupt remapping with Extended Interrupt Mode Enabled before it enabling x2APIC mode in the local APIC units.

The ACPI interfaces for the x2APIC are described in Section 5.2, “ACPI System Description Tables,” of the Advanced Configuration and Power Interface Specification, Revision 4.0a (http://www.acpi.info/spec.htm). The default behavior for BIOS is to pass the control to the operating system with the local x2APICs in xAPIC mode if all APIC IDs reported by CPUID.0BH:EDX are less than 255, and in x2APIC mode if there are any logical processor reporting an APIC ID of 255 or greater.

11.12.8 CPUID Extensions And Topology Enumeration

For Intel 64 and IA-32 processors that support x2APIC, a value of 1 reported by CPUID.01H:ECX[21] indicates that the processor supports x2APIC and the extended topology enumeration leaf (CPUID.0BH).
The extended topology enumeration leaf can be accessed by executing CPUID with EAX = 0BH. Processors that do not support x2APIC may support CPUID leaf 0BH. Software can detect the availability of the extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal or 11 (0BH), then proceed to next step
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. If available, the extended topology enumeration leaf is the preferred mechanism for enumerating topology. The presence of CPUID leaf 0BH in a processor does not guarantee support for x2APIC. If CPUID.EAX=0BH, ECX=0H:EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

The extended topology enumeration leaf is intended to assist software with enumerating processor topology on systems that requires 32-bit x2APIC IDs to address individual logical processors. Details of CPUID leaf 0BH can be found in the reference pages of CPUID in Chapter 3 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

Processor topology enumeration algorithm for processors supporting the extended topology enumeration leaf of CPUID and processors that do not support CPUID leaf 0BH are treated in Section 9.9.4, “Algorithm for Three-Domain Mappings of APIC_ID.”

### 11.12.8.1 Consistency of APIC IDs and CPUID

The consistency of physical x2APIC ID in MSR 802H in x2APIC mode and the 32-bit value returned in CPUID.0BH:EDX is facilitated by processor hardware.

CPUID.0BH:EDX will report the full 32 bit ID, in xAPIC and x2APIC mode. This allows BIOS to determine if a system has processors with IDs exceeding the 8-bit initial APIC ID limit (CPUID.01H:EBX[31:24]). Initial APIC ID (CPUID.01H:EBX[31:24]) is always equal to CPUID.0BH:EDX[7:0].

If the values of CPUID.0BH:EDX reported by all logical processors in a system are less than 255, BIOS can transfer control to OS in xAPIC mode.

If the values of CPUID.0BH:EDX reported by some logical processors in a system are greater than or equal to 255, BIOS must support two options to hand off to OS.

- If BIOS enables logical processors with x2APIC IDs greater than 255, then it should enable x2APIC in the Boot Strap Processor (BSP) and all Application Processors (AP) before passing control to the OS. Applications requiring processor topology information must use OS provided services based on x2APIC IDs or CPUID.0BH leaf.
- If a BIOS transfers control to OS in xAPIC mode, then the BIOS must ensure that only logical processors with CPUID.0BH:EDX value less than 255 are enabled. BIOS initialization on all logical processors with CPUID.0B:EDX values greater than or equal to 255 must (a) disable APIC and execute CLI in each logical processor, and (b) leave these logical processor in the lowest power state so that these processors do not respond to INIT IPI during OS boot. The BSP and all the enabled logical processor operate in xAPIC mode after BIOS passed control to OS. Application requiring processor topology information can use OS provided legacy services based on 8-bit initial APIC IDs or legacy topology information from CPUID.01H and CPUID 04H leaves. Even if the BIOS passes control in xAPIC mode, an OS can switch the processors to x2APIC mode later. BIOS SMM handler should always use the APIC_BASE_MSR, determine the APIC mode and use the corresponding access method.

### 11.12.9 ICR Operation in x2APIC Mode

In x2APIC mode, the layout of the Interrupt Command Register is shown in Figure 11-28. The lower 32 bits of ICR in x2APIC mode is identical to the lower half of the ICR in xAPIC mode, except the Delivery Status bit is removed since it is not needed in x2APIC mode. The destination ID field is expanded to 32 bits in x2APIC mode.

To send an IPI using the ICR, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. Self IPIs can also be sent using the SELF IPI register (see Section 11.12.11).
A single MSR write to the Interrupt Command Register is required for dispatching an interrupt in x2APIC mode. With the removal of the Delivery Status bit, system software no longer has a reason to read the ICR. It remains readable only to aid in debugging; however, software should not assume the value returned by reading the ICR is the last written value.

A destination ID value of FFFF_FFFFH is used for broadcast of interrupts in both logical destination and physical destination modes.

11.12.10 Determining IPI Destination in x2APIC Mode

11.12.10.1 Logical Destination Mode in x2APIC Mode

In x2APIC mode, the Logical Destination Register (LDR) is increased to 32 bits wide. It is a read-only register to system software. This 32-bit value is referred to as "logical x2APIC ID". System software accesses this register via the RDMSR instruction reading the MSR at address 80DH. Figure 11-29 provides the layout of the Logical Destination Register in x2APIC mode.
In the xAPIC mode, the Destination Format Register (DFR) through the MMIO interface determines the choice of a flat logical mode or a clustered logical mode. Flat logical mode is not supported in the x2APIC mode. Hence the Destination Format Register (DFR) is eliminated in x2APIC mode.

The 32-bit logical x2APIC ID field of LDR is partitioned into two sub-fields:

- **Cluster ID (LDR[31:16]):** is the address of the destination cluster
- **Logical ID (LDR[15:0]):** defines a logical ID of the individual local x2APIC within the cluster specified by LDR[31:16].

This layout enables $2^{16}$-1 clusters each with up to 16 unique logical IDs - effectively providing an addressability of $(2^{20} - 16)$ processors in logical destination mode.

It is likely that processor implementations may choose to support less than 16 bits of the cluster ID or less than 16-bits of the Logical ID in the Logical Destination Register. However, system software should be agnostic to the number of bits implemented in the cluster ID and logical ID sub-fields. The x2APIC hardware initialization will ensure that the appropriately initialized logical x2APIC IDs are available to system software and reads of non-implemented bits return zero. This is a read-only register that software must read to determine the logical x2APIC ID of the processor. Specifically, software can apply a 16-bit mask to the lowest 16 bits of the logical x2APIC ID to identify the logical address of a processor within a cluster without needing to know the number of implemented bits in cluster ID and Logical ID sub-fields. Similarly, software can create a message destination address for cluster model, by bit-Oring the Logical X2APIC ID (31:0) of processors that have matching Cluster ID(31:16).

To enable cluster ID assignment in a fashion that matches the system topology characteristics and to enable efficient routing of logical mode lowest priority device interrupts in link based platform interconnects, the LDR are initialized by hardware based on the value of x2APIC ID upon x2APIC state transitions. Details of this initialization are provided in Section 11.12.10.2.

### 11.12.10.2 Deriving Logical x2APIC ID from the Local x2APIC ID

In x2APIC mode, the 32-bit logical x2APIC ID, which can be read from LDR, is derived from the 32-bit local x2APIC ID. Specifically, the 16-bit logical ID sub-field is derived by shifting 1 by the lowest 4 bits of the x2APIC ID, i.e., Logical ID = $1 \ll x2APIC ID[3:0]$. The remaining bits of the x2APIC ID then form the cluster ID portion of the logical x2APIC ID:

$$Logical\ x2APIC\ ID = ([x2APIC\ ID[19:4] \ll 16] | (1 \ll x2APIC\ ID[3:0]))$$

The use of the lowest 4 bits in the x2APIC ID implies that at least 16 APIC IDs are reserved for logical processors within a socket in multi-socket configurations. If more than 16 APIC IDS are reserved for logical processors in a socket/package then multiple cluster IDs can exist within the package.

The LDR initialization occurs whenever the x2APIC mode is enabled (see Section 11.12.5).
11.12.11 SELF IPI Register

SELF IPIs are used extensively by some system software. The x2APIC architecture introduces a new register interface. This new register is dedicated to the purpose of sending self-IPIs with the intent of enabling a highly optimized path for sending self-IPIs.

Figure 11-30 provides the layout of the SELF IPI register. System software only specifies the vector associated with the interrupt to be sent. The semantics of sending a self-IPI via the SELF IPI register are identical to sending a self-targeted edge triggered fixed interrupt with the specified vector. Specifically the semantics are identical to the following settings for an inter-processor interrupt sent via the ICR - Destination Shorthand (ICR[19:18] = 01 (Self)), Trigger Mode (ICR[15] = 0 (Edge)), Delivery Mode (ICR[10:8] = 000 (Fixed)), Vector (ICR[7:0] = Vector).

The SELF IPI register is a write-only register. A RDMSR instruction with address of the SELF IPI register causes a general-protection exception.

The handling and prioritization of a self-IPI sent via the SELF IPI register is architecturally identical to that for an IPI sent via the ICR from a legacy xAPIC unit. Specifically the state of the interrupt would be tracked via the Interrupt Request Register (IRR) and In Service Register (ISR) and Trigger Mode Register (TMR) as if it were received from the system bus. Also sending the IPI via the Self Interrupt Register ensures that interrupt is delivered to the processor core. Specifically completion of the WRMSR instruction to the SELF IPI register implies that the interrupt has been logged into the IRR. As expected for edge triggered interrupts, depending on the processor priority and readiness to accept interrupts, it is possible that interrupts sent via the SELF IPI register or via the ICR with identical vectors can be combined.

11.13 APIC BUS MESSAGE FORMATS

This section describes the message formats used when transmitting messages on the serial APIC bus. The information described here pertains only to the Pentium and P6 family processors.

11.13.1 Bus Message Formats

The local and I/O APICs transmit three types of messages on the serial APIC bus: EOI message, short message, and non-focused lowest priority message. The purpose of each type of message and its format are described below.

11.13.2 EOI Message

Local APICs send 14-cycle EOI messages to the I/O APIC to indicate that a level triggered interrupt has been accepted by the processor. This interrupt, in turn, is a result of software writing into the EOI register of the local APIC. Table 11-8 shows the cycles in an EOI message.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Bit1</th>
<th>Bit0</th>
<th>11 = EOI</th>
<th>Arbitration ID bits 3 through 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ArbID3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 11-30. SELF IPI register
The checksum is computed for cycles 6 through 9. It is a cumulative sum of the 2-bit (Bit1:Bit0) logical data values. The carry out of all but the last addition is added to the sum. If any APIC computes a different checksum than the one appearing on the bus in cycle 10, it signals an error, driving 11 on the APIC bus during cycle 12. In this case, the APICs disregard the message. The sending APIC will receive an appropriate error indication (see Section 11.5.3, "Error Handling") and resend the message. The status cycles are defined in Table 11-11.

### 11.13.2.1 Short Message

Short messages (21-cycles) are used for sending fixed, NMI, SMI, INIT, start-up, ExtINT, and lowest-priority-with-focus interrupts. Table 11-9 shows the cycles in a short message.

<table>
<thead>
<tr>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

**Table 11-9. Short Message (21 Cycles)**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Bit1</th>
<th>Bit0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ArbitID3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>ArbitID2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ArbitID1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>ArbitID0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>DM</td>
<td>M2</td>
</tr>
<tr>
<td>7</td>
<td>M1</td>
<td>M0</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>TM</td>
</tr>
<tr>
<td>9</td>
<td>V7</td>
<td>V6</td>
</tr>
<tr>
<td>10</td>
<td>V5</td>
<td>V4</td>
</tr>
<tr>
<td>11</td>
<td>V3</td>
<td>V2</td>
</tr>
<tr>
<td>12</td>
<td>V1</td>
<td>V0</td>
</tr>
<tr>
<td>13</td>
<td>D7</td>
<td>D6</td>
</tr>
<tr>
<td>14</td>
<td>D5</td>
<td>D4</td>
</tr>
<tr>
<td>15</td>
<td>D3</td>
<td>D2</td>
</tr>
<tr>
<td>16</td>
<td>D1</td>
<td>D0</td>
</tr>
</tbody>
</table>
If the physical delivery mode is being used, then cycles 15 and 16 represent the APIC ID and cycles 13 and 14 are considered don't care by the receiver. If the logical delivery mode is being used, then cycles 13 through 16 are the 8-bit logical destination field.

For shorthands of “all-incl-self” and “all-excl-self,” the physical delivery mode and an arbitration priority of 15 (D0:D3 = 1111) are used. The agent sending the message is the only one required to distinguish between the two cases. It does so using internal information.

When using lowest priority delivery with an existing focus processor, the focus processor identifies itself by driving 10 during cycle 19 and accepts the interrupt. This is an indication to other APICs to terminate arbitration. If the focus processor has not been found, the short message is extended on-the-fly to the non-focused lowest-priority message. Note that except for the EOI message, messages generating a checksum or an acceptance error (see Section 11.5.3, “Error Handling”) terminate after cycle 21.

11.13.2.2 Non-focused Lowest Priority Message

These 34-cycle messages (see Table 11-10) are used in the lowest priority delivery mode when a focus processor is not present. Cycles 1 through 20 are same as for the short message. If during the status cycle (cycle 19) the state of the (A:A) flags is 10B, a focus processor has been identified, and the short message format is used (see Table 11-9). If the (A:A) flags are set to 00B, lowest priority arbitration is started and the 34-cycles of the non-focused lowest priority message are competed. For other combinations of status flags, refer to Section 11.13.2.3, “APIC Bus Status Cycles.”

Table 11-10. Non-Focused Lowest Priority Message (34 Cycles)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Bit0</th>
<th>Bit1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ArbID3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>ArbID2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ArbID1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>ArbID0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>DM</td>
<td>M2</td>
</tr>
<tr>
<td>7</td>
<td>M1</td>
<td>M0</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>TM</td>
</tr>
<tr>
<td>9</td>
<td>V7</td>
<td>V6</td>
</tr>
<tr>
<td>10</td>
<td>V5</td>
<td>V4</td>
</tr>
<tr>
<td>11</td>
<td>V3</td>
<td>V2</td>
</tr>
<tr>
<td>12</td>
<td>V1</td>
<td>V0</td>
</tr>
<tr>
<td>13</td>
<td>D7</td>
<td>D6</td>
</tr>
<tr>
<td>14</td>
<td>D5</td>
<td>D4</td>
</tr>
<tr>
<td>15</td>
<td>D3</td>
<td>D2</td>
</tr>
<tr>
<td>16</td>
<td>D1</td>
<td>D0</td>
</tr>
</tbody>
</table>
Cycles 21 through 28 are used to arbitrate for the lowest priority processor. The processors participating in the arbitration drive their inverted processor priority on the bus. Only the local APICs having free interrupt slots participate in the lowest priority arbitration. If no such APIC exists, the message will be rejected, requiring it to be tried at a later time.

Cycles 29 through 32 are also used for arbitration in case two or more processors have the same lowest priority. In the lowest priority delivery mode, all combinations of errors in cycle 33 (A2 A2) will set the “accept error” bit in the error status register (see Figure 11-9). Arbitration priority update is performed in cycle 20, and is not affected by errors detected in cycle 33. Only the local APIC that wins in the lowest priority arbitration, drives cycle 33. An error in cycle 33 will force the sender to resend the message.

11.13.2.3 APIC Bus Status Cycles

Certain cycles within an APIC bus message are status cycles. During these cycles the status flags (A:A) and (A1:A1) are examined. Table 11-11 shows how these status flags are interpreted, depending on the current delivery mode and existence of a focus processor.
<table>
<thead>
<tr>
<th>Delivery Mode</th>
<th>A Status</th>
<th>A1 Status</th>
<th>A2 Status</th>
<th>Update ArbID and Cycle#</th>
<th>Message Length</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOI</td>
<td>00: CS_OK</td>
<td>10: Accept</td>
<td>XX:</td>
<td>Yes, 13</td>
<td>14 Cycle</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK</td>
<td>11: Retry</td>
<td>XX:</td>
<td>Yes, 13</td>
<td>14 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK</td>
<td>0X: Accept Error</td>
<td>XX:</td>
<td>No</td>
<td>14 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>11: CS_Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>14 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>10: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>14 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>01: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>14 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td>Fixed</td>
<td>00: CS_OK</td>
<td>10: Accept</td>
<td>XX:</td>
<td>Yes, 20</td>
<td>21 Cycle</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK</td>
<td>11: Retry</td>
<td>XX:</td>
<td>Yes, 20</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK</td>
<td>0X: Accept Error</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>11: CS_Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>10: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>01: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td>NMI, SMI, INIT, ExtINT, Start-Up</td>
<td>00: CS_OK</td>
<td>10: Accept</td>
<td>XX:</td>
<td>Yes, 20</td>
<td>21 Cycle</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK</td>
<td>11: Retry</td>
<td>XX:</td>
<td>Yes, 20</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK</td>
<td>0X: Accept Error</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>11: CS_Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>10: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>01: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td>Lowest</td>
<td>00: CS_OK, NoFocus</td>
<td>11: Do Lowest</td>
<td>10: Accept</td>
<td>Yes, 20</td>
<td>34 Cycle</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK, NoFocus</td>
<td>11: Do Lowest</td>
<td>11: Error</td>
<td>Yes, 20</td>
<td>34 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK, NoFocus</td>
<td>11: Do Lowest</td>
<td>0X: Error</td>
<td>Yes, 20</td>
<td>34 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK, NoFocus</td>
<td>10: End and Retry</td>
<td>XX:</td>
<td>Yes, 20</td>
<td>34 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>00: CS_OK, NoFocus</td>
<td>0X: Error</td>
<td>XX:</td>
<td>No</td>
<td>34 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>10: CS_OK, Focus</td>
<td>XX:</td>
<td>XX:</td>
<td>Yes, 20</td>
<td>34 Cycle</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>11: CS_Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>01: Error</td>
<td>XX:</td>
<td>XX:</td>
<td>No</td>
<td>21 Cycle</td>
<td>Yes</td>
</tr>
</tbody>
</table>
This chapter describes the memory cache and cache control mechanisms, the TLBs, and the store buffer in Intel 64 and IA-32 processors. It also describes the memory type range registers (MTRRs) introduced in the P6 family processors and how they are used to control caching of physical memory locations.

12.1 INTERNAL CACHES, TLBS, AND BUFFERS

The Intel 64 and IA-32 architectures support cache, translation look aside buffers (TLBs), and a store buffer for temporary on-chip (and external) storage of instructions and data. (Figure 12-1 shows the arrangement of caches, TLBs, and the store buffer for the Pentium 4 and Intel Xeon processors.) Table 12-1 shows the characteristics of these caches and buffers for the Pentium 4, Intel Xeon, P6 family, and Pentium processors. The sizes and characteristics of these units are machine specific and may change in future versions of the processor. The CPUID instruction returns the sizes and characteristics of the caches and buffers for the processor on which the instruction is executed. See “CPUID—CPU Identification” in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.
Figure 12-2 shows the cache arrangement of Intel Core i7 processor.

Table 12-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors

<table>
<thead>
<tr>
<th>Cache or Buffer</th>
<th>Characteristics</th>
</tr>
</thead>
</table>
| Trace Cache\(^1\)       | • Pentium 4 and Intel Xeon processors (Based on Intel NetBurst\(^{\text{®}}\) microarchitecture): 12 \(K\)\,μotps, 8-way set associative.  
                          | • Intel Core i7, Intel Core 2 Duo, Intel Atom, Intel Core Solo, Pentium M processor: not implemented.  
                          | • P6 family and Pentium processors: not implemented.                                                                                     |
| L1 Instruction Cache    | • Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): not implemented.  
                          | • Intel Core i7 processor: 32-KByte, 4-way set associative.  
                          | • Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M processor: 32-KByte, 8-way set associative.  
                          | • P6 family and Pentium processors: 8- or 16-KByte, 4-way set associative, 32-byte cache line size; 2-way set associative for earlier Pentium processors. |
| L1 Data Cache           | • Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 8-KByte, 4-way set associative, 64-byte cache line size.  
                          | • Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 16-KByte, 8-way set associative, 64-byte cache line size.  
                          | • Intel Atom processors: 24-KByte, 6-way set associative, 64-byte cache line size.  
                          | • Intel Core i7, Intel Core 2 Duo, Intel Core Duo, Intel Core Solo, Pentium M and Intel Xeon processors: 32-KByte, 8-way set associative, 64-byte cache line size.  
                          | • P6 family processors: 16-KByte, 4-way set associative, 32-byte cache line size; 8-KBytes, 2-way set associative for earlier P6 family processors.  
                          | • Pentium processors: 16-KByte, 4-way set associative, 32-byte cache line size; 8-KByte, 2-way set associative for earlier Pentium processors.  |
### Table 12-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors (Contd.)

<table>
<thead>
<tr>
<th>Cache or Buffer</th>
<th>Characteristics</th>
</tr>
</thead>
</table>
| **L2 Unified Cache** | • Intel Core 2 Duo and Intel Xeon processors: up to 4-MByte (or 4MBx2 in quadcore processors), 16-way set associative, 64-byte cache line size.  
• Intel Core 2 Duo and Intel Xeon processors: up to 6-MByte (or 6MBx2 in quadcore processors), 24-way set associative, 64-byte cache line size.  
• Intel Core i7, i5, i3 processors: 256KB-byte, 8-way set associative, 64-byte cache line size.  
• Intel Atom processors: 512-KByte, 8-way set associative, 64-byte cache line size.  
• Intel Core Duo, Intel Core Solo processors: 2-MByte, 8-way set associative, 64-byte cache line size.  
• Pentium 4 and Intel Xeon processors: 256, 512, 1024, or 2048-KByte, 8-way set associative, 64-byte cache line size, 128-byte sector size.  
• Pentium M processor: 1 or 2-MByte, 8-way set associative, 64-byte cache line size.  
• P6 family processors: 128-KByte, 512-KByte, 1-MByte, or 2-MByte, 4-way set associative, 32-byte cache line size.  
• Pentium processor (external optional): System specific, typically 256- or 512-KByte, 4-way set associative, 32-byte cache line size. |
| **L3 Unified Cache** | • Intel Xeon processors: 512-KByte, 1-MByte, 2-MByte, or 4-MByte, 8-way set associative, 64-byte cache line size, 128-byte sector size.  
• Intel Core i7 processor, Intel Xeon processor 5500: Up to 8MByte, 16-way set associative, 64-byte cache line size.  
• Intel Xeon processor 5600: Up to 12MByte, 64-byte cache line size.  
• Intel Xeon processor 7500: Up to 24MByte, 64-byte cache line size. |
| **Instruction TLB (4-KByte Pages)** | • Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 128 entries, 4-way set associative.  
• Intel Atom processors: 32-entries, fully associative.  
• Intel Core i7, i5, i3 processors: 64-entries per thread (128-entries per core), 4-way set associative.  
• Intel Core 2 Duo, Intel Core Duo, Intel Core Solo processors, Pentium M processor: 128 entries, 4-way set associative.  
• P6 family processors: 32 entries, 4-way set associative.  
• Pentium processor: 32 entries, 4-way set associative; fully set associative for Pentium processors with MMX technology. |
| **Data TLB (4-KByte Pages)** | • Intel Core i7, i5, i3 processors, DTLB0: 64-entries, 4-way set associative.  
• Intel Core 2 Duo processors: DTLB0, 16 entries, DTLB1, 256 entries, 4 ways.  
• Intel Atom processors: 16-entry-per-thread micro-TLB, fully associative; 64-entry DTLB, 4-way set associative.  
• Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 64 entry, fully set associative, shared with large page DTLB.  
• Intel Core Duo, Intel Core Solo processors, Pentium M processor: 128 entries, 4-way set associative.  
• Pentium and P6 family processors: 64 entries, 4-way set associative; fully set associative for Pentium processors with MMX technology. |
| **Instruction TLB (Large Pages)** | • Intel Core i7, i5, i3 processors: 7-entries per thread, fully associative.  
• Intel Core 2 Duo processors: 4 entries, 4 ways.  
• Pentium 4 and Intel Xeon processors: large pages are fragmented.  
• Intel Core Duo, Intel Core Solo, Pentium M processor: 2 entries, fully associative.  
• P6 family processors: 2 entries, fully associative.  
• Pentium processor: Uses same TLB as used for 4-KByte pages. |
| **Data TLB (Large Pages)** | • Intel Core i7, i5, i3 processors, DTLB0: 32-entries, 4-way set associative.  
• Intel Core 2 Duo processors: DTLB0, 16 entries, DTLB1, 32 entries, 4 ways.  
• Intel Atom processors: 8 entries, 4-way set associative.  
• Pentium 4 and Intel Xeon processors: 64 entries, fully set associative; shared with small page data TLBs.  
• Intel Core Duo, Intel Core Solo, Pentium M processor: 8 entries, fully associative.  
• P6 family processors: 8 entries, 4-way set associative.  
• Pentium processor: 8 entries, 4-way set associative; uses same TLB as used for 4-KByte pages in Pentium processors with MMX technology. |
| **Second-level Unified TLB (4-KByte Pages)** | • Intel Core i7, i5, i3 processor, STLB: 512-entries, 4-way set associative. |
MEMORY CACHE CONTROL

Intel 64 and IA-32 processors may implement four types of caches: the trace cache, the level 1 (L1) cache, the level 2 (L2) cache, and the level 3 (L3) cache. See Figure 12-1. Cache availability is described below:

- **Intel Core i7, i5, i3 processor family and Intel Xeon processor family based on Nehalem microarchitecture and Westmere microarchitecture** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache. Each processor core has its own L1 and L2. The L3 cache is an inclusive, unified data and instruction cache, shared by all processor cores inside a physical package. No trace cache is implemented.

- **Intel® Core™ 2 processor family and Intel® Xeon® processor family based on Intel® Core™ microarchitecture** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip; it is shared between two processor cores in a dual-core processor implementation. Quad-core processors have two L2, each shared by two processor cores. No trace cache is implemented.

- **Intel Atom® processor** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip. No trace cache is implemented.

- **Intel® Core™ Solo and Intel® Core™ Duo processors** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip. It is shared between two processor cores in a dual-core processor implementation. No trace cache is implemented.

- **Pentium® 4 and Intel® Xeon® processors Based on Intel NetBurst® microarchitecture** — The trace cache caches decoded instructions (μops) from the instruction decoder and the L1 cache contains data. The L2 and L3 caches are unified data and instruction caches located on the processor chip. Dualcore processors have two L2, one in each processor core. Note that the L3 cache is only implemented on some Intel Xeon processors.

- **P6 family processors** — The L1 cache is divided into two sections: one dedicated to caching instructions (pre-decoded instructions) and the other to caching data. The L2 cache is a unified data and instruction cache located on the processor chip. P6 family processors do not implement a trace cache.

- **Pentium® processors** — The L1 cache has the same structure as on P6 family processors. There is no trace cache. The L2 cache is a unified data and instruction cache external to the processor chip on earlier Pentium processors and implemented on the processor chip in later Pentium processors. For Pentium processors where the L2 cache is external to the processor, access to the cache is through the system bus.

For Intel Core i7 processors and processors based on Intel Core, Intel Atom, and Intel NetBurst microarchitectures, Intel Core Duo, Intel Core Solo and Pentium M processors, the cache lines for the L1 and L2 caches (and L3 caches if supported) are 64 bytes wide. The processor always reads a cache line from system memory beginning on a 64-byte boundary. (A 64-byte aligned cache line begins at an address with its 6 least-significant bits clear.) A cache

### Table 12-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors (Contd.)

<table>
<thead>
<tr>
<th>Cache or Buffer</th>
<th>Characteristics</th>
</tr>
</thead>
</table>
| Store Buffer    | - Intel Core i7, i5, i3 processors: 32 entries.  
- Intel Core 2 Duo processors: 20 entries.  
- Intel Atom processors: 8 entries, used for both WC and store buffers.  
- Pentium 4 and Intel Xeon processors: 24 entries.  
- Pentium M processor: 16 entries.  
- P6 family processors: 12 entries.  
- Pentium processor: 2 buffers, 1 entry each (Pentium processors with MMX technology have 4 buffers for 4 entries). |
| Write Combining (WC) Buffer | - Intel Core 2 Duo processors: 8 entries.  
- Intel Atom processors: 8 entries, used for both WC and store buffers.  
- Pentium 4 and Intel Xeon processors: 6 or 8 entries.  
- Intel Core Duo, Intel Core Solo, Pentium M processors: 6 entries.  
- P6 family processors: 4 entries. |

NOTES:

1. Introduced to the IA-32 architecture in the Pentium 4 and Intel Xeon processors.
line can be filled from memory with a 8-transfer burst transaction. The caches do not support partially-filled cache lines, so caching even a single doubleword requires caching an entire line.

The L1 and L2 cache lines in the P6 family and Pentium processors are 32 bytes wide, with cache line reads from system memory beginning on a 32-byte boundary (5 least-significant bits of a memory address clear.) A cache line can be filled from memory with a 4-transfer burst transaction. Partially-filled cache lines are not supported.

The trace cache in processors based on Intel NetBurst microarchitecture is available in all execution modes: protected mode, system management mode (SMM), and real-address mode. The L1, L2, and L3 caches are also available in all execution modes; however, use of them must be handled carefully in SMM (see Section 32.4.2, “SMRAM Caching”).

The TLBs store the most recently used page-directory and page-table entries. They speed up memory accesses when paging is enabled by reducing the number of memory accesses that are required to read the page tables stored in system memory. The TLBs are divided into four groups: instruction TLBs for 4-KByte pages, data TLBs for 4-KByte pages; instruction TLBs for large pages (2-MByte, 4-MByte or 1-GByte pages), and data TLBs for large pages. The TLBs are normally active only in protected mode with paging enabled. When paging is disabled or the processor is in real-address mode, the TLBs maintain their contents until explicitly or implicitly flushed (see Section 12.9, “Invalidating the Translation Lookaside Buffers (TLBs)”).

Processors based on Intel Core microarchitectures implement one level of instruction TLB and two levels of data TLB. Intel Core i7 processor provides a second-level unified TLB.

The store buffer is associated with the processors instruction execution units. It allows writes to system memory and/or the internal caches to be saved and in some cases combined to optimize the processor’s bus accesses. The store buffer is always enabled in all execution modes.

The processor’s caches are for the most part transparent to software. When enabled, instructions and data flow through these caches without the need for explicit software control. However, knowledge of the behavior of these caches may be useful in optimizing software performance. For example, knowledge of cache dimensions and replacement algorithms gives an indication of how large of a data structure can be operated on at once without causing cache thrashing.

In multiprocessor systems, maintenance of cache consistency may, in rare circumstances, require intervention by system software. For these rare cases, the processor provides privileged cache control instructions for use in flushing caches and forcing memory ordering.

There are several instructions that software can use to improve the performance of the L1, L2, and L3 caches, including the PREFETCHh, CLFLUSH, and CLFLUSHOPT instructions and the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD). The use of these instructions are discussed in Section 12.5.5, “Cache Management Instructions.”

12.2 CACHING TERMINOLOGY

IA-32 processors (beginning with the Pentium processor) and Intel 64 processors use the MESI (modified, exclusive, shared, invalid) cache protocol to maintain consistency with internal caches and caches in other processors (see Section 12.4, “Cache Control Protocol”).

When the processor recognizes that an operand being read from memory is cacheable, the processor reads an entire cache line into the appropriate cache (L1, L2, L3, or all). This operation is called a cache line fill. If the memory location containing that operand is still cached the next time the processor attempts to access the operand, the processor can read the operand from the cache instead of going back to memory. This operation is called a cache hit.

When the processor attempts to write an operand to a cacheable area of memory, it first checks if a cache line for that memory location exists in the cache. If a valid cache line does exist, the processor (depending on the write policy currently in force) can write the operand into the cache instead of writing it out to system memory. This operation is called a write hit. If a write misses the cache (that is, a valid cache line is not present for area of memory being written to), the processor performs a cache line fill, write allocation. Then it writes the operand into the cache line and (depending on the write policy currently in force) can also write it out to memory. If the operand is to be written out to memory, it is written first into the store buffer, and then written from the store buffer to memory when the system bus is available. (Note that for the Pentium processor, write misses do not result in a cache line fill; they always result in a write to memory. For this processor, only read misses result in cache line fills.)
MEMORY CACHE CONTROL

When operating in an MP system, IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors have the ability to snoop other processor’s accesses to system memory and to their internal caches. They use this snooping ability to keep their internal caches consistent both with system memory and with the caches in other processors on the bus. For example, in the Pentium and P6 family processors, if through snooping one processor detects that another processor intends to write to a memory location that it currently has cached in shared state, the snooping processor will invalidate its cache line forcing it to perform a cache line fill the next time it accesses the same memory location.

Beginning with the P6 family processors, if a processor detects (through snooping) that another processor is trying to access a memory location that it has modified in its cache, but has not yet written back to system memory, the snooping processor will signal the other processor (by means of the HITM# signal) that the cache line is held in modified state and will perform an implicit write-back of the modified data. The implicit write-back is transferred directly to the initial requesting processor and snooped by the memory controller to assure that system memory has been updated. Here, the processor with the valid data may pass the data to the other processors without actually writing it to system memory; however, it is the responsibility of the memory controller to snoop this operation and update memory.

12.3 METHODS OF CACHING AVAILABLE

The processor allows any area of system memory to be cached in the L1, L2, and L3 caches. In individual pages or regions of system memory, it allows the type of caching (also called memory type) to be specified (see Section 12.5). Memory types currently defined for the Intel 64 and IA-32 architectures are (see Table 12-2):

- **Strong Uncacheable (UC)** — System memory locations are not cached. All reads and writes appear on the system bus and are executed in program order without reordering. No speculative memory accesses, page-table walks, or prefetches of speculated branch targets are made. This type of cache-control is useful for memory-mapped I/O devices. When used with normal RAM, it greatly reduces processor performance.

  **NOTE**

  The behavior of x87 and SIMD instructions referencing memory is implementation dependent. In some implementations, accesses to UC memory may occur more than once. To ensure predictable behavior, use loads and stores of general purpose registers to access UC memory that may have read or write side effects.

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Uncacheable (UC)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Strong Ordering</td>
</tr>
<tr>
<td>Uncacheable (UC-)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Strong Ordering. Can only be selected through the PAT. Can be overridden by WC in MTRRs.</td>
</tr>
<tr>
<td>Write Combining (WC)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.</td>
</tr>
<tr>
<td>Write Through (WT)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
<tr>
<td>Write Protected (WP)</td>
<td>Yes for reads; no for writes</td>
<td>No</td>
<td>Yes</td>
<td>Speculative Processor Ordering. Available by programming MTRRs.</td>
</tr>
</tbody>
</table>

- **Uncacheable (UC-)** — Has same characteristics as the strong uncacheable (UC) memory type, except that this memory type can be overridden by programming the MTRRs for the WC memory type. This memory type is available in processor families starting from the Pentium III processors and can only be selected through the PAT.
• **Write Combining (WC)** — System memory locations are not cached (as with uncacheable memory) and coherency is not enforced by the processor’s bus coherency protocol. Speculative reads are allowed. Writes may be delayed and combined in the write combining buffer (WC buffer) to reduce memory accesses. If the WC buffer is partially filled, the writes may be delayed until the next occurrence of a serializing event; such as an SFENCE or MFENCE instruction, CPUID or other serializing instruction, a read or write to uncacheable memory, an interrupt occurrence, or an execution of a LOCK instruction (including one with an XACQUIRE or XRELEASE prefix). In addition, an execution of the XEND instruction (to end a transactional region) evicts any writes that were buffered before the corresponding execution of the XBEGIN instruction (to begin the transactional region) before evicting any writes that were performed inside the transactional region. This type of cache-control is appropriate for video frame buffers, where the order of writes is unimportant as long as the writes update memory so they can be seen on the graphics display. See Section 12.3.1, “Buffering of Write Combining Memory Locations,” for more information about caching the WC memory type. This memory type is available in the Pentium Pro and Pentium II processors by programming the MTRRs; or in processor families starting from the Pentium III processors by programming the MTRRs or by selecting it through the PAT.

• **Write-through (WT)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. All writes are written to a cache line (when possible) and through to system memory. When writing through to memory, invalid cache lines are never filled, and valid cache lines are either filled or invalidated. Write combining is allowed. This type of cache-control is appropriate for frame buffers or when there are devices on the system bus that access system memory, but do not perform snooping of memory accesses. It enforces coherency between caches in the processors and system memory.

• **Write-back (WB)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. Write misses cause cache line fills (in processor families starting with the P6 family processors), and writes are performed entirely in the cache, when possible. Write combining is allowed. The write-back memory type reduces bus traffic by eliminating many unnecessary writes to system memory. Writes to a cache line are not immediately forwarded to system memory; instead, they are accumulated in the cache. The modified cache lines are written to system memory later, when a write-back operation is performed. Write-back operations are triggered when cache lines need to be deallocated, such as when new cache lines are being allocated in a cache that is already full. They also are triggered by the mechanisms used to maintain cache consistency. This type of cache-control provides the best performance, but it requires that all devices that access system memory on the system bus be able to snoop memory accesses to ensure system memory and cache coherency.

• **Write protected (WP)** — Reads come from cache lines when possible, and read misses cause cache fills. Writes are propagated to the system bus and cause corresponding cache lines on all processors on the bus to be invalidated. Speculative reads are allowed. This memory type is available in processor families starting from the P6 family processors by programming the MTRRs (see Table 12-6).

Table 12-3 shows which of these caching methods are available in the Pentium, P6 Family, Pentium 4, and Intel Xeon processors.

**Table 12-3. Methods of Caching Available in Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 Family, and Pentium Processors**

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4 and Intel Xeon Processors</th>
<th>P6 Family Processors</th>
<th>Pentium Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Uncacheable (UC)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Uncacheable (UC-)</td>
<td>Yes</td>
<td>Yes*</td>
<td>No</td>
</tr>
<tr>
<td>Write Combining (WC)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Write Through (WT)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Write Protected (WP)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**NOTE:**
* Introduced in the Pentium III processor; not available in the Pentium Pro or Pentium II processors
12.3.1 Buffering of Write Combining Memory Locations

Writes to the WC memory type are not cached in the typical sense of the word cached. They are retained in an internal write combining buffer (WC buffer) that is separate from the internal L1, L2, and L3 caches and the store buffer. The WC buffer is not snooped and thus does not provide data coherency. Buffering of writes to WC memory is done to allow software a small window of time to supply more modified data to the WC buffer while remaining as non-intrusive to software as possible. The buffering of writes to WC memory also causes data to be collapsed; that is, multiple writes to the same memory location will leave the last data written in the location and the other writes will be lost.

The size and structure of the WC buffer is not architecturally defined. For the Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4 and Intel Xeon processors; the WC buffer is made up of several 64-byte WC buffers. For the P6 family processors, the WC buffer is made up of several 32-byte WC buffers.

When software begins writing to WC memory, the processor begins filling the WC buffers one at a time. When one or more WC buffers has been filled, the processor has the option of evicting the buffers to system memory. The protocol for evicting the WC buffers is implementation dependent and should not be relied on by software for system memory coherency. When using the WC memory type, software must be sensitive to the fact that the writing of data to system memory is being delayed and must deliberately empty the WC buffers when system memory coherency is required.

Once the processor has started to evict data from the WC buffer into system memory, it will make a bus-transaction style decision based on how much of the buffer contains valid data. If the buffer is full (for example, all bytes are valid), the processor will execute a burst-write transaction on the bus. This results in all 32 bytes (P6 family processors) or 64 bytes (Pentium 4 and more recent processor) being transmitted on the data bus in a single burst transaction. If one or more of the WC buffer’s bytes are invalid (for example, have not been written by software), the processor will transmit the data to memory using “partial write” transactions (one chunk at a time, where a “chunk” is 8 bytes).

This will result in a maximum of 4 partial write transactions (for P6 family processors) or 8 partial write transactions (for the Pentium 4 and more recent processors) for one WC buffer of data sent to memory.

The WC memory type is weakly ordered by definition. Once the eviction of a WC buffer has started, the data is subject to the weak ordering semantics of its definition. Ordering is not maintained between the successive allocation/deallocation of WC buffers (for example, writes to WC buffer 1 followed by writes to WC buffer 2 may appear as buffer 2 followed by buffer 1 on the system bus). When a WC buffer is evicted to memory as partial writes there is no guaranteed ordering between successive partial writes (for example, a partial write for chunk 2 may appear on the bus before the partial write for chunk 1 or vice versa).

The only elements of WC propagation to the system bus that are guaranteed are those provided by transaction atomicity. For example, with a P6 family processor, a completely full WC buffer will always be propagated as a single 32-bit burst transaction using any chunk order. In a WC buffer eviction where data will be evicted as partials, all data contained in the same chunk (0 mod 8 aligned) will be propagated simultaneously. Likewise, for more recent processors starting with those based on Intel NetBurst microarchitectures, a full WC buffer will always be propagated as a single burst transactions, using any chunk order within a transaction. For partial buffer propagations, all data contained in the same chunk will be propagated simultaneously.

12.3.2 Choosing a Memory Type

The simplest system memory model does not use memory-mapped I/O with read or write side effects, does not include a frame buffer, and uses the write-back memory type for all memory. An I/O agent can perform direct memory access (DMA) to write-back memory and the cache protocol maintains cache coherency.

A system can use strong uncacheable memory for other memory-mapped I/O, and should always use strong uncacheable memory for frame buffers or dual-ported memory that contains pixel values displayed on a screen. Frame buffer memory is typically large (a few megabytes) and is usually written more than it is read by the processor. Using strong uncacheable memory for a frame buffer generates very large amounts of bus traffic, because operations on the entire buffer are implemented using partial writes rather than line writes. Using write-through memory for a frame buffer can displace almost all other useful cached
lines in the processor's L2 and L3 caches and L1 data cache. Therefore, systems should use write-combining memory for frame buffers whenever possible.

Software can use page-level cache control, to assign appropriate effective memory types when software will not access data structures in ways that benefit from write-back caching. For example, software may read a large data structure once and not access the structure again until the structure is rewritten by another agent. Such a large data structure should be marked as uncacheable, or reading it will evict cached lines that the processor will be referencing again.

A similar example would be a write-only data structure that is written to (to export the data to another agent), but never read by software. Such a structure can be marked as uncacheable, because software never reads the values that it writes (though as uncacheable memory, it will be written using partial writes, while as write-back memory, it will be written using line writes, which may not occur until the other agent reads the structure and triggers implicit write-backs).

On the Pentium III, Pentium 4, and more recent processors, new instructions are provided that give software greater control over the caching, prefetching, and the write-back characteristics of data. These instructions allow software to use weakly ordered or processor ordered memory types to improve processor performance, but when necessary to force strong ordering on memory reads and/or writes. They also allow software greater control over the caching of data. For a description of these instructions and their intended use, see Section 12.5.5, "Cache Management Instructions."

### 12.3.3 Code Fetches in Uncacheable Memory

Programs may execute code from uncacheable (UC) memory, but the implications are different from accessing data in UC memory. When doing code fetches, the processor never transitions from cacheable code to UC code speculatively. It also never speculatively fetches branch targets that result in UC code.

The processor may fetch the same UC cache line multiple times in order to decode an instruction once. It may decode consecutive UC instructions in a cache line without fetching between each instruction. It may also fetch additional cachelines from the same or a consecutive 4-KByte page in order to decode one non-speculative UC instruction (this can be true even when the instruction is contained fully in one line).

Because of the above and because cache line sizes may change in future processors, software should avoid placing memory-mapped I/O with read side effects in the same page or in a subsequent page used to execute UC code.

### 12.4 CACHE CONTROL PROTOCOL

The following section describes the cache control protocol currently defined for the Intel 64 and IA-32 architectures.

In the L1 data cache and in the L2/L3 unified caches, the MESI (modified, exclusive, shared, invalid) cache protocol maintains consistency with caches of other processors. The L1 data cache and the L2/L3 unified caches have two MESI status flags per cache line. Each line can be marked as being in one of the states defined in Table 12-4. In general, the operation of the MESI protocol is transparent to programs.

<table>
<thead>
<tr>
<th>Cache Line State</th>
<th>M (Modified)</th>
<th>E (Exclusive)</th>
<th>S (Shared)</th>
<th>I (Invalid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This cache line is valid?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>The memory copy is...</td>
<td>Out of date</td>
<td>Valid</td>
<td>Valid</td>
<td>—</td>
</tr>
<tr>
<td>Copies exist in caches of other</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>processors?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A write to this line ...</td>
<td>Does not go to the system bus.</td>
<td>Does not go to the system bus.</td>
<td>Causes the processor to gain exclusive ownership of the line.</td>
<td>Goes directly to the system bus.</td>
</tr>
</tbody>
</table>
The L1 instruction cache in P6 family processors implements only the “SI” part of the MESI protocol, because the instruction cache is not writable. The instruction cache monitors changes in the data cache to maintain consistency between the caches when instructions are modified. See Section 12.6, “Self-Modifying Code,” for more information on the implications of caching instructions.

12.5 CACHE CONTROL

The Intel 64 and IA-32 architectures provide a variety of mechanisms for controlling the caching of data and instructions and for controlling the ordering of reads and writes between the processor, the caches, and memory. These mechanisms can be divided into two groups:

- **Cache control registers and bits** — The Intel 64 and IA-32 architectures define several dedicated registers and various bits within control registers and page- and directory-table entries that control the caching system memory locations in the L1, L2, and L3 caches. These mechanisms control the caching of virtual memory pages and of regions of physical memory.

- **Cache control and memory ordering instructions** — The Intel 64 and IA-32 architectures provide several instructions that control the caching of data, the ordering of memory reads and writes, and the prefetching of data. These instructions allow software to control the caching of specific data structures, to control memory coherency for specific locations in memory, and to force strong memory ordering at specific locations in a program.

The following sections describe these two groups of cache control mechanisms.

12.5.1 Cache Control Registers and Bits

Figure 12-3 depicts cache-control mechanisms in IA-32 processors. Other than for the matter of memory address space, these work the same in Intel 64 processors.

The Intel 64 and IA-32 architectures provide the following cache-control registers and bits for use in enabling or restricting caching to various pages or regions in memory:

- **CD flag, bit 30 of control register CR0** — Controls caching of system memory locations (see Section 2.5, “Control Registers”). If the CD flag is clear, caching is enabled for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. When the CD flag is set, caching is restricted in the processor’s caches (cache hierarchy) for the P6 and more recent processor families and prevented for the Pentium processor (see note below). With the CD flag set, however, the caches will still respond to snoop traffic. Caches should be explicitly flushed to ensure memory coherency. For highest processor performance, both the CD and the NW flags in control register CR0 should be cleared. Table 12-5 shows the interaction of the CD and NW flags.

The effect of setting the CD flag is somewhat different for processor families starting with P6 family than the Pentium processor (see Table 12-5). To ensure memory coherency after the CD flag is set, the caches should be explicitly flushed (see Section 12.5.3, “Preventing Caching”). Setting the CD flag for the P6 and more recent processor families modifies cache line fill and update behavior. Also, setting the CD flag on these processors do not force strict ordering of memory accesses unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 9.2.5, “Strengthening or Weakening the Memory-Ordering Model”).
CD and NW Flags control overall caching of system memory

Figure 12-3. Cache-Control Registers and Bits Available in Intel 64 and IA-32 Processors

1. G flag only available in P6 and later processor families
2. The maximum physical address size is reported by CPUID leaf function 80000008H. The maximum physical address size of FFFFFFFFH applies only if 36-bit physical addressing is used.
3. MTRRs available only in P6 and later processor families; similar control available in Pentium processor with the KEN# and WB/WT# pins.
4. PAT available only in Pentium III and later processor families.
5. L3 in processors based on Intel NetBurst microarchitecture can be disabled using IA32_MISC_ENABLE MSR.
### Table 12-5. Cache Operating Modes

<table>
<thead>
<tr>
<th>CD</th>
<th>NW</th>
<th>Caching and Read/Write Policy</th>
<th>L1</th>
<th>L2/L3¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Normal Cache Mode. Highest performance cache operation.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Read hits access the cache; read misses may cause replacement.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write hits update the cache.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only writes to shared lines and write misses update system memory.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write misses cause cache line fills.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write hits can change shared lines to modified under control of the MTRRs and with associated read invalidation cycle.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (Pentium processor only.) Write misses do not cause cache line fills.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (Pentium processor only.) Write hits can change shared lines to exclusive under control of the WB/WT#.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Invalidation is allowed.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• External snoop traffic is supported.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Invalid setting. Generates a general-protection exception (#GP) with an error code of 0.</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>No-fill Cache Mode. Memory coherency is maintained.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (Pentium 4 and later processor families.) State of processor after a power up or reset.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Read hits access the cache; read misses do not cause replacement (see Pentium 4 and Intel Xeon processors reference below).</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write hits update the cache.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only writes to shared lines and write misses update system memory.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write misses access memory.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write hits can change shared lines to exclusive under control of the MTRRs and with associated read invalidation cycle.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (Pentium processor only.) Write hits can change shared lines to exclusive under control of the WB/WT#.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (P6 and later processor families only.) Strict memory ordering is not enforced unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 7.2.4., &quot;Strengthening or Weakening the Memory Ordering Model&quot;).</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Invalidation is allowed.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• External snoop traffic is supported.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Memory coherency is not maintained.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (P6 family and Pentium processors.) State of the processor after a power up or reset.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Read hits access the cache; read misses do not cause replacement.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write hits update the cache and change exclusive lines to modified.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Shared lines remain shared after write hit.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write misses access memory.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Invalidation is inhibited when snooping; but is allowed with INVD and WBINVD instructions.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• External snoop traffic is supported.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The L2/L3 column in this table is definitive for the Pentium 4, Intel Xeon, and P6 family processors. It is intended to represent what could be implemented in a system based on a Pentium processor with an external, platform specific, write-back L2 cache.

2. The Pentium 4 and more recent processor families do not support this mode; setting the CD and NW bits to 1 selects the no-fill cache mode.

3. Not supported In Intel Atom processors. If CD = 1 in an Intel Atom processor, caching is disabled.
• **NW flag, bit 29 of control register CR0** — Controls the write policy for system memory locations (see Section 2.5, “Control Registers”). If the NW and CD flags are clear, write-back is enabled for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. Table 12-5 shows how the other combinations of CD and NW flags affects caching.

**NOTES**

For the Pentium 4 and Intel Xeon processors, the NW flag is a don’t care flag; that is, when the CD flag is set, the processor uses the no-fill cache mode, regardless of the setting of the NW flag.

For Intel Atom processors, the NW flag is a don’t care flag; that is, when the CD flag is set, the processor disables caching, regardless of the setting of the NW flag.

For the Pentium processor, when the L1 cache is disabled (the CD and NW flags in control register CR0 are set), external snoops are accepted in DP (dual-processor) systems and inhibited in uniprocessor systems.

When snoops are inhibited, address parity is not checked and APCHK# is not asserted for a corrupt address; however, when snoops are accepted, address parity is checked and APCHK# is asserted for corrupt addresses.

• **PCD and PWT flags in paging-structure entries** — Control the memory type used to access paging structures and pages (see Section 4.9, “Paging and Memory Typing”).

• **PCD and PWT flags in control register CR3** — Control the memory type used to access the first paging structure of the current paging-structure hierarchy (see Section 4.9, “Paging and Memory Typing”).

• **G (global) flag in the page-directory and page-table entries (introduced to the IA-32 architecture in the P6 family processors)** — Controls the flushing of TLB entries for individual pages. See Section 4.10, “Caching Translation Information,” for more information about this flag.

• **PGE (page global enable) flag in control register CR4** — Enables the establishment of global pages with the G flag. See Section 4.10, “Caching Translation Information,” for more information about this flag.

• **Memory type range registers (MTRRs) (introduced in P6 family processors)** — Control the type of caching used in specific regions of physical memory. Any of the caching types described in Section 12.3, “Methods of Caching Available,” can be selected. See Section 12.11, “Memory Type Range Registers (MTRRs),” for a detailed description of the MTRRs.

• **Page Attribute Table (PAT) MSR (introduced in the Pentium III processor)** — Extends the memory typing capabilities of the processor to permit memory types to be assigned on a page-by-page basis (see Section 12.12, “Page Attribute Table (PAT)”).

• **Third-Level Cache Disable flag, bit 6 of the IA32_MISC_ENABLE MSR (Available only in processors based on Intel NetBurst microarchitecture)** — Allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches.

• **KEN# and WB/WT# pins (Pentium processor)** — Allow external hardware to control the caching method used for specific areas of memory. They perform similar (but not identical) functions to the MTRRs in the P6 family processors.

• **PCD and PWT pins (Pentium processor)** — These pins (which are associated with the PCD and PWT flags in control register CR3 and in the page-directory and page-table entries) permit caching in an external L2 cache to be controlled on a page-by-page basis, consistent with the control exercised on the L1 cache of these processors. The P6 and more recent processor families do not provide these pins because the L2 cache in internal to the chip package.

12.5.2 **Precedence of Cache Controls**

The cache control flags and MTRRs operate hierarchically for restricting caching. That is, if the CD flag is set, caching is prevented globally (see Table 12–5). If the flag is clear, the page-level cache control flags and/or the MTRRs can be used to restrict caching. If there is an overlap of page-level and MTRR caching controls, the mechanism that prevents caching has precedence. For example, if an MTRR makes a region of system memory uncacheable, a page-level caching control cannot be used to enable caching for a page in that region. The converse is also
true; that is, if a page-level caching control designates a page as uncacheable, an MTRR cannot be used to make the page cacheable.

In cases where there is an overlap in the assignment of the write-back and write-through caching policies to a page and a region of memory, the write-through policy takes precedence. The write-combining policy (which can only be assigned through an MTRR or the PAT) takes precedence over either write-through or write-back.

The selection of memory types at the page level varies depending on whether PAT is being used to select memory types for pages, as described in the following sections.

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. Using IA32_MISC_ENABLE[bit 6] takes precedence over the CD flag, MTRRs, and PAT for the L3 cache in those processors. That is, when the third-level cache disable flag is set (cache disabled), the other cache controls have no affect on the L3 cache; when the flag is clear (enabled), the cache controls have the same affect on the L3 cache as they have on the L1 and L2 caches.

IA32_MISC_ENABLE[bit 6] is not supported in Intel Core i7 processors, nor processors based on Intel Core, and Intel Atom microarchitectures.

### 12.5.2.1 Selecting Memory Types for Pentium Pro and Pentium II Processors

The Pentium Pro and Pentium II processors do not support the PAT. Here, the effective memory type for a page is selected with the MTRRs and the PCD and PWT bits in the page-table or page-directory entry for the page. Table 12-6 describes the mapping of MTRR memory types and page-level caching attributes to effective memory types, when normal caching is in effect (the CD and NW flags in control register CR0 are clear). Combinations that appear in gray are implementation-defined for the Pentium Pro and Pentium II processors. System designers are encouraged to avoid these implementation-defined combinations.

<table>
<thead>
<tr>
<th>MTRR Memory Type</th>
<th>PCD Value</th>
<th>PWT Value</th>
<th>Effective Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>X</td>
<td>X</td>
<td>UC</td>
</tr>
<tr>
<td>WC</td>
<td>0</td>
<td>0</td>
<td>WC</td>
</tr>
<tr>
<td>WC</td>
<td>0</td>
<td>1</td>
<td>WC</td>
</tr>
<tr>
<td>WC</td>
<td>1</td>
<td>0</td>
<td>WC</td>
</tr>
<tr>
<td>WC</td>
<td>1</td>
<td>1</td>
<td>WC</td>
</tr>
<tr>
<td>WT</td>
<td>0</td>
<td>X</td>
<td>WT</td>
</tr>
<tr>
<td>WT</td>
<td>1</td>
<td>X</td>
<td>UC</td>
</tr>
<tr>
<td>WP</td>
<td>0</td>
<td>0</td>
<td>WP</td>
</tr>
<tr>
<td>WP</td>
<td>0</td>
<td>1</td>
<td>WP</td>
</tr>
<tr>
<td>WP</td>
<td>1</td>
<td>0</td>
<td>WP</td>
</tr>
<tr>
<td>WP</td>
<td>1</td>
<td>1</td>
<td>UC</td>
</tr>
<tr>
<td>WB</td>
<td>0</td>
<td>0</td>
<td>WB</td>
</tr>
<tr>
<td>WB</td>
<td>0</td>
<td>1</td>
<td>WT</td>
</tr>
<tr>
<td>WB</td>
<td>1</td>
<td>X</td>
<td>UC</td>
</tr>
</tbody>
</table>

**NOTE:**
1. These effective memory types also apply to the Pentium 4, Intel Xeon, and Pentium III processors when the PAT bit is not used (set to 0) in page-table and page-directory entries.

When normal caching is in effect, the effective memory type shown in Table 12-6 is determined using the following rules:
1. If the PCD and PWT attributes for the page are both 0, then the effective memory type is identical to the MTRR-defined memory type.
2. If the PCD flag is set, then the effective memory type is UC.
3. If the PCD flag is clear and the PWT flag is set, the effective memory type is WT for the WB memory type and the MTRR-defined memory type for all other memory types.
4. Setting the PCD and PWT flags to opposite values is considered model-specific for the WP and WC memory types and architecturally-defined for the WB, WT, and UC memory types.

12.5.2.2 Selecting Memory Types for Pentium III and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M, Pentium 4, Intel Xeon, and Pentium III processors use the PAT to select effective page-level memory types. Here, a memory type for a page is selected by the MTRRs and the value in a PAT entry that is selected with the PAT, PCD, and PWT bits in a page-table or page-directory entry (see Section 12.12.3, “Selecting a Memory Type from the PAT”). Table 12-7 describes the mapping of MTRR memory types and PAT entry types to effective memory types, when normal caching is in effect (the CD and NW flags in control register CR0 are clear).

Table 12-7. Effective Page-Level Memory Types for Pentium III and More Recent Processor Families

<table>
<thead>
<tr>
<th>MTRR Memory Type</th>
<th>PAT Entry Value</th>
<th>Effective Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>UC-</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>UC</td>
</tr>
<tr>
<td>WC</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>UC-</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>WC</td>
</tr>
<tr>
<td>WT</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>UC-</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>WT</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>WT</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>WP</td>
</tr>
<tr>
<td>WB</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>UC-</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>WT</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>WP</td>
</tr>
</tbody>
</table>
12.5.2.3 Writing Values Across Pages with Different Memory Types

If two adjoining pages in memory have different memory types, and a word or longer operand is written to a memory location that crosses the page boundary between those two pages, the operand might be written to memory twice. This action does not present a problem for writes to actual memory; however, if a device is mapped to the memory space assigned to the pages, the device might malfunction.

12.5.3 Preventing Caching

To disable the L1, L2, and L3 caches after they have been enabled and have received cache fills, perform the following steps:

1. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.
2. Flush all caches using the WBINVD instruction.
3. Disable the MTRRs and set the default memory type to uncached or set all MTRRs for the uncached memory type (see the discussion of the discussion of the TYPE field and the E flag in Section 12.11.2.1, “IA32_MTRR_DEF_TYPE MSR”).

The caches must be flushed (step 2) after the CD flag is set to ensure system memory coherency. If the caches are not flushed, cache hits on reads will still occur and data will be read from valid cache lines.

The intent of the three separate steps listed above address three distinct requirements: (i) discontinue new data replacing existing data in the cache (ii) ensure data already in the cache are evicted to memory, (iii) ensure subsequent memory references observe UC memory type semantics. Different processor implementation of caching control hardware may allow some variation of software implementation of these three requirements. See note below.

NOTES

1. The UC attribute comes from the MTRRs and the processors are not required to snoop their caches since the data could never have been cached. This attribute is preferred for performance reasons.
2. The UC attribute came from the page-table or page-directory entry and processors are required to check their caches because the data may be cached due to page aliasing, which is not recommended.
3. These combinations were specified as “undefined” in previous editions of the Intel® 64 and IA-32 Architectures Software Developer’s Manual. However, all processors that support both the PAT and the MTRRs determine the effective page-level memory types for these combinations as given.
remove code from the cache completely, a second WBINVD instruction must be executed after the MTRRs have been disabled.

For Intel Atom processors, setting the CD flag forces all physical memory to observe UC semantics (without requiring memory type of physical memory to be set explicitly). Consequently, software does not need to issue a second WBINVD as some other processor generations might require.

12.5.4 Disabling and Enabling the L3 Cache

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag (bit 6 of the IA32_MISC_ENABLE MSR) allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches. Prior to using this control to disable or enable the L3 cache, software should disable and flush all the processor caches, as described earlier in Section 12.5.3, “Preventing Caching,” to prevent of loss of information stored in the L3 cache. After the L3 cache has been disabled or enabled, caching for the whole processor can be restored.

Newer Intel 64 processor with L3 do not support IA32_MISC_ENABLE[bit 6], the procedure described in Section 12.5.3, “Preventing Caching,” apply to the entire cache hierarchy.

12.5.5 Cache Management Instructions

The Intel 64 and IA-32 architectures provide several instructions for managing the L1, L2, and L3 caches. The INVD and WBINVD instructions are privileged instructions and operate on the L1, L2, and L3 caches as a whole. The PREFETCHh, CLFLUSH, and CLFLUSHOPT instructions and the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD) offer more granular control over caching, and are available to all privileged levels.

The INVD and WBINVD instructions are used to invalidate the contents of the L1, L2, and L3 caches. The INVD instruction invalidates all internal cache entries, then generates a special-function bus cycle that indicates that external caches also should be invalidated. The INVD instruction should be used with care. It does not force a write-back of modified cache lines; therefore, data stored in the caches and not written back to system memory will be lost. Unless there is a specific requirement or benefit to invalidating the caches without writing back the modified lines (such as, during testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

The WBINVD instruction first writes back any modified lines in all the internal caches, then invalidates the contents of the L1, L2, and L3 caches. It ensures that cache coherency with main memory is maintained regardless of the write policy in effect (that is, write-through or write-back). Following this operation, the WBINVD instruction generates one (P6 family processors) or two (Pentium and Intel486 processors) special-function bus cycles to indicate to external cache controllers that write-back of modified data followed by invalidation of external caches should occur. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.

The PREFETCHh instructions allow a program to suggest to the processor that a cache line from a specified location in system memory be prefetched into the cache hierarchy (see Section 12.8, “Explicit Caching”).

The CLFLUSH and CLFLUSHOPT instructions allow selected cache lines to be flushed from memory. These instructions give a program the ability to explicitly free up cache space, when it is known that cached section of system memory will not be accessed in the near future.

The non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD) allow data to be moved from the processor’s registers directly into system memory without being also written into the L1, L2, and/or L3 caches. These instructions can be used to prevent cache pollution when operating on data that is going to be modified only once before being stored back into system memory. These instructions operate on data in the general-purpose, MMX, and XMM registers.
12.5.6  L1 Data Cache Context Mode

L1 data cache context mode is a feature of processors based on the Intel NetBurst microarchitecture that support Intel Hyper-Threading Technology. When CPUID.1:ECX[bit 10] = 1, the processor supports setting L1 data cache context mode using the L1 data cache context mode flag (IA32_MISC_ENABLE[bit 24]). Selectable modes are adaptive mode (default) and shared mode.

The BIOS is responsible for configuring the L1 data cache context mode.

12.5.6.1  Adaptive Mode

Adaptive mode facilitates L1 data cache sharing between logical processors. When running in adaptive mode, the L1 data cache is shared across logical processors in the same core if:

- CR3 control registers for logical processors sharing the cache are identical.
- The same paging mode is used by logical processors sharing the cache.

In this situation, the entire L1 data cache is available to each logical processor (instead of being competitively shared).

If CR3 values are different for the logical processors sharing an L1 data cache or the logical processors use different paging modes, processors compete for cache resources. This reduces the effective size of the cache for each logical processor. Aliasing of the cache is not allowed (which prevents data thrashing).

12.5.6.2  Shared Mode

In shared mode, the L1 data cache is competitively shared between logical processors. This is true even if the logical processors use identical CR3 registers and paging modes.

In shared mode, linear addresses in the L1 data cache can be aliased, meaning that one linear address in the cache can point to different physical locations. The mechanism for resolving aliasing can lead to thrashing. For this reason, IA32_MISC_ENABLE[bit 24] = 0 is the preferred configuration for processors based on the Intel NetBurst microarchitecture that support Intel Hyper-Threading Technology.

12.6  SELF-MODIFYING CODE

A write to a memory location in a code segment that is currently cached in the processor causes the associated cache line (or lines) to be invalidated. This check is based on the physical address of the instruction. In addition, the P6 family and Pentium processors check whether a write to a code segment may modify an instruction that has been prefetched for execution. If the write affects a prefetched instruction, the prefetch queue is invalidated. This latter check is based on the linear address of the instruction. For the Pentium 4 and Intel Xeon processors, a write or a snoop of an instruction in a code segment, where the target instruction is already decoded and resident in the trace cache, invalidates the entire trace cache. The latter behavior means that programs that self-modify code can cause severe degradation of performance when run on the Pentium 4 and Intel Xeon processors.

In practice, the check on linear addresses should not create compatibility problems among IA-32 processors. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. Systems software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction, will execute a serializing operation, such as a CPUID instruction, before the modified instruction is executed, which will automatically resynchronize the instruction cache and prefetch queue. (See Section 9.1.3, "Handling Self- and Cross-Modifying Code," for more information about the use of self-modifying code.)

For Intel486 processors, a write to an instruction in the cache will modify it in both the cache and memory, but if the instruction was prefetched before the write, the old version of the instruction could be the one executed. To prevent the old instruction from being executed, flush the instruction prefetch unit by coding a jump instruction immediately after any write that modifies an instruction.
12.7 IMPLICIT CACHING (PENTIUM 4, INTEL® XEON®, AND P6 FAMILY PROCESSORS)

Implicit caching occurs when a memory element is made potentially cacheable, although the element may never have been accessed in the normal von Neumann sequence. Implicit caching occurs on the P6 and more recent processor families due to aggressive prefetching, branch prediction, and TLB miss handling. Implicit caching is an extension of the behavior of existing Intel386, Intel486, and Pentium processor systems, since software running on these processor families also has not been able to deterministically predict the behavior of instruction prefetch.

To avoid problems related to implicit caching, the operating system must explicitly invalidate the cache when changes are made to cacheable data that the cache coherency mechanism does not automatically handle. This includes writes to dual-ported or physically aliased memory boards that are not detected by the snooping mechanisms of the processor, and changes to page-table entries in memory.

The code in Example 12-1 shows the effect of implicit caching on page-table entries. The linear address F000H points to physical location B000H (the page-table entry for F000H contains the value B000H), and the page-table entry for linear address F000 is PTE_F000.

Example 12-1. Effect of Implicit Caching on Page-Table Entries

```assembly
mov EAX, CR3; Invalidate the TLB
mov CR3, EAX; by copying CR3 to itself
mov PTE_F000, A000H; Change F000H to point to A000H
mov EBX, [F000H];
```

Because of speculative execution in the P6 and more recent processor families, the last MOV instruction performed would place the value at physical location B000H into EBX, rather than the value at the new physical address A000H. This situation is remedied by placing a TLB invalidation between the load and the store.

12.8 EXPLICIT CACHING

The Pentium III processor introduced four new instructions, the PREFETCHh instructions, that provide software with explicit control over the caching of data. These instructions provide "hints" to the processor that the data requested by a PREFETCHh instruction should be read into cache hierarchy now or as soon as possible, in anticipation of its use. The instructions provide different variations of the hint that allow selection of the cache level into which data will be read.

The PREFETCHh instructions can help reduce the long latency typically associated with reading data from memory and thus help prevent processor "stalls." However, these instructions should be used judiciously. Overuse can lead to resource conflicts and hence reduce the performance of an application. Also, these instructions should only be used to prefetch data from memory; they should not be used to prefetch instructions. For more detailed information on the proper use of the prefetch instruction, refer to Chapter 7, "Optimizing Cache Usage," in the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

12.9 INVALIDATING THE TRANSLATION LOOKASIDE BUFFERS (TLBs)

The processor updates its address translation caches (TLBs) transparently to software. Several mechanisms are available, however, that allow software and hardware to invalidate the TLBs either explicitly or as a side effect of another operation. Most details are given in Section 4.10.4, "Invalidation of TLBs and Paging-Structure Caches." In addition, the following operations invalidate all TLB entries, irrespective of the setting of the G flag:

- Asserting or de-asserting the FLUSH# pin.
- (Pentium 4, Intel Xeon, and later processors only.) Writing to an MTRR (with a WRMSR instruction).
- Writing to control register CR0 to modify the PG or PE flag.
MEMORY CACHE CONTROL

- (Pentium 4, Intel Xeon, and later processors only.) Writing to control register CR4 to modify the PSE, PGE, or PAE flag.
- Writing to control register CR4 to change the PCIDE flag from 1 to 0.

See Section 4.10, “Caching Translation Information,” for additional information about the TLBs.

12.10 STORE BUFFER

Intel 64 and IA-32 processors temporarily store each write (store) to memory in a store buffer. The store buffer improves processor performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or to a cache is complete. It also allows writes to be delayed for more efficient use of memory-access bus cycles.

In general, the existence of the store buffer is transparent to software, even in systems that use multiple processors. The processor ensures that write operations are always carried out in program order. It also ensures that the contents of the store buffer are always drained to memory in the following situations:
- When an exception or interrupt is generated.
- (P6 and more recent processor families only) When a serializing instruction is executed.
- When an I/O instruction is executed.
- When a LOCK operation is performed.
- (P6 and more recent processor families only) When a BINIT operation is performed.
- (Pentium III, and more recent processor families only) When using an SFENCE instruction to order stores.
- (Pentium 4 and more recent processor families only) When using an MFENCE instruction to order stores.


12.11 MEMORY TYPE RANGE REGISTERS (MTRRS)

The following section pertains only to the P6 and more recent processor families.

The memory type range registers (MTRRs) provide a mechanism for associating the memory types (see Section 12.3, “Methods of Caching Available”) with physical-address ranges in system memory. They allow the processor to optimize operations for different types of memory such as RAM, ROM, frame-buffer memory, and memory-mapped I/O devices. They also simplify system hardware design by eliminating the memory control pins used for this function on earlier IA-32 processors and the external logic needed to drive them.

The MTRR mechanism allows multiple ranges to be defined in physical memory, and it defines a set of model-specific registers (MSRs) for specifying the type of memory that is contained in each range. Table 12-8 shows the memory types that can be specified and their properties; Figure 12-4 shows the mapping of physical memory with MTRRs. See Section 12.3, “Methods of Caching Available,” for a more detailed description of each memory type.

Following a hardware reset, the P6 and more recent processor families disable all the fixed and variable MTRRs, which in effect makes all of physical memory uncacheable. Initialization software should then set the MTRRs to a specific, system-defined memory map. Typically, the BIOS (basic input/output system) software configures the MTRRs. The operating system or executive is then free to modify the memory map using the normal page-level cacheability attributes.

In a multiprocessor system using a processor in the P6 family or a more recent family, each processor MUST use the identical MTRR memory map so that software will have a consistent view of memory.

NOTE

In multiple processor systems, the operating system must maintain MTRR consistency between all the processors in the system (that is, all processors must use the same MTRR values). The P6 and more recent processor families provide no hardware support for maintaining this consistency.
12.11.1 MTRR Feature Identification

The availability of the MTRR feature is model-specific. Software can determine if MTRRs are supported on a processor by executing the CPUID instruction and reading the state of the MTRR flag (bit 12) in the feature information register (EDX).

If the MTRR flag is set (indicating that the processor implements MTRRs), additional information about MTRRs can be obtained from the 64-bit IA32_MTRRCAP MSR (named MTRRcap MSR for the P6 family processors). The IA32_MTRRCAP MSR is a read-only MSR that can be read with the RDMSR instruction. Figure 12-5 shows the contents of the IA32_MTRRCAP MSR. The functions of the flags and field in this register are as follows:

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Encoding in MTRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncacheable (UC)</td>
<td>00H</td>
</tr>
<tr>
<td>Write Combining (WC)</td>
<td>01H</td>
</tr>
<tr>
<td>Reserved*</td>
<td>02H</td>
</tr>
<tr>
<td>Reserved*</td>
<td>03H</td>
</tr>
<tr>
<td>Write-through (WT)</td>
<td>04H</td>
</tr>
<tr>
<td>Write-protected (WP)</td>
<td>05H</td>
</tr>
<tr>
<td>Writeback (WB)</td>
<td>06H</td>
</tr>
<tr>
<td>Reserved*</td>
<td>7H through FFH</td>
</tr>
</tbody>
</table>

**Table 12-8. Memory Types That Can Be Encoded in MTRRs**

* Use of these encodings results in a general-protection exception (#GP).

---

**NOTE:**

---

**Figure 12-4. Mapping Physical Memory With MTRRs**
MEMORY CACHE CONTROL

• **VCNT (variable range registers count) field, bits 0 through 7** — Indicates the number of variable ranges implemented on the processor.

• **FIX (fixed range registers supported) flag, bit 8** — Fixed range MTRRs (IA32_MTRR_FIX64K_00000 through IA32_MTRR_FIX4K_0F8000) are supported when set; no fixed range registers are supported when clear.

• **WC (write combining) flag, bit 10** — The write-combining (WC) memory type is supported when set; the WC type is not supported when clear.

• **SMRR (System-Management Range Register) flag, bit 11** — The system-management range register (SMRR) interface is supported when bit 11 is set; the SMRR interface is not supported when clear.

Bit 9 and bits 12 through 63 in the IA32_MTRRCAP MSR are reserved. If software attempts to write to the IA32_MTRRCAP MSR, a general-protection exception (#GP) is generated.

Software must read IA32_MTRRCAP VCNT field to determine the number of variable MTRRs and query other feature bits in IA32_MTRRCAP to determine additional capabilities that are supported in a processor. For example, some processors may report a value of '8' in the VCNT field, other processors may report VCNT with different values.

![Figure 12-5. IA32_MTRRCAP Register](image)

12.11.2 Setting Memory Ranges with MTRRs

The memory ranges and the types of memory specified in each range are set by three groups of registers: the IA32_MTRR_DEF_TYPE MSR, the fixed-range MTRRs, and the variable range MTRRs. These registers can be read and written to using the RDMSR and WRMSR instructions, respectively. The IA32_MTRRCAP MSR indicates the availability of these registers on the processor (see Section 12.11.1, “MTRR Feature Identification”).

12.11.2.1 IA32_MTRR_DEF_TYPE MSR

The IA32_MTRR_DEF_TYPE MSR (named MTRRdefType MSR for the P6 family processors) sets the default properties of the regions of physical memory that are not encompassed by MTRRs. The functions of the flags and field in this register are as follows:

• **Type field, bits 0 through 7** — Indicates the default memory type used for those physical memory address ranges that do not have a memory type specified for them by an MTRR (see Table 12-8 for the encoding of this field). The legal values for this field are 0, 1, 4, 5, and 6. All other values result in a general-protection exception (#GP) being generated.

Intel recommends the use of the UC (uncached) memory type for all physical memory addresses where memory does not exist. To assign the UC type to nonexistent memory locations, it can either be specified as the default type in the Type field or be explicitly assigned with the fixed and variable MTRRs.
MEMORY CACHE CONTROL

• FE (fixed MTRRs enabled) flag, bit 10 — Fixed-range MTRRs are enabled when set; fixed-range MTRRs are
disabled when clear. When the fixed-range MTRRs are enabled, they take priority over the variable-range
MTRRs when overlaps in ranges occur. If the fixed-range MTRRs are disabled, the variable-range MTRRs can
still be used and can map the range ordinarily covered by the fixed-range MTRRs.

• E (MTRRs enabled) flag, bit 11 — MTRRs are enabled when set; all MTRRs are disabled when clear, and the
UC memory type is applied to all of physical memory. When this flag is set, the FE flag can disable the fixed-
range MTRRs; when the flag is clear, the FE flag has no affect. When the E flag is set, the type specified in the
default memory type field is used for areas of memory not already mapped by either a fixed or variable MTRR.

Bits 8 and 9, and bits 12 through 63, in the IA32_MTRR_DEF_TYPE MSR are reserved; the processor generates a
general-protection exception (#GP) if software attempts to write nonzero values to them.

12.11.2.2 Fixed Range MTRRs

The fixed memory ranges are mapped with 11 fixed-range registers of 64 bits each. Each of these registers is
divided into 8-bit fields that are used to specify the memory type for each of the sub-ranges the register controls:

• Register IA32_MTRR_FIX64K_00000 — Maps the 512-KByte address range from 0H to 7FFFFH. This range
is divided into eight 64-KByte sub-ranges.

• Registers IA32_MTRR_FIX16K_80000 and IA32_MTRR_FIX16K_A0000 — Maps the two 128-KByte
address ranges from 80000H to BFFFFH. This range is divided into sixteen 16-KByte sub-ranges, 8 ranges per
register.

• Registers IA32_MTRR_FIX4K_C0000 through IA32_MTRR_FIX4K_F8000 — Maps eight 32-KByte
address ranges from C0000H to FFFFFH. This range is divided into sixty-four 4-KByte sub-ranges, 8 ranges per
register.

Table 12-9 shows the relationship between the fixed physical-address ranges and the corresponding fields of the
fixed-range MTRRs; Table 12-8 shows memory type encoding for MTRRs.

For the P6 family processors, the prefix for the fixed range MTRRs is MTRRfix.

12.11.2.3 Variable Range MTRRs

The Pentium 4, Intel Xeon, and P6 family processors permit software to specify the memory type for m variable-
size address ranges, using a pair of MTRRs for each range. The number m of ranges supported is given in bits 7:0
of the IA32_MTRRCAP MSR (see Figure 12-5 in Section 12.11.1).

The first entry in each pair (IA32_MTRR_PHYSBASEn) defines the base address and memory type for the range;
the second entry (IA32_MTRR_PHYSMASKn) contains a mask used to determine the address range. The “n” suffix
is in the range 0 through m–1 and identifies a specific register pair.

For P6 family processors, the prefixes for these variable range MTRRs are MTRRphysBase and MTRRphysMask.
Figure 12-7 shows flags and fields in these registers. The functions of these flags and fields are:

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 12-8 for the encoding of this field).

- **PhysBase field, bits 12 through (MAXPHYADDR-1)** — Specifies the base address of the address range. This 24-bit value, in the case where MAXPHYADDR is 36 bits, is extended by 12 bits at the low end to form the base address (this automatically aligns the address on a 4-KByte boundary).

- **PhysMask field, bits 12 through (MAXPHYADDR-1)** — Specifies a mask (24 bits if the maximum physical address size is 36 bits, 28 bits if the maximum physical address size is 40 bits). The mask determines the range of the region being mapped, according to the following relationships:
  
  - Address_Within_Range AND PhysMask = PhysBase AND PhysMask
  
  - This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 12.11.3, "Example Base and Mask Calculations."

  - The width of the PhysMask field depends on the maximum physical address size supported by the processor.

CPUID.80000008H reports the maximum physical address size supported by the processor. If CPUID.80000008H is not available, software may assume that the processor supports a 36-bit physical address size (then PhysMask is 24 bits wide and the upper 28 bits of IA32_MTRR_PHYSMASKn are reserved). See the Note below.

- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

### Table 12-9. Address Mapping for Fixed-Range MTRRs

<table>
<thead>
<tr>
<th>Address Range (hexadecimal)</th>
<th>MTRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 56</td>
<td>55 48 47 40 39 32</td>
</tr>
<tr>
<td>70000-7FFF</td>
<td>60000-6FFF 50000-5FFF</td>
</tr>
<tr>
<td>9C000-9FFFF</td>
<td>98000-9BFFF 94000-97FFF</td>
</tr>
<tr>
<td>BC000-BFFFF</td>
<td>B8000-BBFFF B4000-B7FFF</td>
</tr>
<tr>
<td>C7000-C7FFFF</td>
<td>C6000-C6FFF C5000-C5FFF</td>
</tr>
<tr>
<td>CF000-CFFFF</td>
<td>CE000-CFFFF CD000-CDFFF</td>
</tr>
<tr>
<td>D7000-D7FFFF</td>
<td>D5000-D5FFF D4000-D4FFF</td>
</tr>
<tr>
<td>D000-0FFF</td>
<td>DE000-DEFFF DD000-DDFFF</td>
</tr>
<tr>
<td>E7000-E7FFFF</td>
<td>E6000-E6FFF E5000-E5FFF</td>
</tr>
<tr>
<td>EF000-EFFFF</td>
<td>EE000-EEFFF ED000-EDFFF</td>
</tr>
<tr>
<td>F7000-F7FFFF</td>
<td>F6000-F6FFF F5000-F5FFF</td>
</tr>
<tr>
<td>FF000-FFFFF</td>
<td>FE000-FFFFF FD000-FDFFF</td>
</tr>
</tbody>
</table>
All other bits in the IA32_MTRR_PHYSBASE\textsubscript{n} and IA32_MTRR_PHYSMASK\textsubscript{n} registers are reserved; the processor generates a general-protection exception (\#GP) if software attempts to write to them.

Some mask values can result in ranges that are not continuous. In such ranges, the area not mapped by the mask value is set to the default memory type, unless some other MTRR specifies a type for that range. Intel does not encourage the use of “discontinuous” ranges.

**NOTE**

It is possible for software to parse the memory descriptions that BIOS provides by using the ACPI/INT15 e820 interface mechanism. This information then can be used to determine how MTRRs are initialized (for example: allowing the BIOS to define valid memory ranges and the maximum memory range supported by the platform, including the processor).

See Section 12.11.4.1, “MTRR Precedences,” for information on overlapping variable MTRR ranges.

### 12.11.2.4 System-Management Range Register Interface

If IA32_MTRRCAP[bit 11] is set, the processor supports the SMRR interface to restrict access to a specified memory address range used by system-management mode (SMM) software (see Section 32.4.2.1). If the SMRR interface is supported, SMM software is strongly encouraged to use it to protect the SMI code and data stored by SMI handler in the SMRAM region.

The system-management range registers consist of a pair of MSRs (see Figure 12-8). The IA32_SMRR_PHYSBASE MSR defines the base address for the SMRAM memory range and the memory type used to access it in SMM. The IA32_SMRR_PHYSMASK MSR contains a valid bit and a mask that determines the SMRAM address range protected by the SMRR interface. These MSRs may be written only in SMM; an attempt to write them outside of SMM causes a general-protection exception.\(^\dagger\)

Figure 12-8 shows flags and fields in these registers. The functions of these flags and fields are the following:

1. For some processor models, these MSRs can be accessed by RDMSR and WRMSR only if the SMRR interface has been enabled using a model-specific bit in the IA32_FEATURE_CONTROL MSR.
MEMORY CACHE CONTROL

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 12-8 for the encoding of this field).
- **PhysBase field, bits 12 through 31** — Specifies the base address of the address range. The address must be less than 4 GBytes and is automatically aligned on a 4-KByte boundary.
- **PhysMask field, bits 12 through 31** — Specifies a mask that determines the range of the region being mapped, according to the following relationships:
  - $\text{Address}_{\text{Within Range}} \text{ AND } \text{PhysMask} = \text{PhysBase AND PhysMask}$
  - This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 12.11.3, "Example Base and Mask Calculations."
- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

Before attempting to access these SMRR registers, software must test bit 11 in the IA32_MTRRCAP register. If SMRR is not supported, reads from or writes to registers cause general-protection exceptions.

When the valid flag in the IA32_SMRR_PHYSMASK MSR is 1, accesses to the specified address range are treated as follows:
- If the logical processor is in SMM, accesses uses the memory type in the IA32_SMRR_PHYSBASE MSR.
- If the logical processor is not in SMM, write accesses are ignored and read accesses return a fixed value for each byte. The uncacheable memory type (UC) is used in this case.

The above items apply even if the address range specified overlaps with a range specified by the MTRRs.

![IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK SMRR Pair](image)

12.11.3 Example Base and Mask Calculations

The examples in this section apply to processors that support a maximum physical address size of 36 bits. The base and mask values entered in variable-range MTRR pairs are 24-bit values that the processor extends to 36-bits.

For example, to enter a base address of 2 MBytes (200000H) in the IA32_MTRR_PHYSBASE3 register, the 12 least-significant bits are truncated and the value 000200H is entered in the PhysBase field. The same operation must be performed on mask values. For example, to map the address range from 200000H to 3FFFFFH (2 MBytes to 4 MBytes), a mask value of FFFE0000H is required. Again, the 12 least-significant bits of this mask value are truncated, so that the value entered in the PhysMask field of IA32_MTRR_PHYSMASK3 is FFE00H. This mask is chosen so that when any address in the 200000H to 3FFFFFH range is AND’d with the mask value, it will return the same value as when the base address is AND’d with the mask value (which is 200000H).
To map the address range from 400000H to 7FFFFFH (4 MBytes to 8 MBytes), a base value of 000400H is entered in the PhysBase field and a mask value of FFFC00H is entered in the PhysMask field.

**Example 12-2. Setting-Up Memory for a System**

Here is an example of setting up the MTRRs for an system. Assume that the system has the following characteristics:

- 96 MBytes of system memory is mapped as write-back memory (WB) for highest system performance.
- A custom 4-MByte I/O card is mapped to uncached memory (UC) at a base address of 64 MBytes. This restriction forces the 96 MBytes of system memory to be addressed from 0 to 64 MBytes and from 68 MBytes to 100 MBytes, leaving a 4-MByte hole for the I/O card.
- An 8-MByte graphics card is mapped to write-combining memory (WC) beginning at address A0000000H.
- The BIOS area from 15 MBytes to 16 MBytes is mapped to UC memory.

The following settings for the MTRRs will yield the proper mapping of the physical address space for this system configuration.

<table>
<thead>
<tr>
<th>MTRR PhysBase</th>
<th>MTRR PhysMask</th>
<th>Cache Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0006H</td>
<td>0000 00FF FC00 0800H</td>
<td>Caches 0-64 MByte as WB cache type.</td>
</tr>
<tr>
<td>0000 0000 0400 0006H</td>
<td>0000 00FF FE00 0800H</td>
<td>Caches 64-96 MByte as WB cache type.</td>
</tr>
<tr>
<td>0000 0000 0600 0006H</td>
<td>0000 00FF FFC0 0800H</td>
<td>Caches 96-100 MByte as WB cache type.</td>
</tr>
<tr>
<td>0000 0000 0400 0000H</td>
<td>0000 00FF FFC0 0800H</td>
<td>Caches 64-68 MByte as UC cache type.</td>
</tr>
<tr>
<td>0000 0000 00F0 0000H</td>
<td>0000 00FF FFF0 0800H</td>
<td>Caches 15-16 MByte as UC cache type.</td>
</tr>
<tr>
<td>0000 0000 A000 0001H</td>
<td>0000 00FF FF80 0800H</td>
<td>Caches A0000000-A0B00000 as WC type.</td>
</tr>
</tbody>
</table>

This MTRR setup uses the ability to overlap any two memory ranges (as long as the ranges are mapped to WB and UC memory types) to minimize the number of MTRR registers that are required to configure the memory environment. This setup also fulfills the requirement that two register pairs are left for operating system usage.

**12.11.3.1 Base and Mask Calculations for Greater-Than 36-bit Physical Address Support**

For Intel 64 and IA-32 processors that support greater than 36 bits of physical address size, software should query CPUID.80000008H to determine the maximum physical address. See the example.

**Example 12-3. Setting-Up Memory for a System with a 40-Bit Address Size**

If a processor supports 40-bits of physical address size, then the PhysMask field (in IA32_MTRR_PHYSMASKn registers) is 28 bits instead of 24 bits. For this situation, Example 12-2 should be modified as follows:

<table>
<thead>
<tr>
<th>MTRR PhysBase</th>
<th>MTRR PhysMask</th>
<th>Cache Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0006H</td>
<td>0000 00FF FC00 0800H</td>
<td>Caches 0-64 MByte as WB cache type.</td>
</tr>
</tbody>
</table>
12.11.4 Range Size and Alignment Requirement

A range that is to be mapped to a variable-range MTRR must meet the following “power of 2” size and alignment rules:

1. The minimum range size is 4 KBytes and the base address of the range must be on at least a 4-KByte boundary.

2. For ranges greater than 4 KBytes, each range must be of length $2^n$ and its base address must be aligned on a $2^n$ boundary, where $n$ is a value equal to or greater than 12. The base-address alignment value cannot be less than its length. For example, an 8-KByte range cannot be aligned on a 4-KByte boundary. It must be aligned on at least an 8-KByte boundary.

12.11.4.1 MTRR Precedences

If the MTRRs are not enabled (by setting the E flag in the IA32_MTRR_DEF_TYPE MSR), then all memory accesses are of the UC memory type. If the MTRRs are enabled, then the memory type used for a memory access is determined as follows:

1. If the physical address falls within the first 1 MByte of physical memory and fixed MTRRs are enabled, the processor uses the memory type stored for the appropriate fixed-range MTRR.

2. Otherwise, the processor attempts to match the physical address with a memory type set by the variable-range MTRRs:
   - If one variable memory range matches, the processor uses the memory type stored in the IA32_MTRR_PHYSBASEEn register for that range.
   - If two or more variable memory ranges match and the memory types are identical, then that memory type is used.
   - If two or more variable memory ranges match and one of the memory types is UC, the UC memory type used.
   - If two or more variable memory ranges match and the memory types are WT and WB, the WT memory type is used.
   - For overlaps not defined by the above rules, processor behavior is undefined.

3. If no fixed or variable memory range matches, the processor uses the default memory type.
12.11.5 MTRR Initialization

On a hardware reset, the P6 and more recent processors clear the valid flags in variable-range MTRRs and clear the E flag in the IA32_MTRR_DEF_TYPE MSR to disable all MTRRs. All other bits in the MTRRs are undefined.

Prior to initializing the MTRRs, software (normally the system BIOS) must initialize all fixed-range and variable-range MTRR register fields to 0. Software can then initialize the MTRRs according to known types of memory, including memory on devices that it auto-configures. Initialization is expected to occur prior to booting the operating system.

See Section 12.11.8, "MTRR Considerations in MP Systems," for information on initializing MTRRs in MP (multiple-processor) systems.

12.11.6 Remapping Memory Types

A system designer may re-map memory types to tune performance or because a future processor may not implement all memory types supported by the Pentium 4, Intel Xeon, and P6 family processors. The following rules support coherent memory-type re-mappings:

1. A memory type should not be mapped into another memory type that has a weaker memory ordering model. For example, the uncacheable type cannot be mapped into any other type, and the write-back, write-through, and write-protected types cannot be mapped into the weakly ordered write-combining type.

2. A memory type that does not delay writes should not be mapped into a memory type that does delay writes, because applications of such a memory type may rely on its write-through behavior. Accordingly, the write-back type cannot be mapped into the write-through type.

3. A memory type that views write data as not necessarily stored and read back by a subsequent read, such as the write-protected type, can only be mapped to another type with the same behavior (and there are no others for the Pentium 4, Intel Xeon, and P6 family processors) or to the uncacheable type.

In many specific cases, a system designer can have additional information about how a memory type is used, allowing additional mappings. For example, write-through memory with no associated write side effects can be mapped into write-back memory.

12.11.7 MTRR Maintenance Programming Interface

The operating system maintains the MTRRs after booting and sets up or changes the memory types for memory-mapped devices. The operating system should provide a driver and application programming interface (API) to access and set the MTRRs. The function calls MemTypeGet() and MemTypeSet() define this interface.

12.11.7.1 MemTypeGet() Function

The MemTypeGet() function returns the memory type of the physical memory range specified by the parameters base and size. The base address is the starting physical address and the size is the number of bytes for the memory range. The function automatically aligns the base address and size to 4-KByte boundaries. Pseudocode for the MemTypeGet() function is given in Example 12-4.
Example 12-4. MemTypeGet() Pseudocode

```c
#define MIXED_TYPES -1     /* 0 < MIXED_TYPES || MIXED_TYPES > 256 */

IF CPU_FEATURES.MTRR /* processor supports MTRRs */
    THEN
        Align BASE and SIZE to 4-KByte boundary;
        IF (BASE + SIZE) wrap physical-address space
            THEN return INVALID;
        FI;
        IF MTRRdefType.E = 0
            THEN return UC;
        FI;
        FirstType := Get4KMemType (BASE);
            /* Obtains memory type for first 4-KByte range. */
            /* See Get4KMemType (4KByteRange) in Example 12-5. */
        FOR each additional 4-KByte range specified in SIZE
            NextType := Get4KMemType (4KByteRange);
                IF NextType != FirstType
                    THEN return Mixed_Types;
                FI;
        ROF;
        return FirstType;
    ELSE return UNSUPPORTED;
FI;
```

If the processor does not support MTRRs, the function returns UNSUPPORTED. If the MTRRs are not enabled, then the UC memory type is returned. If more than one memory type corresponds to the specified range, a status of MIXED_TYPES is returned. Otherwise, the memory type defined for the range (UC, WC, WT, WB, or WP) is returned.

The pseudocode for the Get4KMemType() function in Example 12-5 obtains the memory type for a single 4-KByte range at a given physical address. The sample code determines whether an PHY_ADDRESS falls within a fixed range by comparing the address with the known fixed ranges: 0 to 7FFFFH (64-KByte regions), 80000H to BFFFFH (16-KByte regions), and C0000H to FFFFFH (4-KByte regions). If an address falls within one of these ranges, the appropriate bits within one of its MTRRs determine the memory type.

Example 12-5. Get4KMemType() Pseudocode

```c
IF IA32_MTRRCAP.FIX AND MTRRdefType.FE /* fixed registers enabled */
    THEN IF PHY_ADDRESS is within a fixed range
        return IA32_MTRR_FIX.Type;
    FI;
FOR each variable-range MTRR in IA32_MTRRCAP.VCNT
    IF IA32_MTRR_PHYSMASK.V = 0
        THEN continue;
    FI;
    IF (PHY_ADDRESS AND IA32_MTRR_PHYSMASK.Mask) =
        (IA32_MTRR_PHYSBASE.Base
            AND IA32_MTRR_PHYSMASK.Mask)
        THEN
            return IA32_MTRR_PHYSBASE.Type;
    FI;
    ROF;
return MTRRdefType.Type;
```
12.11.7.2 MemTypeSet() Function

The MemTypeSet() function in Example 12-6 sets a MTRR for the physical memory range specified by the parameters base and size to the type specified by type. The base address and size are multiples of 4 KBytes and the size is not 0.

Example 12-6. MemTypeSet Pseudocode

IF CPU_FEATURES.MTRR (* processor supports MTRRs *)
    THEN
        IF BASE and SIZE are not 4-KByte aligned or size is 0
            THEN return INVALID;
        FI;
        IF (BASE + SIZE) wrap 4-GByte address space
            THEN return INVALID;
        FI;
        IF TYPE is invalid for Pentium 4, Intel Xeon, and P6 family processors
            THEN return UNSUPPORTED;
        FI;
        IF TYPE is WC and not supported
            THEN return UNSUPPORTED;
        FI;
        IF IA32_MTRRCAP.FIX is set AND range can be mapped using a fixed-range MTRR
            THEN
                pre_mtrr_change();
                update affected MTRR;
                post_mtrr_change();
            FI;
        ELSE (* try to map using a variable MTRR pair *)
            IF IA32_MTRRCAP.VCNT = 0
                THEN return UNSUPPORTED;
            FI;
            IF conflicts with current variable ranges
                THEN return RANGE_OVERLAP;
            FI;
            IF no MTRRs available
                THEN return VAR_NOTAVAILABLE;
            FI;
            IF BASE and SIZE do not meet the power of 2 requirements for variable MTRRs
                THEN return INVALID_VAR_REQUEST;
            FI;
            pre_mtrr_change();
            Update affected MTRRs;
            post_mtrr_change();
        FI;
    FI;

pre_mtrr_change()
BEGIN
    disable interrupts;
    Save current value of CR4;
    disable and flush caches;
MEMORY CACHE CONTROL

flush TLBs;
disable MTRRs;
IF multiprocessing
    THEN maintain consistency through IPIs;
FI;
END

post_mtrr_change()
BEGIN
    flush caches and TLBs;
    enable MTRRs;
    enable caches;
    restore value of CR4;
    enable interrupts;
END

The physical address to variable range mapping algorithm in the MemTypeSet function detects conflicts with current variable range registers by cycling through them and determining whether the physical address in question matches any of the current ranges. During this scan, the algorithm can detect whether any current variable ranges overlap and can be concatenated into a single range.

The pre_mtrr_change() function disables interrupts prior to changing the MTRRs, to avoid executing code with a partially valid MTRR setup. The algorithm disables caching by setting the CD flag and clearing the NW flag in control register CR0. The caches are invalidated using the WBINVD instruction. The algorithm flushes all TLB entries either by clearing the page-global enable (PGE) flag in control register CR4 (if PGE was already set) or by updating control register CR3 (if PGE was already clear). Finally, it disables MTRRs by clearing the E flag in the IA32_MTRR_DEF_TYPE MSR.

After the memory type is updated, the post_mtrr_change() function re-enables the MTRRs and again invalidates the caches and TLBs. This second invalidation is required because of the processor's aggressive prefetch of both instructions and data. The algorithm restores interrupts and re-enables caching by setting the CD flag.

An operating system can batch multiple MTRR updates so that only a single pair of cache invalidations occur.

12.11.8 MTRR Considerations in MP Systems
In MP (multiple-processor) systems, the operating systems must maintain MTRR consistency between all the processors in the system. The Pentium 4, Intel Xeon, and P6 family processors provide no hardware support to maintain this consistency. In general, all processors must have the same MTRR values.

This requirement implies that when the operating system initializes an MP system, it must load the MTRRs of the boot processor while the E flag in register MTRRdefType is 0. The operating system then directs other processors to load their MTRRs with the same memory map. After all the processors have loaded their MTRRs, the operating system signals them to enable their MTRRs. Barrier synchronization is used to prevent further memory accesses until all processors indicate that the MTRRs are enabled. This synchronization is likely to be a shoot-down style algorithm, with shared variables and interprocessor interrupts.

Any change to the value of the MTRRs in an MP system requires the operating system to repeat the loading and enabling process to maintain consistency, using the following procedure:
1. Broadcast to all processors to execute the following code sequence.
2. Disable interrupts.
3. Wait for all processors to reach this point.
4. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.)
5. Flush all caches using the WBINVD instructions. On a processor that supports self-snooping (enumerating CPUID.01H:EDX.SS[bit 27] as 1), this step may be unnecessary. However, if there are changes for which self-snooping behavior would be problematic (e.g., changing the memory type of a cache line from WB to UC for memory-mapped I/O), execution of WBINVD would still be required.
6. If either the PGE or PCIDE flag is set in control register CR4, flush all TLBs by clearing one or both of these flags.
7. If the PGE and PCIDE flags are both clear in control register CR4, flush all TLBs by executing a MOV from control register CR3 to another register and then a MOV from that register back to CR3.
8. Disable all range registers (by clearing the E flag in register MTRRdefType). If only variable ranges are being modified, software may clear the valid bits for the affected register pairs instead.
9. Update the MTRRs.
10. Enable all range registers (by setting the E flag in register MTRRdefType). If only variable-range registers were modified and their individual valid bits were cleared, then set the valid bits for the affected ranges instead.
11. Flush all caches and all TLBs a second time. (The TLB flush is required for Pentium 4, Intel Xeon, and P6 family processors. Executing the WBINVD instruction is not needed when using Pentium 4, Intel Xeon, and P6 family processors, but it may be needed in future systems.)
12. Enter the normal cache mode to re-enable caching. (Set the CD and NW flags in control register CR0 to 0.)
13. Restore the values of the PGE and/or PCIDE flags in control register CR4, if cleared in Step 6 (above).
14. Wait for all processors to reach this point.
15. Enable interrupts.

12.11.9 Large Page Size Considerations

The MTRRs provide memory typing for a limited number of regions that have a 4 KByte granularity (the same granularity as 4-KByte pages). The memory type for a given page is cached in the processor’s TLBs. When using large pages (2 MBytes, 4 MBytes, or 1 GBytes), a single page-table entry covers multiple 4-KByte granules, each with a single memory type. Because the memory type for a large page is cached in the TLB, the processor can behave in an undefined manner if a large page is mapped to a region of memory that MTRRs have mapped with multiple memory types.

Undefined behavior can be avoided by ensuring that all MTRR memory-type ranges within a large page are of the same type. If a large page maps to a region of memory containing different MTRR-defined memory types, the PCD and PWT flags in the page-table entry should be set for the most conservative memory type for that range. For example, a large page used for memory mapped I/O and regular memory is mapped as UC memory. Alternatively, the operating system can map the region using multiple 4-KByte pages each with its own memory type.

The requirement that all 4-KByte ranges in a large page are of the same memory type implies that large pages with different memory types may suffer a performance penalty, since they must be marked with the lowest common denominator memory type. The same consideration apply to 1 GByte pages, each of which may consist of multiple 2-Mbyte ranges.

The Pentium 4, Intel Xeon, and P6 family processors provide special support for the physical memory range from 0 to 4 MBytes, which is potentially mapped by both the fixed and variable MTRRs. This support is invoked when a Pentium 4, Intel Xeon, or P6 family processor detects a large page overlapping the first 1 MByte of this memory range with a memory type that conflicts with the fixed MTRRs. Here, the processor maps the memory range as multiple 4-KByte pages within the TLB. This operation ensures correct behavior at the cost of performance. To avoid this performance penalty, operating-system software should reserve the large page option for regions of memory at addresses greater than or equal to 4 MBytes.

12.12 PAGE ATTRIBUTE TABLE (PAT)

The Page Attribute Table (PAT) extends the IA-32 architecture’s page-table format to allow memory types to be assigned to regions of physical memory based on linear address mappings. The PAT is a companion feature to the MTRRs; that is, the MTRRs allow mapping of memory types to regions of the physical address space, where the PAT allows mapping of memory types to pages within the linear address space. The MTRRs are useful for statically describing memory types for physical ranges, and are typically set up by the system BIOS. The PAT extends the functions of the PCD and PWT bits in page tables to allow all five of the memory types that can be assigned with the MTRRs (plus one additional memory type) to also be assigned dynamically to pages of the linear address space.
MEMORY CACHE CONTROL

The PAT was introduced to IA-32 architecture on the Pentium III processor. It is also available in the Pentium 4 and Intel Xeon processors.

12.12.1 Detecting Support for the PAT Feature

An operating system or executive can detect the availability of the PAT by executing the CPUID instruction with a value of 1 in the EAX register. Support for the PAT is indicated by the PAT flag (bit 16 of the values returned to EDX register). If the PAT is supported, the operating system or executive can use the IA32_PAT MSR to program the PAT. When memory types have been assigned to entries in the PAT, software can then use of the PAT-index bit (PAT) in the page-table and page-directory entries along with the PCD and PWT bits to assign memory types from the PAT to individual pages.

Note that there is no separate flag or control bit in any of the control registers that enables the PAT. The PAT is always enabled on all processors that support it, and the table lookup always occurs whenever paging is enabled, in all paging modes.

12.12.2 IA32_PAT MSR

The IA32_PAT MSR is located at MSR address 277H (see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4). Figure 12-9. shows the format of the 64-bit IA32_PAT MSR.

The IA32_PAT MSR contains eight page attribute fields: PA0 through PA7. The three low-order bits of each field are used to specify a memory type. The five high-order bits of each field are reserved, and must be set to all 0s. Each of the eight page attribute fields can contain any of the memory type encodings specified in Table 12-10.

Note that for the P6 family processors, the IA32_PAT MSR is named the PAT MSR.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>Uncacheable (UC)</td>
</tr>
<tr>
<td>01H</td>
<td>Write Combining (WC)</td>
</tr>
<tr>
<td>02H</td>
<td>Reserved*</td>
</tr>
<tr>
<td>03H</td>
<td>Reserved*</td>
</tr>
<tr>
<td>04H</td>
<td>Write Through (WT)</td>
</tr>
<tr>
<td>05H</td>
<td>Write Protected (WP)</td>
</tr>
<tr>
<td>06H</td>
<td>Write Back (WB)</td>
</tr>
<tr>
<td>07H</td>
<td>Uncached (UC-)</td>
</tr>
<tr>
<td>08H - FFH</td>
<td>Reserved*</td>
</tr>
</tbody>
</table>

Table 12-10. Memory Types That Can Be Encoded With PAT

NOTE:

* Using these encodings will result in a general-protection exception (#GP).
12.12.3 Selecting a Memory Type from the PAT

To select a memory type for a page from the PAT, a 3-bit index made up of the PAT, PCD, and PWT bits must be encoded in the page-table or page-directory entry for the page. Table 12-11 shows the possible encodings of the PAT, PCD, and PWT bits and the PAT entry selected with each encoding. The PAT bit is bit 7 in page-table entries that point to 4-KByte pages and bit 12 in paging-structure entries that point to larger pages. The PCD and PWT bits are bits 4 and 3, respectively, in paging-structure entries that point to pages of any size.

The PAT entry selected for a page is used in conjunction with the MTRR setting for the region of physical memory in which the page is mapped to determine the effective memory type for the page, as shown in Table 12-7.

<table>
<thead>
<tr>
<th>PAT</th>
<th>PCD</th>
<th>PWT</th>
<th>PAT Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PAT0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>PAT1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>PAT2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>PAT3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PAT4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>PAT5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>PAT6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>PAT7</td>
</tr>
</tbody>
</table>

12.12.4 Programming the PAT

Table 12-12 shows the default setting for each PAT entry following a power up or reset of the processor. The setting remain unchanged following a soft reset (INIT reset).

<table>
<thead>
<tr>
<th>PAT Entry</th>
<th>Memory Type Following Power-up or Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAT0</td>
<td>WB</td>
</tr>
<tr>
<td>PAT1</td>
<td>WT</td>
</tr>
<tr>
<td>PAT2</td>
<td>UC-</td>
</tr>
<tr>
<td>PAT3</td>
<td>UC</td>
</tr>
<tr>
<td>PAT4</td>
<td>WB</td>
</tr>
<tr>
<td>PAT5</td>
<td>WT</td>
</tr>
<tr>
<td>PAT6</td>
<td>UC-</td>
</tr>
<tr>
<td>PAT7</td>
<td>UC</td>
</tr>
</tbody>
</table>

The values in all the entries of the PAT can be changed by writing to the IA32_PAT MSR using the WRMSR instruction. The IA32_PAT MSR is read and write accessible (use of the RDMSR and WRMSR instructions, respectively) to software operating at a CPL of 0. Table 12-10 shows the allowable encoding of the entries in the PAT. Attempting to write an undefined memory type encoding into the PAT causes a general-protection (#GP) exception to be generated.

The operating system (OS) is responsible for ensuring that changes to a PAT entry occur in a manner that maintains the consistency of the processor caches and translation lookaside buffers (TLB). It requires the OS to invalidate all affected TLB entries (including global entries) and all entries in all paging-structure caches. It may also require flushing of the processor caches in certain situations. This can be accomplished in various ways, including the sequence below or by following the procedure specified in Section 12.11.8, “MTRR Considerations in MP Systems.” (See Section 4.10.4, “Invalidation of TLBs and Paging-Structure Caches” for additional background information.) Also note that in a multi-processor environment, it is the software's responsibility to resolve differences in conflicting memory types across logical processors that may arise from changes to the PAT (e.g., if two
logical processors map a linear address to the same physical address but have PATs that specify a different memory type for that physical address).

Example of a sequence to invalidate the processor TLBs and caches (if necessary):

1. If the PCIDE or PGE flag is set in CR4, flush TLBs by clearing one of those flags (then restore the flag via a subsequent CR4 write).

   Otherwise, flush TLBs by executing a MOV from control register CR3 to another register and then a MOV from that register back to CR3.

2. In the case that there are changes to memory-type mappings for which cache self-snoop behavior would be problematic given the existing mappings (e.g., changing a cache line’s memory type from WB to UC to be used for memory-mapped I/O), then cache flushing is also required. This can be done by executing CLFLUSH operations for all affected cache lines or by executing the WBINVD instruction (recommended only if there are a large number of affected mappings or if it is unknown which mappings are affected).

The PAT allows any memory type to be specified in the page tables, and therefore it is possible to have a single physical page mapped to two or more different linear addresses, each with different memory types. Intel does not support this practice because it may lead to undefined operations that can result in a system failure. In particular, a WC page must never be aliased to a cacheable page because WC writes may not check the processor caches.

When remapping a page that was previously mapped as a cacheable memory type to a WC page, an operating system can avoid this type of aliasing by doing the following:

1. Remove the previous mapping to a cacheable memory type in the page tables; that is, make them not present.

2. Flush the TLBs of processors that may have used the mapping, even speculatively.

3. Create a new mapping to the same physical address with a new memory type, for instance, WC.

4. Flush the caches on all processors that may have used the mapping previously. Note on processors that support self-snooping, CPUID feature flag bit 27, this step is unnecessary.

Operating systems that use a page directory as a page table (to map large pages) and enable page size extensions must carefully scrutinize the use of the PAT index bit for the 4-KByte page-table entries. The PAT index bit for a page-table entry (bit 7) corresponds to the page size bit in a page-directory entry. Therefore, the operating system can only use PAT entries PA0 through PA3 when setting the caching type for a page table that is also used as a page directory. If the operating system attempts to use PAT entries PA4 through PA7 when using this memory as a page table, it effectively sets the PS bit for the access to this memory as a page directory.

For compatibility with earlier IA-32 processors that do not support the PAT, care should be taken in selecting the encodings for entries in the PAT (see Section 12.12.5, "PAT Compatibility with Earlier IA-32 Processors").

### 12.12.5 PAT Compatibility with Earlier IA-32 Processors

For IA-32 processors that support the PAT, the IA32_PAT MSR is always active. That is, the PCD and PWT bits in page-table entries and in page-directory entries (that point to pages) are always select a memory type for a page indirectly by selecting an entry in the PAT. They never select the memory type for a page directly as they do in earlier IA-32 processors that do not implement the PAT (see Table 12-6).

To allow compatibility for code written to run on earlier IA-32 processor that do not support the PAT, the PAT mechanism has been designed to allow backward compatibility to earlier processors. This compatibility is provided through the ordering of the PAT, PCD, and PWT bits in the 3-bit PAT entry index. For processors that do not implement the PAT, the PAT index bit (bit 7 in the page-table entries and bit 12 in the page-directory entries) is reserved and set to 0. With the PAT bit reserved, only the first four entries of the PAT can be selected with the PCD and PWT bits. At power-up or reset (see Table 12-12), these first four entries are encoded to select the same memory types as the PCD and PWT bits would normally select directly in an IA-32 processor that does not implement the PAT. So, if encodings of the first four entries in the PAT are left unchanged following a power-up or reset, code written to run on earlier IA-32 processors that do not implement the PAT will run correctly on IA-32 processors that do implement the PAT.
This chapter describes those features of the Intel® MMX™ technology that must be considered when designing or enhancing an operating system to support MMX technology. It covers MMX instruction set emulation, the MMX state, aliasing of MMX registers, saving MMX state, task and context switching considerations, exception handling, and debugging.

### 13.1 EMULATION OF THE MMX INSTRUCTION SET

The IA-32 or Intel 64 architecture does not support emulation of the MMX instructions, as it does for x87 FPU instructions. The EM flag in control register CR0 (provided to invoke emulation of x87 FPU instructions) cannot be used for MMX instruction emulation. If an MMX instruction is executed when the EM flag is set, an invalid opcode exception (UD#) is generated. Table 13-1 shows the interaction of the EM, MP, and TS flags in control register CR0 when executing MMX instructions.

**Table 13-1. Action Taken By MMX Instructions for Different Combinations of EM, MP, and TS**

<table>
<thead>
<tr>
<th>CR0 Flags</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM  MP* TS</td>
<td>Execute.</td>
</tr>
<tr>
<td>0  1  0</td>
<td>#NM exception.</td>
</tr>
<tr>
<td>0  1  1</td>
<td>#UD exception.</td>
</tr>
<tr>
<td>1  1  0</td>
<td>#UD exception.</td>
</tr>
<tr>
<td>1  1  1</td>
<td>#UD exception.</td>
</tr>
</tbody>
</table>

**NOTE:**
* For processors that support the MMX instructions, the MP flag should be set.

### 13.2 THE MMX STATE AND MMX REGISTER ALIASING

The MMX state consists of eight 64-bit registers (MM0 through MM7). These registers are aliased to the low 64-bits (bits 0 through 63) of floating-point registers R0 through R7 (see Figure 13-1). Note that the MMX registers are mapped to the physical locations of the floating-point registers (R0 through R7), not to the relative locations of the registers in the floating-point register stack (ST0 through ST7). As a result, the MMX register mapping is fixed and is not affected by value in the Top Of Stack (TOS) field in the floating-point status word (bits 11 through 13).
When a value is written into an MMX register using an MMX instruction, the value also appears in the corresponding floating-point register in bits 0 through 63. Likewise, when a floating-point value written into a floating-point register by an x87 FPU, the low 64 bits of that value also appears in the corresponding MMX register.

The execution of MMX instructions have several side effects on the x87 FPU state contained in the floating-point registers, the x87 FPU tag word, and the x87 FPU status word. These side effects are as follows:

- When an MMX instruction writes a value into an MMX register, at the same time, bits 64 through 79 of the corresponding floating-point register are set to all 1s.
- When an MMX instruction (other than the EMMS instruction) is executed, each of the tag fields in the x87 FPU tag word is set to 00B (valid). (See also Section 13.2.1, “Effect of MMX, x87 FPU, FXSAVE, and FXRSTOR Instructions on the x87 FPU Tag Word.”)
- When the EMMS instruction is executed, each tag field in the x87 FPU tag word is set to 11B (empty).
- Each time an MMX instruction is executed, the TOS value is set to 000B.

Execution of MMX instructions does not affect the other bits in the x87 FPU status word (bits 0 through 10 and bits 14 and 15) or the contents of the other x87 FPU registers that comprise the x87 FPU state (the x87 FPU control word, instruction pointer, data pointer, or opcode registers).

Table 13-2 summarizes the effects of the MMX instructions on the x87 FPU state.
13.2.1 Effect of MMX, x87 FPU, FXSAVE, and FXRSTOR Instructions on the x87 FPU Tag Word

Table 13-3 summarizes the effect of MMX and x87 FPU instructions and the FXSAVE and FXRSTOR instructions on the tags in the x87 FPU tag word and the corresponding tags in an image of the tag word stored in memory.

The values in the fields of the x87 FPU tag word do not affect the contents of the MMX registers or the execution of MMX instructions. However, the MMX instructions do modify the contents of the x87 FPU tag word, as is described in Section 13.2, “The MMX State and MMX Register Aliasing.” These modifications may affect the operation of the x87 FPU when executing x87 FPU instructions, if the x87 FPU state is not initialized or restored prior to beginning x87 FPU instruction execution.

Note that the FSAVE, FXSAVE, and FSTENV instructions (which save x87 FPU state information) read the x87 FPU tag register and contents of each of the floating-point registers, determine the actual tag values for each register (empty, nonzero, zero, or special), and store the updated tag word in memory. After executing these instructions, all the tags in the x87 FPU tag word are set to empty (11B). Likewise, the EMMS instruction clears MMX state from the MMX/ floating-point registers by setting all the tags in the x87 FPU tag word to 11B.

13.3 SAVING AND RESTORING THE MMX STATE AND REGISTERS

Because the MMX registers are aliased to the x87 FPU data registers, the MMX state can be saved to memory and restored from memory as follows:
• Execute an FSAVE, FNSAVE, or FXSAVE instruction to save the MMX state to memory. (The FXSAVE instruction also saves the state of the XMM and MXCSR registers.)
• Execute an FRSTOR or FXRSTOR instruction to restore the MMX state from memory. (The FXRSTOR instruction also restores the state of the XMM and MXCSR registers.)

The save and restore methods described above are required for operating systems (see Section 13.4, "Saving MMX State on Task or Context Switches"). Applications can in some cases save and restore only the MMX registers in the following way:
• Execute eight MOVQ instructions to save the contents of the MMX0 through MMX7 registers to memory. An EMMS instruction may then (optionally) be executed to clear the MMX state in the x87 FPU.
• Execute eight MOVQ instructions to read the saved contents of MMX registers from memory into the MMX0 through MMX7 registers.

NOTE
The IA-32 architecture does not support scanning the x87 FPU tag word and then only saving valid entries.

13.4 SAVING MMX STATE ON TASK OR CONTEXT SWITCHES

When switching from one task or context to another, it is often necessary to save the MMX state. As a general rule, if the existing task switching code for an operating system includes facilities for saving the state of the x87 FPU, these facilities can also be relied upon to save the MMX state, without rewriting the task switch code. This reliance is possible because the MMX state is aliased to the x87 FPU state (see Section 13.2, "The MMX State and MMX Register Aliasing").

With the introduction of the FXSAVE and FXRSTOR instructions and of SSE/SSE2/SSE3/SSSE3 extensions, it is possible (and more efficient) to create state saving facilities in the operating system or executive that save the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3 state in one operation. Section 14.4, "Designing OS Facilities for Saving x87 FPU, SSE, AND EXTENDED States on Task or Context Switches," describes how to design such facilities. The techniques describes in this section can be adapted to saving only the MMX and x87 FPU state if needed.

13.5 EXCEPTIONS THAT CAN OCCUR WHEN EXECUTING MMX INSTRUCTIONS

MMX instructions do not generate x87 FPU floating-point exceptions, nor do they affect the processor’s status flags in the EFLAGS register or the x87 FPU status word. The following exceptions can be generated during the execution of an MMX instruction:
• Exceptions during memory accesses:
  — Stack-segment fault (#SS).
  — General protection (#GP).
  — Page fault (#PF).
  — Alignment check (#AC), if alignment checking is enabled.
• System exceptions:
  — Invalid Opcode (#UD), if the EM flag in control register CR0 is set when an MMX instruction is executed (see Section 13.1, "Emulation of the MMX Instruction Set").
  — Device not available (#NM), if an MMX instruction is executed when the TS flag in control register CR0 is set. (See Section 14.4.1, "Using the TS Flag to Control the Saving of the x87 FPU and SSE State.")
• Floating-point error (#MF). (See Section 13.5.1, "Effect of MMX Instructions on Pending x87 Floating-Point Exceptions.")
• Other exceptions can occur indirectly due to the faulty execution of the exception handlers for the above exceptions.
13.5.1 Effect of MMX Instructions on Pending x87 Floating-Point Exceptions

If an x87 FPU floating-point exception is pending and the processor encounters an MMX instruction, the processor generates a x87 FPU floating-point error (#MF) prior to executing the MMX instruction, to allow the pending exception to be handled by the x87 FPU floating-point error exception handler. While this exception handler is executing, the x87 FPU state is maintained and is visible to the handler. Upon returning from the exception handler, the MMX instruction is executed, which will alter the x87 FPU state, as described in Section 13.2, “The MMX State and MMX Register Aliasing.”

13.6 DEBUGGING MMX CODE

The debug facilities operate in the same manner when executing MMX instructions as when executing other IA-32 or Intel 64 architecture instructions.

To correctly interpret the contents of the MMX or x87 FPU registers from the FSAVE/FNSAVE or FXSAVE image in memory, a debugger needs to take account of the relationship between the x87 FPU register’s logical locations relative to TOS and the MMX register’s physical locations.

In the x87 FPU context, ST<sub>n</sub> refers to an x87 FPU register at location <i>n</i> relative to the TOS. However, the tags in the x87 FPU tag word are associated with the physical locations of the x87 FPU registers (R0 through R7). The MMX registers always refer to the physical locations of the registers (with MM0 through MM7 being mapped to R0 through R7). Figure 13-2 shows this relationship. Here, the inner circle refers to the physical location of the x87 FPU and MMX registers. The outer circle refers to the x87 FPU register’s relative location to the current TOS.

When the TOS equals 0 (case A in Figure 13-2), ST0 points to the physical location R0 on the floating-point stack. MM0 maps to ST0, MM1 maps to ST1, and so on.

When the TOS equals 2 (case B in Figure 13-2), ST0 points to the physical location R2. MM0 maps to ST6, MM1 maps to ST7, MM2 maps to ST0, and so on.

![Figure 13-2. Mapping of MMX Registers to x87 FPU Data Register Stack](image-url)
CHAPTER 14
SYSTEM PROGRAMMING FOR INSTRUCTION SET EXTENSIONS AND PROCESSOR EXTENDED STATES

This chapter describes system programming features for instruction set extensions operating on the processor state extension known as the SSE state (XMM registers, MXCSR) and for other processor extended states. Instruction set extensions operating on the SSE state include the streaming SIMD extensions (SSE), streaming SIMD extensions 2 (SSE2), streaming SIMD extensions 3 (SSE3), Supplemental SSE3 (SSSE3), and SSE4. Collectively, these are called **SSE extensions** and the corresponding instructions are called **SSE instructions**. FXSAVE/FXRSTOR instructions can be used save/restore SSE state along with FP state. See Section 10.5 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for information about FXSAVE and FXRSTOR.

Sections 14.1 through 14.4 cover system programming requirements to enable the SSE extensions, providing operating system or executive support for the SSE extensions, SIMD floating-point exceptions, exception handling, and task (context) switching. These sections primarily discuss use of FXSAVE/FXRSTOR to save/restore SSE state.

**XSAVE feature set** refers to extensions to the Intel architecture that will allow system executives to implement support for multiple processor extended states along with FP/SSE states that may be introduced over time without requiring the system executive to be modified each time a new processor state extension is introduced. XSAVE feature set provide mechanisms to enumerate the supported extended states, enable some or all of them for software use, instructions to save/restore the states and enumerate the layout of the states when saved to memory. XSAVE/XRSTOR instructions are part of the XSAVE feature set. These instructions are introduced after the introduction of FP/SSE states but can be used to manage legacy FP/SSE state along with processor extended states. See Chapter 13 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for information about XSAVE feature set.

System programming for managing processor extended states is described in sections 14.5 through 14.6. XSAVE feature set is designed to be compatible with FXSAVE/FXRSTOR and hence much of the material through sections 14.1 to 14.4 related to SSE state also applies to XSAVE feature set with the exception of enumeration and saving/restoring state.

**XSAVE Compaction** is an XSAVE feature that allows operating systems to allocate space for only the states saved to conserve memory usage. A new instruction called XSAVEC is introduced to save extended states in compacted format and XRSTOR instruction is enhanced to comprehend compacted format. System programming for managing processor extended states in compacted format is also described in section 14.5.

**Supervisor state** is an extended state that can only be accessed in ring 0. XSAVE feature set has been enhanced to manage supervisor states. Two new ring 0 instructions, XSAVES/XRSTORS, are introduced to save/restore supervisor states along with other XSAVE managed states. They are privileged instruction and only operate in compacted format. System programming for managing supervisor states is described in section 14.7.

Each XSAVE managed features may have additional feature specific system programming requirements such as exception handlers etc. Feature specific system programming requirements for XSAVE managed features are described in Section 14.8.

### 14.1 PROVIDING OPERATING SYSTEM SUPPORT FOR SSE EXTENSIONS

To use SSE extensions, the operating system or executive must provide support for initializing the processor to use these extensions, for handling SIMD floating-point exceptions, and for using FXSAVE and FXRSTOR (Section 10.5 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) to manage context. XSAVE feature set can also be used to manage SSE state along with other processor extended states as described in 14.5. This section primarily focuses on using FXSAVE/FXRSTOR to manage SSE state. Because SSE extensions share the same state, experience the same sets of non-numerical and numerical exception behavior, these guidelines that

---

1. The collection also includes PCLMULQDQ and AES instructions operating on XMM state.
Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2),” and Chapter 12, "Program-

14.1.1 Adding Support to an Operating System for SSE Extensions
The following guidelines describe functions that an operating system or executive must perform to support SSE extensions:

1. Check that the processor supports the SSE extensions.
2. Check that the processor supports the FXSAVE and FXRSTOR instructions or the XSAVE feature set.
3. Provide an initialization for the SSE states.
4. Provide support for the FXSAVE and FXRSTOR instructions or the XSAVE feature set.
5. Provide support (if necessary) in non-numeric exception handlers for exceptions generated by the SSE instruc-
tions.
6. Provide an exception handler for the SIMD floating-point exception (#XM).

The following sections describe how to implement each of these guidelines.

14.1.2 Checking for CPU Support
If the processor attempts to execute an unsupported SSE instruction, the processor generates an invalid-opcode
exception (#UD). Before an operating system or executive attempts to use SSE extensions, it should check that
support is present by confirming the following bit values returned by the CPUID instruction:

- CPUID.1:EDX.SSE[bit 25] = 1
- CPUID.1:EDX.SSE2[bit 26] = 1
- CPUID.1:ECX.SSE3[bit 0] = 1
- CPUID.1:ECX.SSSE3[bit 9] = 1
- CPUID.1:ECX.SSE4_1[bit 19] = 1
- CPUID.1:ECX.SSE4_2[bit 20] = 1

(To use POPCNT instruction, software must check CPUID.1:ECX.POPCNT[bit 23] = 1.)

Separate checks must be made to ensure that the processor supports either FXSAVE and FXRSTOR or the XSAVE
feature set. See Section 10.5 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and
Chapter 13 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, respectively.

14.1.3 Initialization of the SSE Extensions
The operating system or executive should carry out the following steps to set up SSE extensions for use by appli-
cation programs:

1. Set CR4.OSFXSR[bit 9] = 1. Setting this flag implies that the operating system provides facilities for saving
and restoring SSE state using FXSAVE and FXRSTOR instructions. These instructions may be used to save the
SSE state during task switches and when invoking the SIMD floating-point exception (#XM) handler (see
Section 14.1.5, "Providing a Handler for the SIMD Floating-Point Exception (#XM)").

If the processor does not support the FXSAVE and FXRSTOR instructions, attempting to set the OSFXSR flag
causes a general-protection exception (#GP) to be generated.
2. Set CR4.OSXMMEXCPT[bit 10] = 1. Setting this flag implies that the operating system provides a SIMD floating-point exception (#XM) handler (see Section 14.1.5, "Providing a Handler for the SIMD Floating-Point Exception (#XM)").

**NOTE**
The OSFXSR and OSXMMEXCPT bits in control register CR4 must be set by the operating system. The processor has no other way of detecting operating-system support for the FXSAVE and FXRSTOR instructions or for handling SIMD floating-point exceptions.

3. Clear CR0.EM[bit 2] = 0. This action disables emulation of the x87 FPU, which is required when executing SSE instructions (see Section 2.5, "Control Registers").

4. Set CR0.MP[bit 1] = 1. This setting is required for Intel 64 and IA-32 processors that support the SSE extensions (see Section 10.2.1, "Configuring the x87 FPU Environment").

Table 14-1 and Table 14-2 show the actions of the processor when an SSE instruction is executed, depending on the following:
- OSFXSR and OSXMMEXCPT flags in control register CR4
- SSE/SSE2/SSE3/SSSE3/SSE4 feature flags returned by CPUID
- EM, MP, and TS flags in control register CR0

### Table 14-1. Action Taken for Combinations of OSFXSR, OSXMMEXCPT, SSE, SSE2, SSE3, EM, MP, and TS

<table>
<thead>
<tr>
<th>OSFXSR</th>
<th>OSXMMEXCPT</th>
<th>SSE, SSE2, SSE3, SSE4</th>
<th>EM</th>
<th>MP</th>
<th>TS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>#UD exception.</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>#UD exception.</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>#UD exception.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Execute instruction; #UD exception if unmasked SIMD floating-point exception is detected.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Execute instruction; #XM exception if unmasked SIMD floating-point exception is detected.</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>#NM exception.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. For execution of any SSE instruction except the PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, MOVNTI, and CLFLUSH instructions.
2. Exception conditions due to CR4.OSFXSR or CR4.OSXMMEXCPT do not apply to FISTTP.
3. Only applies to DPPS, DPPD, ROUNDP, ROUNDPD, ROUNDSS, ROUNDSD.
4. For processors that support the MMX instructions, the MP flag should be set.
5. X = Don't care.
The SIMD floating-point exception mask bits (bits 7 through 12), the flush-to-zero flag (bit 15), the denormals-are-zero flag (bit 6), and the rounding control field (bits 13 and 14) in the MXCSR register should be left in their default values of 0. This permits the application to determine how these features are to be used.

14.1.4 Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE Instructions

SSE instructions can generate the same type of memory-access exceptions (such as page faults and limit violations) and other non-numeric exceptions as other Intel 64 and IA-32 architecture instructions generate.

Ordinarily, existing exception handlers can handle these and other non-numeric exceptions without code modification. However, depending on the mechanisms used in existing exception handlers, some modifications might need to be made.

The SSE extensions can generate the non-numeric exceptions listed below:

- Memory Access Exceptions:
  - Stack-segment fault (#SS).
  - General protection exception (#GP). Executing most SSE instructions with an unaligned 128-bit memory reference generates a general-protection exception. (The MOVUPS and MOVUPD instructions allow unaligned loads or stores of 128-bit memory locations, without generating a general-protection exception.) A 128-bit reference within the stack segment that is not aligned to a 16-byte boundary will also generate a general-protection exception, instead a stack-segment fault exception (#SS).
  - Page fault (#PF).
  - Alignment check (#AC). When enabled, this type of alignment check operates on operands that are less than 128-bits in size: 16-bit, 32-bit, and 64-bit. To enable the generation of alignment check exceptions, do the following:
    - Set the AM flag (bit 18 of control register CR0)
    - Set the AC flag (bit 18 of the EFLAGS register)
    - CPL must be 3

If alignment check exceptions are enabled, 16-bit, 32-bit, and 64-bit misalignment will be detected for the MOVUPD and MOVUPS instructions; detection of 128-bit misalignment is not guaranteed and may vary with implementation.

<table>
<thead>
<tr>
<th>CR4</th>
<th>CPUID</th>
<th>CR0 Flags</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSFXSR</td>
<td>SSSE3</td>
<td>SSSE4_1</td>
<td>EM</td>
</tr>
<tr>
<td>0</td>
<td>X³</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 14-2. Action Taken for Combinations of OSFXSR, SSSE3, SSE4, EM, and TS

NOTES:
1. Applies to SSE4_1 instructions except DPPS, DPPD, ROUNDPS, ROUNDPD, ROUNDSS, ROUNDSD.
2. Applies to SSE4_2 instructions except CRC32 and POPCNT.
3. X = Don’t care.
• System Exceptions:
  — Invalid-opcode exception (#UD). This exception is generated when executing SSE instructions under the following conditions:
    • SSE/SSE2/SSE3/SSSE3/SSE4_1/SSE4_2 feature flags returned by CPUID are set to 0. This condition does not affect the CLFLUSH instruction, nor POPCNT.
    • The CLFSH feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the CLFLUSH instruction.
    • The POPCNT feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the POPCNT instruction.
    • The EM flag (bit 2) in control register CR0 is set to 1, regardless of the value of TS flag (bit 3) of CR0. This condition does not affect the PAUSE, PREFETCHh, MOVNTI, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32, and POPCNT instructions.
    • The OSFXSR flag (bit 9) in control register CR4 is set to 0. This condition does not affect the PSHUFW, MOVNTQ, MOVNTI, PAUSE, PREFETCHh, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32, and POPCNT instructions.
    • Executing an instruction that causes a SIMD floating-point exception when the OSXMMEXCPT flag (bit 10) in control register CR4 is set to 0. See Section 14.4.1, “Using the TS Flag to Control the Saving of the x87 FPU and SSE State.”
  — Device not available (#NM). This exception is generated by executing a SSE instruction when the TS flag (bit 3) of CR0 is set to 1.

Other exceptions can occur during delivery of the above exceptions.

14.1.5 Providing a Handler for the SIMD Floating-Point Exception (#XM)

SSE instructions do not generate numeric exceptions on packed integer operations. They can generate the following numeric (SIMD floating-point) exceptions on packed and scalar single precision and double precision floating-point operations.
• Invalid operation (#I)
• Divide-by-zero (#Z)
• Denormal operand (#D)
• Numeric overflow (#O)
• Numeric underflow (#U)
• Inexact result (Precision) (#P)

These SIMD floating-point exceptions (with the exception of the denormal operand exception) are defined in the IEEE Standard 754 for Floating-Point Arithmetic and represent the same conditions that cause x87 FPU floating-point error exceptions (#MF) to be generated for x87 FPU instructions.

Each of these exceptions can be masked, in which case the processor returns a reasonable result to the destination operand without invoking an exception handler. However, if any of these exceptions are left unmasked, detection of the exception condition results in a SIMD floating-point exception (#XM) being generated. See Chapter 6, "Interrupt 19—SIMD Floating-Point Exception (#XM)."

To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. The section titled "SSE and SSE2 SIMD Floating-Point Exceptions” in Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (#XM), the OSXMEXCPT flag (bit 10) must be set in control register CR4.
14.1.5.1 Numeric Error flag and IGNNE#  
SSE extensions ignore the NE flag in control register CR0 (that is, they treat it as if it were always set) and the IGNNE# pin. When an unmasked SIMD floating-point exception is detected, it is always reported by generating a SIMD floating-point exception (#XM).

14.2 EMULATION OF SSE EXTENSIONS  
The Intel 64 and IA-32 architectures do not support emulation of the SSE instructions, as they do for x87 FPU instructions.  
The EM flag in control register CR0 (provided to invoke emulation of x87 FPU instructions) cannot be used to invoke emulation of SSE instructions. If an SSE instruction is executed when CR0.EM = 1, an invalid opcode exception (#UD) is generated. See Table 14-1.

14.3 SAVING AND RESTORING SSE STATE  
The SSE state consists of the state of the XMM and MXCSR registers. Intel recommends the following method for saving and restoring this state:
• Execute the FXSAVE instruction to save the state of the XMM and MXCSR registers to memory.
• Execute the FXRSTOR instruction to restore the state of the XMM and MXCSR registers from the image saved in memory earlier.

This save and restore method is required for all operating systems. XSAVE feature set can also be used to save/restore SSE state. See Section 14.5, “The XSAVE Feature Set and Processor Extended State Management,” for using the XSAVE feature set to save/restore SSE state.

In some cases, applications may choose to save only the XMM and MXCSR registers in the following manner:
• Execute MOVDQ instructions to save the contents of the XMM registers to memory.
• Execute a STMXCSR instruction to save the state of the MXCSR register to memory.

Such applications must restore the XMM and MXCSR registers as follows:
• Execute MOVDQ instructions to load the saved contents of the XMM registers from memory into the XMM registers.
• Execute a LDMXCSR instruction to restore the state of the MXCSR register from memory.

14.4 DESIGNING OS FACILITIES FOR SAVING X87 FPU, SSE, AND EXTENDED STATES ON TASK OR CONTEXT SWITCHES  
The x87 FPU and SSE state consist of the state of the x87 FPU, XMM, and MXCSR registers. The FXSAVE and FXRSTOR instructions provide a fast method for saving and restoring this state. The XSAVE feature set can also be used to save FP and SSE state along with other extended states (see Section 14.5).

Older operating systems may use FSAVE/FNSAVE and FRSTOR to save the x87 FPU state. These facilities can be extended to save and restore SSE state by substituting FXSAVE and FXRSTOR or the XSAVE feature set in place of FSAVE/FNSAVE and FRSTOR.

If task or context switching facilities are written from scratch, any of several approaches may be taken for using the FXSAVE and FXRSTOR instructions or the XSAVE feature set to save and restore x87 FPU and SSE state:
• The operating system can require applications that are intended to be run as tasks take responsibility for saving the states prior to a task suspension during a task switch and for restoring the states when the task is resumed. This approach is appropriate for cooperative multitasking operating systems, where the application has control over (or is able to determine) when a task switch is about to occur and can save state prior to the task switch.
The operating system can take the responsibility for saving the states as part of the task switch process and restoring the state of the registers when a suspended task is resumed. This approach is appropriate for preemptive multitasking operating systems, where the application cannot know when it is going to be preempted and cannot prepare in advance for task switching.

The operating system can take the responsibility for saving the states as part of the task switch process, but delay the restoring of the states until an instruction operating on the states is actually executed by the new task. See Section 14.4.1, "Using the TS Flag to Control the Saving of the x87 FPU and SSE State," for more information. This approach is called lazy restore.

The use of lazy restore mechanism in context switches is not recommended when XSAVE feature set is used to save/restore states for the following reasons.

— With XSAVE feature set, Intel processors have optimizations in place to avoid saving the state components that are in their initial configurations or when they have not been modified since they were restored last. These optimizations eliminate the need for lazy restore. See section 13.5.4 in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

— Intel processors have power optimizations when state components are in their initial configurations. Use of lazy restore retains the non-initial configuration of the last thread and is not power efficient.

— Not all extended states support lazy restore mechanisms. As such, when one or more such states are enabled it becomes very inefficient to use lazy restore as it results in two separate state restore, one in context switch for the states that does not support lazy restore and one in the #NM handler for states that support lazy restore.

### 14.4.1 Using the TS Flag to Control the Saving of the x87 FPU and SSE State

The TS flag in control register CR0 is provided to allow the operating system to delay saving/restoring the x87 FPU and SSE state until an instruction that actually accesses this state is encountered in a new task. When the TS flag is set, the processor monitors the instruction stream for x87 FPU, MMX, SSE instructions. When the processor detects one of these instructions, it raises a device-not-available exception (#NM) prior to executing the instruction. The #NM exception handler can then be used to save the x87 FPU and SSE state for the previous task (using an FXSAVE, XSAVE, or XSAVEOPT instruction) and load the x87 FPU and SSE state for the current task (using an FXRSTOR or XRSOTR instruction). If the task never encounters an x87 FPU, MMX, or SSE instruction, the device-not-available exception will not be raised and a task state will not be saved/restored unnecessarily.

**NOTE**

The CRC32 and POPCNT instructions do not operate on the x87 FPU or SSE state. They operate on the general-purpose registers and are not involved with the techniques described above.

The TS flag can be set either explicitly (by executing a MOV instruction to control register CR0) or implicitly (using the IA-32 architecture’s native task switching mechanism). When the native task switching mechanism is used, the processor automatically sets the TS flag on a task switch. After the device-not-available handler has saved the x87 FPU and SSE state, it should execute the CLTS instruction to clear the TS flag.

### 14.5 THE XSAVE FEATURE SET AND PROCESSOR EXTENDED STATE MANAGEMENT

The architecture of XSAVE feature set is described in Chapter 13 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. The XSAVE feature set includes the following:

— An extensible data layout for existing and future processor state extensions. The layout of the XSAVE area extends from the 512-byte FXSAVE/FXRSTOR layout to provide compatibility and migration path from managing the legacy FXSAVE/FXRSTOR area. The XSAVE area is described in more detail in Section 13.4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

• Control register enhancement and dedicated register for enabling each processor extended state. See Section 13.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

• Instructions to save state to and restore state from the XSAVE area. See Section 13.7 through Section 13.9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Operating systems can utilize XSAVE feature set to manage both FP/SSE state and processor extended states. CPUID leaf ODH enumerates XSAVE feature set related information. The following guidelines provide the steps an operating system needs to take to support legacy FP/SSE states and processor extended states.

1. Check that the processor supports the XSAVE feature set
2. Determine the set of XSAVE managed features that the operating system intends to enable and calculate the size of the buffer needed to save/restore the states during context switch and other flows
3. Enable use of XSAVE feature set and XSAVE managed features
4. Provide an initialization for the XSAVE managed feature state components
5. Provide (if necessary) required exception handlers for exceptions generated each of the XSAVE managed features.

14.5.1 Checking the Support for XSAVE Feature Set

Support for XSAVE Feature set is enumerated in CPUID.1.ECX.XSAVE[bit 26]. Enumeration of this bit indicates that the processor supports XSAVE/XRSTOR instructions to manage state and XSETBV/XGETBV on XCR0 to enable and get enabled states. An operating system needs to enable XSAVE feature set as described later.

Additionally, CPUID.(EAX=ODH, ECX=1).EAX enumerates additional XSAVE sub features such as optimized save, compaction, and supervisor state support. The following table summarizes XSAVE sub features. Once an operating system enables XSAVE feature set, all the sub-features enumerated are also available. There is no need to enable each additional sub feature.

Table 14-3. CPUID.(EAX=ODH, ECX=1) EAX Bit Assignment

<table>
<thead>
<tr>
<th>EAX Bit Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>If set, indicates availability of the XSAVEOPT instruction.</td>
</tr>
<tr>
<td>1</td>
<td>If set, indicates availability of the XSAVEC instruction and the corresponding compaction enhancements to the legacy XRSTOR instruction.</td>
</tr>
<tr>
<td>2</td>
<td>If set, indicates support for execution of XGETBV with ECX=1. This execution returns the state-component bitmap XINUSE. If XINUSE[i] = 0, state component i is in its initial configuration. Execution of XSETBV with ECX=1 causes a #GP.</td>
</tr>
<tr>
<td>3</td>
<td>If set, indicates support for XSAVES/XRSTORS and IA32_XSS MSR</td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

14.5.2 Determining the XSAVE Managed Feature States And The Required Buffer Size

Each XSAVE managed feature has one or more state components associated with it. An operating system policy needs to determine the XSAVE managed features to support and determine the corresponding state components to enable. When determining the XSAVE managed features to support, operating system needs to take into account the dependencies between them (e.g., AVX feature depends on SSE feature). Similarly, when a XSAVE managed feature has more than one state component, all of them need to be enabled. Each logical processor enumerates supported XSAVE state components in CPUID.(EAX=ODH, ECX=0).EDX:EAX. An operating system may enable all or a subset of the state components enumerated by the processor based on the OS policy.

The size of the memory buffer needed to save enabled XSAVE state components depends on whether the OS opts-in to use compacted format or not. Section 13.4.3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the layout of the extended region of the XSAVE area.
14.5.3 Enable the Use Of XSAVE Feature Set And XSAVE State Components

Operating systems need to enable the use of XSAVE feature set by writing to CR4.OSXSAVE[bit 18] to enable
XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using
XSAVE/XRSTOR. When the XSAVE feature set is enabled, all enumerated XSAVE sub features such as optimized
save, compaction, and supervisor state support are also enabled. Operating systems also need to enable the
XSAVE state components in XCR0 using the XSETBV instruction.

XSAVE state components can subsequently be disabled in XCR0. However, disabling state components of AVX or
AVX-512 that are not in initial configuration may incur power and performance penalty on SSE and AVX instructions
respectively. If AVX state is disabled when it is not in its initial configuration, subsequent SSE instructions may
incur a penalty. If AVX-512 state is disabled when it is not in its initial configuration, subsequent SSE and AVX
instructions may incur a penalty. It is recommended that the operating systems and VMM set AVX or AVX-512 state
components to their initial configuration before disabling them. This can be achieved by one of the two methods
below.

- Using XRSTOR: Operating system or VMM can set the state of AVX or AVX-512 state components using XRSTOR
  instruction before disabling them in XCR0.
- Using VZEROUPPER: Operating system or VMM can set AVX and AVX-512 state components to their initial
  configuration using VZEROUPPER instruction before disabling them in XCR0. Note that this will set both AVX
  and AVX-512 state components to their initial configuration. If the intent is to only disable AVX-512 state,
  operating system or VMM will need to save AVX state before executing VZEROUPPER and restore it afterwards.

14.5.4 Provide an Initialization for the XSAVE State Components

The XSAVE header of a newly allocated XSAVE area should be initialized to all zeroes before saving context. An
operating system may choose to establish beginning state-component values for a task by executing XRSTOR from
an XSAVE area that the OS has configured. If it is desired to begin state component i in its initial configuration, the
OS should clear bit i in the XSTATE_BV field in the XSAVE header; otherwise, it should set that bit and place the
desired beginning value in the appropriate location in the XSAVE area.

When a buffer is allocated for compacted size, software must ensure that the XCOMP_BV field is setup correctly
before restoring from the buffer. Bit 63 of the XCOMP_BV field indicates that the save area is in the compacted
format and the remaining bits indicate the states that have space allocated in the save area. If the buffer is first
used to save the state in compacted format, then the save instructions will setup the XCOMP_BV field appropri-
ately. If the buffer is first used to restore the state, then software must set up the XCOMP_BV field.

14.5.5 Providing the Required Exception Handlers

Instructions part of each XSAVE managed features may generate exceptions and operating system may need to
enable such exceptions and provide handlers for them. Section 14.8 describes feature specific OS requirements for
each XSAVE managed features.

14.6 INTEROPERABILITY OF THE XSAVE FEATURE SET AND FXSAVE/FXRSTOR

The FXSAVE instruction writes x87 FPU and SSE state information to a 512-byte FXSAVE save area. FXRSTOR
restores the processor’s x87 FPU and SSE states from an FXSAVE area. The XSAVE features set supports x87 FPU
and SSE states using the same layout as the FXSAVE area to provide interoperability of FXSAVE versus XSAVE, and
FXRSTOR versus XRSTOR. The XSAVE feature set allows system software to manage SSE state independent of x87
FPU states. Thus system software that had been using FXSAVE and FXRSTOR to manage x87 FPU and SSE states
can transition to using the XSAVE feature set to manage x87 FPU, SSE, and other processor extended states in a
systematic and forward-looking manner. See Section 10.5 and Chapter 13 of the Intel® 64 and IA-32 Architectures
Software Developer’s Manual, Volume 1, for more details.

System software can implement forward-looking processor extended state management using the XSAVE feature
set. In this case, system software must specify the bit vector mask in EDX:EAX appropriately when executing
XSAVE/XRSTOR instructions.
For instance, the OS can supply instructions in the XSAVE feature set with a bit vector in EDX:EAX with the two least significant bits (corresponding to x87 FPU and SSE state) equal to 0. Then, the XSAVE instruction will not write the processor’s x87 FPU and SSE state into memory. Similarly, the XRSTOR instruction executed with a value in EDX:EAX with the least two significant bit equal to 0 will not restore nor initialize the processor’s x87 FPU and SSE state.

The processor’s action as a result of executing XRSTOR is given in Section 13.8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. The instruction may be used to initialize x87 FPU or XMM registers. When the MXCSR register is updated from memory, reserved bit checking is enforced. The saving/restoring of MXCSR is bound to the SSE state, independent of the x87 FPU state. The action of XSAVE is given in Section 13.7 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

### 14.7 THE XSAVE FEATURE SET AND PROCESSOR SUPERVISOR STATE MANAGEMENT

Supervisor state is a processor state that is only accessible in ring 0. An extension to the XSAVE feature set, enumerated by CPUID.(EAX=0DH, ECX=1).EAX[bit 3] allows the management of the supervisor states using the XSAVE feature set. See Chapter 13 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for the details of the supervisor state XSAVE feature set extension. The supervisor state extension includes the following:

- CPUID enhancements to enumerate the set of supervisor states and their sizes that can be managed by the XSAVE feature set.
- The IA32_XSS MSR to enable the XSAVE feature set to manage one or more enumerated supervisor states.
- A pair of privileged save/restore instructions, XSAVES and XRSTORS, to save/restore supervisor states along with other XSAVE managed feature states.

The guidelines to enable the XSAVE feature set to manage supervisor state are very similar to the steps outlined in Section 13.6 with the differences noted below. The set of supervisor states that can be managed by the XSAVE feature set is enumerated in (EAX=0DH, ECX=1).EDX:ECX. XSAVE managed supervisor states are enabled in the IA32_XSS MSR instead of the XCR0 control register. There are semantic differences between user states enabled in XCR0 and supervisor state enabled in the IA32_XSS MSR. A supervisor state enabled in the IA32_XSS MSR:

- May be accessed via other mechanisms such as RDMSR/WRMSR even when they are not enabled in the IA32_XSS MSR. Enabling a supervisor state in the IA32_XSS MSR merely indicates that the state can be saved/restored using XSAVES/XRSTORS instructions.
- May have side effects when saving/restoring the state such as disabling/enabling the feature associated with the state. This behavior is feature specific and will be documented along with the feature description.
- May generate faults when saving/restoring the state. XSAVES/XRSTORS will follow the faulting behavior of RDMSR/WRMSR respectively if the corresponding state is also accessible using RDMSR/WRMSR.
- XRSTORS may fault when restoring the state for supervisor features that are already enabled via feature specific mechanisms. This behavior is feature specific and will be documented along with the feature description.

When a supervisor state is disabled via a feature specific mechanism, the state does not automatically get marked as INIT. Hence XSAVES/XRSTORS will continue to save/restore the state subject to available optimizations. If the software does not intend to preserve the state when it disables the feature, it should initialize it to hardware INIT value with the XRSTORS instruction so that XSAVES/XRSTORS perform optimally for that state.

### 14.8 SYSTEM PROGRAMMING FOR XSAVE MANAGED FEATURES

This section describes system programming requirement for each XSAVE managed features that are feature specific, such as exception handling.
14.8.1 Intel® Advanced Vector Extensions (Intel® AVX)

Intel AVX instructions comprises of 256-bit and 128-bit instructions that operates on 256-bit YMM registers. The XSAVE feature set allows software to save and restore the state of these registers. See Chapter 13 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

For processors that support YMM states, the YMM state exists in all operating modes. However, the available instruction interfaces to access YMM states may vary in different modes.

Operating systems must use the XSAVE feature set for YMM state management. The XSAVE feature set also provides flexible and efficient interface to manage XMM/MXCSR states and x87 FPU states in conjunction with newer processor extended states like YMM states. Operating systems may need to be aware of the following when supporting AVX.

- Saving/Restoring AVX state in non-compacted format without SSE state will also save/restore MXCSR even though MXCSR is not part of AVX state. This does not happen when compacted format is used.
- Few AVX instructions such as VZEROUPPER/VZEROALL may operate on future expansion of YMM registers.

An operating system must enable its YMM state management to support AVX and any 256-bit extensions that operate on YMM registers. Otherwise, an attempt to execute an instruction in AVX extensions (including an enhanced 128-bit SIMD instructions using VEX encoding) will cause a #UD exception.

AVX instructions may generate SIMD floating-point exceptions. An OS must enable SIMD floating-point exception support by setting CR4.OSXMMEXCPT[bit 10]=1.

14.8.2 Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

Intel AVX-512 instructions are encoded using EVEX prefix. The EVEX encoding scheme can support 512-bit, 256-bit and 128-bit instructions that operate on opmask, ZMM, YMM, and XMM registers.

For processors that support the Intel AVX-512 family of instructions, the extended processor states (ZMM and opmask registers) exist in all operating modes. However, the access to these states may vary in different modes. The processor's support for instruction extensions that employ EVEX prefix encoding is independent of the processor's support for using XSAVE feature set on those states.

Instructions requiring EVEX prefix encoding are generally supported in 64-bit, 32-bit modes, and 16-bit protected mode. They are not supported in Real mode, Virtual-8086 mode or entering into SMM mode. Note that bits MAXVL-1:256 (511:256) of ZMM register state are maintained across transitions into and out of these modes. Because the XSAVE feature set instruction can operate in all operating modes, it is possible that the processor's ZMM register state can be modified by software in any operating mode by executing XRSTOR.

Operating systems must use the XSAVE/XRSTOR/XSAVEOPT instructions for ZMM and opmask state management. An OS must enable its ZMM and opmask state management to support Intel AVX-512 Foundation instructions. Otherwise, an attempt to execute an instruction in Intel AVX-512 Foundation instructions (including a scalar 128-bit SIMD instructions using EVEX encoding) will cause a #UD exception. An operating system, which enables the AVX-512 state to support Intel AVX-512 Foundation instructions, is also sufficient to support the rest of the Intel AVX-512 family of instructions. Note that even though ZMM8-ZMM31 are not accessible in 32 bit mode, a 32 bit OS is still required to allocate memory for the entire ZMM state.

Intel AVX-512 Foundation instructions may generate SIMD floating-point exceptions. An OS must enable SIMD floating-point exception support by setting CR4.OSXMMEXCPT[bit 10]=1.
This chapter describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

15.1 ENHANCED INTEL SPEEDSTEP® TECHNOLOGY

Enhanced Intel SpeedStep® Technology was introduced in the Pentium M processor. The technology enables the management of processor power consumption via performance state transitions. These states are defined as discrete operating points associated with different voltages and frequencies.

Enhanced Intel SpeedStep Technology differs from previous generations of Intel SpeedStep® Technology in two ways:

- Centralization of the control mechanism and software interface in the processor by using model-specific registers.
- Reduced hardware overhead; this permits more frequent performance state transitions.

Previous generations of the Intel SpeedStep Technology require processors to be in a deep sleep state, holding off bus master transfers for the duration of a performance state transition. Performance state transitions under the Enhanced Intel SpeedStep Technology are discrete transitions to a new target frequency.

Support is indicated by CPUID, using ECX feature bit 07. Enhanced Intel SpeedStep Technology is enabled by setting IA32_MISC_ENABLE MSR, bit 16. On reset, bit 16 of IA32_MISC_ENABLE MSR is cleared.

15.1.1 Software Interface For Initiating Performance State Transitions

State transitions are initiated by writing a 16-bit value to the IA32_PERF_CTL register, see Figure 15-2. If a transition is already in progress, transition to a new value will subsequently take effect.

Reads of IA32_PERF_CTL determine the last targeted operating point. The current operating point can be read from IA32_PERF_STATUS. IA32_PERF_STATUS is updated dynamically.

The 16-bit encoding that defines valid operating points is model-specific. Applications and performance tools are not expected to use either IA32_PERF_CTL or IA32_PERF_STATUS and should treat both as reserved. Performance monitoring tools can access model-specific events and report the occurrences of state transitions.

15.2 P-STATE HARDWARE COORDINATION

The Advanced Configuration and Power Interface (ACPI) defines performance states (P-states) that are used to facilitate system software’s ability to manage processor power consumption. Different P-states correspond to different performance levels that are applied while the processor is actively executing instructions. Enhanced Intel SpeedStep Technology supports P-states by providing software interfaces that control the operating frequency and voltage of a processor.

With multiple processor cores residing in the same physical package, hardware dependencies may exist for a subset of logical processors on a platform. These dependencies may impose requirements that impact the coordination of P-state transitions. As a result, multi-core processors may require an OS to provide additional software support for coordinating P-state transitions for those subsets of logical processors.

ACPI firmware can choose to expose P-states as dependent and hardware-coordinated to OS power management (OSPM) policy. To support OSPMs, multi-core processors must have additional built-in support for P-state hardware coordination and feedback.

Intel 64 and IA-32 processors with dependent P-states amongst a subset of logical processors permit hardware coordination of P-states and provide a hardware-coordination feedback mechanism using IA32_MPERF MSR and
POWER AND THERMAL MANAGEMENT

IA32_APERF MSR. See Figure 15-1 for an overview of the two 64-bit MSRs and the bullets below for a detailed description.

- Use CPUID to check the P-State hardware coordination feedback capability bit. CPUID.06H.ECX[Bit 0] = 1 indicates IA32_MPERF MSR and IA32_APERF MSR are present.
- IA32_MPERF MSR (E7H) increments in proportion to a fixed frequency, which is configured when the processor is booted.
- IA32_APERF MSR (E8H) increments in proportion to actual performance, while accounting for hardware coordination of P-state and TM1/TM2; or software initiated throttling.
- The MSRs are per logical processor; they measure performance only when the targeted processor is in the C0 state.
- Only the IA32_APERF/IA32_MPERF ratio is architecturally defined; software should not attach meaning to the content of the individual of IA32_APERF or IA32_MPERF MSRs.
- When either MSR overflows, both MSRs are reset to zero and continue to increment.
- Both MSRs are full 64-bits counters. Each MSR can be written to independently. However, software should follow the guidelines illustrated in Example 15-1.

If P-states are exposed by the BIOS as hardware coordinated, software is expected to confirm processor support for P-state hardware coordination feedback and use the feedback mechanism to make P-state decisions. The OSPM is expected to either save away the current MSR values (for determination of the delta of the counter ratio at a later time) or reset both MSRs (execute WRMSR with 0 to these MSRs individually) at the start of the time window used for making the P-state decision. When not resetting the values, overflow of the MSRs can be detected by checking whether the new values read are less than the previously saved values.

Example 15-1 demonstrates steps for using the hardware feedback mechanism provided by IA32_APERF MSR and IA32_MPERF MSR to determine a target P-state.

Example 15-1. Determine Target P-state From Hardware Coordinated Feedback

DWORD PercentBusy; // Percentage of processor time not idle.
    // Measure “PercentBusy” during previous sampling window.
    // Typically, “PercentBusy” is measure over a time scale suitable for
    // power management decisions
    //
    // RDMSR of MCNT and ACNT should be performed without delay.
    // Software needs to exercise care to avoid delays between
    // the two RDMSRs (for example, interrupts).
    MCNT = RDMSR(IA32_MPERF);
    ACNT = RDMSR(IA32_APERF);

    // PercentPerformance indicates the percentage of the processor
    // that is in use. The calculation is based on the PercentBusy,
    // that is the percentage of processor time not idle and the P-state
    // hardware coordinated feedback using the ACNT/MCNT ratio.
    // Note that both values need to be calculated over the same
POWER AND THERMAL MANAGEMENT

```c
// time window.
PercentPerformance = PercentBusy * (ACNT/MCNT);

// This example does not cover the additional logic or algorithms
// necessary to coordinate multiple logical processors to a target P-state.

TargetPstate = FindPstate(PercentPerformance);

if (TargetPstate ≠ currentPstate) {
    SetPState(TargetPstate);
}

// WRMSR of MCNT and ACNT should be performed without delay.
// Software needs to exercise care to avoid delays between
// the two WRMSRs (for example, interrupts).
WRMSR(IA32_MPERF, 0);
WRMSR(IA32_APERF, 0);
```

15.3 SYSTEM SOFTWARE CONSIDERATIONS AND OPPORTUNISTIC PROCESSOR PERFORMANCE OPERATION

An Intel 64 processor may support a form of processor operation that takes advantage of design headroom to opportunistically increase performance. The Intel® Turbo Boost Technology can convert thermal headroom into higher performance across multi-threaded and single-threaded workloads. The Intel® Dynamic Acceleration Technology feature can convert thermal headroom into higher performance if only one thread is active.

15.3.1 Intel® Dynamic Acceleration Technology

The Intel Core 2 Duo processor T 7700 introduces Intel Dynamic Acceleration Technology. Intel Dynamic Acceleration Technology takes advantage of thermal design headroom and opportunistically allows a single core to operate at a higher performance level when the operating system requests increased performance.

15.3.2 System Software Interfaces for Opportunistic Processor Performance Operation

Opportunistic processor performance operation, applicable to Intel Dynamic Acceleration Technology and Intel® Turbo Boost Technology, has the following characteristics:

- A transition from a normal state of operation (e.g., Intel Dynamic Acceleration Technology/Turbo mode disengaged) to a target state is not guaranteed, but may occur opportunistically after the corresponding enable mechanism is activated, the headroom is available and certain criteria are met.
- The opportunistic processor performance operation is generally transparent to most application software.
- System software (BIOS and Operating system) must be aware of hardware support for opportunistic processor performance operation and may need to temporarily disengage opportunistic processor performance operation when it requires more predictable processor operation.
- When opportunistic processor performance operation is engaged, the OS should use hardware coordination feedback mechanisms to prevent un-intended policy effects if it is activated during inappropriate situations.

15.3.2.1 Discover Hardware Support and Enabling of Opportunistic Processor Performance Operation

If an Intel 64 processor has hardware support for opportunistic processor performance operation, the power-on default state of IA32_MISC_ENABLE[38] indicates the presence of such hardware support. For Intel 64 processors that support opportunistic processor performance operation, the default value is 1, indicating its presence. For processors that do not support opportunistic processor performance operation, the default value is 0. The power-
on default value of IA32_MISC_ENABLE[38] allows BIOS to detect the presence of hardware support of opportunistic processor performance operation. IA32_MISC_ENABLE[38] is shared across all logical processors in a physical package. It is written by BIOS during platform initiation to enable/disable opportunistic processor performance operation in conjunction of OS power management capabilities, see Section 15.3.2.2. BIOS can set IA32_MISC_ENABLE[38] with 1 to disable opportunistic processor performance operation; it must clear the default value of IA32_MISC_ENABLE[38] to 0 to enable opportunistic processor performance operation. OS and applications must use CPUID leaf 06H if it needs to detect processors that have opportunistic processor performance operation enabled. When CPUID is executed with EAX = 06H on input, Bit 1 of EAX in Leaf 06H (i.e., CPUID.06H:EAX[1]) indicates opportunistic processor performance operation, such as Intel Dynamic Acceleration Technology, has been enabled by BIOS. Opportunistic processor performance operation can be disabled by setting bit 38 of IA32_MISC_ENABLE. This mechanism is intended for BIOS only. If IA32_MISC_ENABLE[38] is set, CPUID.06H:EAX[1] will return 0.

15.3.2.2 OS Control of Opportunistic Processor Performance Operation

There may be phases of software execution in which system software cannot tolerate the non-deterministic aspects of opportunistic processor performance operation. For example, when calibrating a real-time workload to make a CPU reservation request to the OS, it may be undesirable to allow the possibility of the processor delivering increased performance that cannot be sustained after the calibration phase.

System software can temporarily disengage opportunistic processor performance operation by setting bit 32 of the IA32_PERF_CTL MSR (0199H), using a read-modify-write sequence on the MSR. The opportunistic processor performance operation can be re-engaged by clearing bit 32 in IA32_PERF_CTL MSR, using a read-modify-write sequence. The DISENGAGE bit in IA32_PERF_CTL is not reflected in bit 32 of the IA32_PERF_STATUS MSR (0198H), and it is not shared between logical processors in a physical package. In order for OS to engage Intel Dynamic Acceleration Technology/Turbo mode, the BIOS must:

- Enable opportunistic processor performance operation, as described in Section 15.3.2.1.
- Expose the operating points associated with Intel Dynamic Acceleration Technology/Turbo mode to the OS.

15.3.2.3 Required Changes to OS Power Management P-State Policy

Intel Dynamic Acceleration Technology and Intel Turbo Boost Technology can provide opportunistic performance greater than the performance level corresponding to the Processor Base frequency of the processor (see CPUID’s processor frequency information). System software can use a pair of MSRs to observe performance feedback. Software must query for the presence of IA32_APERF and IA32_MPERF (see Section 15.2). The ratio between IA32_APERF and IA32_MPERF is architecturally defined and a value greater than unity indicates performance increase occurred during the observation period due to Intel Dynamic Acceleration Technology. Without incorporating such performance feedback, the target P-state evaluation algorithm can result in a non-optimal P-state target.
There are other scenarios under which OS power management may want to disable Intel Dynamic Acceleration Technology, some of these are listed below:

- When engaging ACPI defined passive thermal management, it may be more effective to disable Intel Dynamic Acceleration Technology for the duration of passive thermal management.
- When the user has indicated a policy preference of power savings over performance, OS power management may want to disable Intel Dynamic Acceleration Technology while that policy is in effect.

15.3.3 Intel® Turbo Boost Technology

Intel Turbo Boost Technology is supported in Intel Core i7 processors and Intel Xeon processors based on Nehalem microarchitecture. It uses the same principle of leveraging thermal headroom to dynamically increase processor performance for single-threaded and multi-threaded/multi-tasking environment. The programming interface described in Section 15.3.2 also applies to Intel Turbo Boost Technology.

15.3.4 Performance and Energy Bias Hint Support

Intel 64 processors may support additional software hint to guide the hardware heuristic of power management features to favor increasing dynamic performance or conserve energy consumption.

Software can detect the processor's capability to support the performance-energy bias preference hint by examining bit 3 of ECX in CPUID leaf 6. The processor supports this capability if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H).

Software can program the lowest four bits of IA32_ENERGY_PERF_BIAS MSR with a value from 0 - 15. The values represent a sliding scale, where a value of 0 (the default reset value) corresponds to a hint preference for highest performance and a value of 15 corresponds to the maximum energy savings. A value of 7 roughly translates into a hint to balance performance with energy consumption.

The layout of IA32_ENERGY_PERF_BIAS is shown in Figure 15-3. The scope of IA32_ENERGY_PERF_BIAS is per logical processor, which means that each of the logical processors in the package can be programmed with a different value. This may be especially important in virtualization scenarios, where the performance / energy requirements of one logical processor may differ from the other. Conflicting “hints” from various logical processors at higher hierarchy level will be resolved in favor of performance over energy savings.

Software can use whatever criteria it sees fit to program the MSR with an appropriate value. However, the value only serves as a hint to the hardware and the actual impact on performance and energy savings is model specific.

15.4 HARDWARE-CONTROLLED PERFORMANCE STATES (HWP)

Intel processors may contain support for Hardware-Controlled Performance States (HWP), which autonomously selects performance states while utilizing OS supplied performance guidance hints. The Enhanced Intel SpeedStep® Technology provides a means for the OS to control and monitor discrete frequency-based operating points via the IA32_PERF_CTL and IA32_PERF_STATUS MSRs.
In contrast, HWP is an implementation of the ACPI-defined Collaborative Processor Performance Control (CPPC), which specifies that the platform enumerates a continuous, abstract unit-less, performance value scale that is not tied to a specific performance state / frequency by definition. While the enumerated scale is roughly linear in terms of a delivered integer workload performance result, the OS is required to characterize the performance value range to comprehend the delivered performance for an applied workload.

When HWP is enabled, the processor autonomously selects performance states as deemed appropriate for the applied workload and with consideration of constraining hints that are programmed by the OS. These OS-provided hints include minimum and maximum performance limits, preference towards energy efficiency or performance, and the specification of a relevant workload history observation time window. The means for the OS to override HWP's autonomous selection of performance state with a specific desired performance target is also provided, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations.

15.4.1 HWP Programming Interfaces

The programming interfaces provided by HWP include the following:

- The CPUID instruction allows software to discover the presence of HWP support in an Intel processor. Specifically, execute CPUID instruction with EAX=06H as input will return 5 bit flags covering the following aspects in bits 7 through 11 of CPUID.06H:EAX:
  - Availability of HWP baseline resource and capability, CPUID.06H:EAX[bit 7]: If this bit is set, HWP provides several new architectural MSRs: IA32_PM_ENABLE, IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS.
  - Availability of HWP Notification upon dynamic Guaranteed Performance change, CPUID.06H:EAX[bit 8]: If this bit is set, HWP provides IA32_HWP_INTERRUPT MSR to enable interrupt generation due to dynamic Performance changes and excursions.
  - Availability of HWP Activity window control, CPUID.06H:EAX[bit 9]: If this bit is set, HWP allows software to program activity window in the IA32_HWP_REQUEST MSR.
  - Availability of HWP energy/performance preference control, CPUID.06H:EAX[bit 10]: If this bit is set, HWP allows software to set an energy/performance preference hint in the IA32_HWP_REQUEST MSR.
  - Availability of HWP package level control, CPUID.06H:EAX[bit 11]: If this bit is set, HWP provides the IA32_HWP_REQUEST_PKG MSR to convey OS Power Management’s control hints for all logical processors in the physical package.

Table 15-1. Architectural and Non-Architectural MSRs Related to HWP

<table>
<thead>
<tr>
<th>Address</th>
<th>Architectural</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>770H</td>
<td>Y</td>
<td>IA32_PM_ENABLE</td>
<td>Enable/Disable HWP.</td>
</tr>
<tr>
<td>771H</td>
<td>Y</td>
<td>IA32_HWP_CAPABILITIES</td>
<td>Enumerates the HWP performance range (static and dynamic).</td>
</tr>
<tr>
<td>772H</td>
<td>Y</td>
<td>IA32_HWP_REQUEST_PKG</td>
<td>Conveys OSPM's control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for all logical processor in the physical package.</td>
</tr>
<tr>
<td>773H</td>
<td>Y</td>
<td>IA32_HWP_INTERRUPT</td>
<td>Controls HWP native interrupt generation (Guaranteed Performance changes, excursions).</td>
</tr>
<tr>
<td>774H</td>
<td>Y</td>
<td>IA32_HWP_REQUEST</td>
<td>Conveys OSPM's control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for a single logical processor.</td>
</tr>
<tr>
<td>775H</td>
<td>Y</td>
<td>IA32_HWP_PECI_REQUEST_INFO</td>
<td>Conveys embedded system controller requests to override some of the OS HWP Request settings via the PECI mechanism.</td>
</tr>
<tr>
<td>777H</td>
<td>Y</td>
<td>IA32_HWP_STATUS</td>
<td>Status bits indicating changes to Guaranteed Performance and excursions to Minimum Performance.</td>
</tr>
<tr>
<td>64EH</td>
<td>N</td>
<td>MSR_PPERF</td>
<td>Productive Performance Count.</td>
</tr>
</tbody>
</table>
Additionally, HWP may provide a non-architectural MSR, MSR_PPERF, which provides a quantitative metric to software of hardware’s view of workload scalability. This hardware’s view of workload scalability is implementation specific.

### 15.4.2 Enabling HWP

The layout of the IA32_PM_ENABLE MSR is shown in Figure 15-4. The bit fields are described below:

![Figure 15-4. IA32_PM_ENABLE MSR](image)

- **HWP_ENABLE (bit 0, R/W1Once)** — Software sets this bit to enable HWP with autonomous selection of processor P-States. When set, the processor will disregard input from the legacy performance control interface (IA32_PERF_CTL). Note this bit can only be enabled once from the default value. Once set, writes to the HWP_ENABLE bit are ignored. Only RESET will clear this bit. Default = zero (0).
- **Bits 63:1** are reserved and must be zero.

After software queries CPUID and verifies the processor’s support of HWP, system software can write 1 to IA32_PM_ENABLE.HWP_ENABLE (bit 0) to enable hardware controlled performance states. The default value of IA32_PM_ENABLE MSR at power-on is 0, i.e., HWP is disabled.

Additional MSRs associated with HWP may only be accessed after HWP is enabled, with the exception of IA32_HWP_INTERRUPT and MSR_PPERF. Accessing the IA32_HWP_INTERRUPT MSR requires only HWP is present as enumerated by CPUID but does not require enabling HWP.

IA32_PM_ENABLE is a package level MSR, i.e., writing to it from any logical processor within a package affects all logical processors within that package.

### 15.4.3 HWP Performance Range and Dynamic Capabilities

The OS reads the IA32_HWP_CAPABILITIES MSR to comprehend the limits of the HWP-managed performance range as well as the dynamic capability, which may change during processor operation. The enumerated performance range values reported by IA32_HWP_CAPABILITIES directly map to initial frequency targets (prior to workload-specific frequency optimizations of HWP). However the mapping is processor family specific.

The layout of the IA32_HWP_CAPABILITIES MSR is shown in Figure 15-5. The bit fields are described below:
• **Highest_Performance** (bits 7:0, RO) — Value for the maximum non-guaranteed performance level.

• **Guaranteed_Performance** (bits 15:8, RO) — Current value for the guaranteed performance level. This value can change dynamically as a result of internal or external constraints, e.g., thermal or power limits.

• **Most_Efficient_Performance** (bits 23:16, RO) — Current value of the most efficient performance level. This value can change dynamically as a result of workload characteristics.

• **Lowest_Performance** (bits 31:24, RO) — Value for the lowest performance level that software can program to IA32_HWP_REQUEST.

• Bits 63:32 are reserved and must be zero.

The value returned in the **Guaranteed_Performance** field is hardware’s best-effort approximation of the available performance given current operating constraints. Changes to the Guaranteed_Performance value will primarily occur due to a shift in operational mode. This includes a power or other limit applied by an external agent, e.g., RAPL (see Figure 15.10.1), or the setting of a Configurable TDP level (see model-specific controls related to Programmable TDP Limit in Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.). Notification of a change to the Guaranteed_Performance occurs via interrupt (if configured) and the IA32_HWP_Status MSR. Changes to Guaranteed_Performance are indicated when a macroscopically meaningful change in performance occurs i.e., sustained for greater than one second. Consequently, notification of a change in Guaranteed Performance will typically occur no more frequently than once per second. Rapid changes in platform configuration, e.g., docking/undocking, with corresponding changes to a Configurable TDP level could potentially cause more frequent notifications.

The value returned by the **Most_Efficient_Performance** field provides the OS with an indication of the practical lower limit for the IA32_HWP_REQUEST. The processor may not honor IA32_HWP_REQUEST.Maximum Performance settings below this value.

### 15.4.4 Managing HWP

#### 15.4.4.1 IA32_HWP_REQUEST MSR (Address: 774H Logical Processor Scope)

Typically, the operating system controls HWP operation for each logical processor via the writing of control hints / constraints to the IA32_HWP_REQUEST MSR. The layout of the IA32_HWP_REQUEST MSR is shown in Figure 15-6. The bit fields are described below Figure 15-6.

Operating systems can control HWP by writing both IA32_HWP_REQUEST and IA32_HWP_REQUEST_PKG MSRs (see Section 15.4.4.2). Five valid bits within the IA32_HWP_REQUEST MSR let the operating system flexibly select which of its five hint / constraint fields should be derived by the processor from the IA32_HWP_REQUEST MSR and which should be derived from the IA32_HWP_REQUEST_PKG MSR. These five valid bits are supported if CPUID[6].EAX[17] is set.
When the IA32_HWP_REQUEST MSR Package Control bit is set, any valid bit that is NOT set indicates to the processor to use the respective field value from the IA32_HWP_REQUEST_PKG MSR. Otherwise, the values are derived from the IA32_HWP_REQUEST MSR. The valid bits are ignored when the IA32_HWP_REQUEST MSR Package Control bit is zero.

- **Minimum_Performance (bits 7:0, RW)** — Conveys a hint to the HWP hardware. The OS programs the minimum performance hint to achieve the required quality of service (QOS) or to meet a service level agreement (SLA) as needed. Note that an excursion below the level specified is possible due to hardware constraints. The default value of this field is IA32_HWP_CAPABILITIES.Lowest_Performance.

- **Maximum_Performance (bits 15:8, RW)** — Conveys a hint to the HWP hardware. The OS programs this field to limit the maximum performance that is expected to be supplied by the HWP hardware. Excursions above the limit requested by OS are possible due to hardware coordination between the processor cores and other components in the package. The default value of this field is IA32_HWP_CAPABILITIES.Highest_Performance.

- **Desired_Performance (bits 23:16, RW)** — Conveys a hint to the HWP hardware. When set to zero, hardware autonomous selection determines the performance target. When set to a non-zero value (between the range of Lowest_Performance and Highest_Performance of IA32_HWP_CAPABILITIES) conveys an explicit performance request hint to the hardware; effectively disabling HW Autonomous selection. The Desired_Performance input is non-constraining in terms of Performance and Energy Efficiency optimizations, which are independently controlled. The default value of this field is 0.

- **Energy_Performance_Preference (bits 31:24, RW)** — Conveys a hint to the HWP hardware. The OS may write a range of values from 0 (performance preference) to 0FFH (energy efficiency preference) to influence the rate of performance increase/decrease and the result of the hardware’s energy efficiency and performance optimizations. The default value of this field is 80H. Note: If CPUID.06H:EAX[bit 10] indicates that this field is not supported, HWP uses the value of the IA32_ENERGY_PERF_BIAS MSR to determine the energy efficiency/performance preference.

- **Activity_Window (bits 41:32, RW)** — Conveys a hint to the HWP hardware specifying a moving workload history observation window for performance/frequency optimizations. If 0, the hardware will determine the appropriate window size. When writing a non-zero value to this field, this field is encoded in the format of bits 38:32 as a 7-bit mantissa and bits 41:39 as a 3-bit exponent value in powers of 10. The resultant value is in microseconds. Thus, the minimal/maximum activity window size is 1 microsecond/1270 seconds. Combined with the Energy_Performance_Preference input, Activity_Window influences the rate of performance increase.
• **Package_Control** (bit 42, RW) — When set, causes this logical processor's IA32_HWP_REQUEST control inputs to be derived from the IA32_HWP_REQUEST_PKG MSR.

• Bits 58:43 are reserved and must be zero.

• **Activity_Window_Valid** (bit 59, RW) — When set, indicates to the processor to derive the Activity Window field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

• **EPP_Valid** (bit 60, RW) — When set, indicates to the processor to derive the EPP field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

• **Desired_Valid** (bit 61, RW) — When set, indicates to the processor to derive the Desired Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

• **Maximum_Valid** (bit 62, RW) — When set, indicates to the processor to derive the Maximum Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

• **Minimum_Valid** (bit 63, RW) — When set, indicates to the processor to derive the Minimum Performance field value from the IA32_HWP_REQUEST MSR even if the package control bit is set. Otherwise, derive it from the IA32_HWP_REQUEST_PKG MSR. The default value of this field is 0.

The HWP hardware clips and resolves the field values as necessary to the valid range. Reads return the last value written not the clipped values.

Processors may support a subset of IA32_HWP_REQUEST fields as indicated by CPUID. Reads of non-supported fields will return 0. Writes to non-supported fields are ignored.

The OS may override HWP's autonomous selection of performance state with a specific performance target by setting the Desired_Performance field to a non-zero value, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations, which are influenced by the Energy Performance Preference field.

Software may disable all hardware optimizations by setting Minimum_Performance = Maximum_Performance (subject to package coordination).

Note: The processor may run below the Minimum_Performance level due to hardware constraints including: power, thermal, and package coordination constraints. The processor may also run below the Minimum_Performance level for short durations (few milliseconds) following C-state exit, and when Hardware Duty Cycling (see Section 15.5) is enabled.

When the IA32_HWP_REQUEST MSR is set to fast access mode, writes of this MSR are posted, i.e., the WRMSR instruction retires before the data reaches its destination within the processor. It may retire even before all preceding IA stores are globally visible, i.e., it is not an architecturally serializing instruction anymore (no store fence). A new CPUID bit indicates this new characteristic of the IA32_HWP_REQUEST MSR (see Section 15.4.8 for additional details).
15.4.4.2 IA32_HWP_REQUEST_PKG MSR (Address: 772H Package Scope)

The structure of the IA32_HWP_REQUEST_PKG MSR (package-level) is identical to the IA32_HWP_REQUEST MSR with the exception of the the Package Control bit field and the five valid bit fields, which do not exist in the IA32_HWP_REQUEST_PKG MSR. Field values written to this MSR apply to all logical processors within the physical package with the exception of logical processors whose IA32_HWP_REQUEST.Package Control field is clear (zero). Single P-state Control mode is only supported when IA32_HWP_REQUEST_PKG is not supported.

15.4.4.3 IA32_HWP_PECI_REQUEST_INFO MSR (Address 775H Package Scope)

When an embedded system controller is integrated in the platform, it can override some of the OS HWP Request settings via the PECI mechanism. PECI initiated settings take precedence over the relevant fields in the IA32_HWP_REQUEST MSR and in the IA32_HWP_REQUEST_PKG MSR, irrespective of the Package Control bit or the Valid Bit values described above. PECI can independently control each of: Minimum Performance, Maximum Performance and EPP fields. This MSR contains both the PECI induced values and the control bits that indicate whether the embedded controller actually set the processor to use the respective value.

PECI override is supported if CPUID[6].EAX[16] is set.
The layout of the IA32_HWP_PECI_REQUEST_INFO MSR is shown in Figure 15-8. This MSR is writable by the embedded controller but is read-only by software executing on the CPU. This MSR has Package scope. The bit fields are described below:

- **Minimum_Performance (bits 7:0, RO)** — Used by the OS to read the latest value of PECI minimum performance input.
- **Maximum_Performance (bits 15:8, RO)** — Used by the OS to read the latest value of PECI maximum performance input.
- Bits 23:16 are reserved and must be zero.
- **Energy_Performance_Preference (bits 31:24, RO)** — Used by the OS to read the latest value of PECI energy performance preference input.
- Bits 59:32 are reserved and must be zero.
- **EPP_PECI_Override (bit 60, RO)** — Indicates whether PECI if currently overriding the Energy Performance Preference input. If set(1), PECI is overriding the Energy Performance Preference input. If clear(0), OS has control over Energy Performance Preference input.
- Bit 61 is reserved and must be zero.
- **Max_PECI_Override (bit 62, RO)** — Indicates whether PECI if currently overriding the Maximum Performance input. If set(1), PECI is overriding the Maximum Performance input. If clear(0), OS has control over Maximum Performance input.
- **Min_PECI_Override (bit 63, RO)** — Indicates whether PECI if currently overriding the Minimum Performance input. If set(1), PECI is overriding the Minimum Performance input. If clear(0), OS has control over Minimum Performance input.

**HWP Request Field Hierarchical Resolution**

HWP Request field resolution is fed by three MSRs: IA32_HWP_REQUEST, IA32_HWP_REQUEST_PKG, and IA32_HWP_PECI_REQUEST_INFO. The flow that the processor goes through to resolve which field value is chosen is shown below.

For each of the two HWP Request fields; Desired and Activity Window:

- If IA32_HWP_REQUEST.PACKAGE_CONTROL = 1 and IA32_HWP_REQUEST.<field> valid bit = 0
  
  Resolved Field Value = IA32_HWP_REQUEST_PKG.<field>

  Else
  
  Resolved Field Value = IA32_HWP_REQUEST.<field>

For each of the three HWP Request fields; Min, Max, and EPP:

- If IA32_HWP_PECI_REQUEST_INFO.<field> PECI Override bit = 1
  
  Resolved Field Value = IA32_HWP_PECI_REQUEST_INFO.<field>

  Else if IA32_HWP_REQUEST.PACKAGE_CONTROL = 1 and IA32_HWP_REQUEST.<field> valid bit = 0
  
  Resolved Field Value = IA32_HWP_REQUEST_PKG.<field>

  Else
  
  Resolved Field Value = IA32_HWP_REQUEST.<field>

**15.4.4.4 IA32_HWP_CTL MSR (Address: 776H Logical Processor Scope)**

IA32_HWP_CTL[0] controls the behavior of IA32_HWP_REQUEST Package Control [bit 42]. This control bit allows the IA32_HWP_REQUEST MSR to stay in INIT mode most of the time (Control Bit is equal to its RESET value of 0) thus avoiding actual saving/restoring of the MSR contents when the OS adds it to the register set saved and restored by XSAVES/XRSTORS.

- When IA32_HWP_CTL[0] = 0:
  
  - If IA32_HWP_REQUEST[42] = 0, the processor ignores all fields of the IA32_HWP_REQUEST_PKG MSR and selects all HWP values (Min, Max, EPP, Desired, Activity Window) from the IA32_HWP_REQUEST MSR.
  
  - If IA32_HWP_REQUEST[42] = 1, the processor selects the HWP values (Min, Max, EPP, Desired, Activity Window) either from the IA32_HWP_REQUEST MSR or from the IA32_HWP_REQUEST_PKG MSR according
to the values contained in the IA32_HWP_REQUEST MSR bit fields [bits 63:59]. See Section 15.4.4.1 for additional details.

- When IA32_HWP_CTL[0] = 1, the behavior is reversed:
  - If IA32_HWP_REQUEST[42] = 1, the processor ignores all fields of the IA32_HWP_REQUEST_PKG MSR and selects all HWP values (Min, Max, EPP, Desired, Activity Window) from the IA32_HWP_REQUEST MSR.
  - If IA32_HWP_REQUEST[42] = 0, the processor selects the HWP values (Min, Max, EPP, Desired, Activity Window) either from the IA32_HWP_REQUEST MSR or from the IA32_HWP_REQUEST_PKG MSR according to the values contained in the IA32_HWP_REQUEST MSR bit fields [bits 63:59]. See Section 15.4.4.1 for additional details.

Section 15-2 summarizes the IA32_HWP_CTL MSR bit 0 control behavior.

### Table 15-2. IA32_HWP_CTL MSR Bit 0 Behavior

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread request</td>
<td>Defines which HWP Request MSR is used, whether thread level or package level. When the package MSR is used, the thread MSR valid bits define which thread MSR fields override the package (default 0).</td>
</tr>
<tr>
<td>IA32_HWP_CTL[PKG_CTL]</td>
<td>IA32_HWP_REQUEST[PKG_CTL]</td>
</tr>
<tr>
<td>IA32_HWP_REQUEST</td>
<td>HWP Request MSR Used</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This MSR is supported if CPUID[6].EAX[22] is set.

If the IA32_PM_ENABLE[HWP_ENABLE] (bit 0) is not set, access to this MSR will generate a #GP fault.

#### 15.4.5 HWP Feedback

The processor provides several types of feedback to the OS during HWP operation.

The IA32_MPERF MSR and IA32_APERF MSR mechanism (see Section 15.2) allows the OS to calculate the resultant effective frequency delivered over a time period. Energy efficiency and performance optimizations directly impact the resultant effective frequency delivered.

The layout of the IA32_HWP_STATUS MSR is shown in Figure 15-9. It provides feedback regarding changes to IA32_HWP_CAPABILITIES.Guaranteed_Performance, IA32_HWP_CAPABILITIES.Highest_Performance, excursions to IA32_HWP_CAPABILITIES.Minimum_Performance, and PECI_Override entry/exit events. The bit fields are described below:

- **Guaranteed_Performance_Change (bit 0, RWC0)** — If set (1), a change to Guaranteed_Performance has occurred. Software should query IA32_HWP_CAPABILITIES.Guaranteed_Performance value to ascertain the new Guaranteed Performance value and to assess whether to re-adjust HWP hints via IA32_HWP_REQUEST. Software must clear this bit by writing a zero (0).

- **Bit 1 is reserved and must be zero.**

- **Excursion_To_Minimum (bit 2, RWC0)** — If set (1), an excursion to Minimum_Performance of IA32_HWP_REQUEST has occurred. Software must clear this bit by writing a zero (0).

- **Highest_Change (bit 3, RWC0)** — If set (1), a change to Highest Performance has occurred. Software should query IA32_HWP_CAPABILITIES to ascertain the new Highest Performance value. Software must clear this bit by writing a zero (0). Interrupts upon Highest Performance change are supported if CPUID[6].EAX[15] is set.

- **PECI_Override_Entry (bit 4, RWC0)** — If set (1), an embedded/management controller has started a PECI override of one or more OS control hints (Min, Max, EPP) specified in IA32_HWP_REQUEST or IA32_HWP_REQUEST_PKG. Software may query IA32_HWP_PECI_REQUEST_INFO MSR to ascertain which fields are now overridden via the PECI mechanism and what their values are (see Section 15.4.4.3 for additional details).
Software must clear this bit by writing a zero (0). Interrupts upon PECI override entry are supported if CPUID[6].EAX[16] is set.

- **PECI Override Exit (bit 5, RWC0)** — If set (1), an embedded/management controller has stopped overriding one or more OS control hints (Min, Max, EPP) specified in IA32_HWP_REQUEST or IA32_HWP_REQUEST_PKG. Software may query IA32_HWP_PECI_REQUEST_INFO MSR to ascertain which fields are still overridden via the PECI mechanism and which fields are now back under software control (see Section 15.4.4.3 for additional details). Software must clear this bit by writing a zero (0). Interrupts upon PECI override exit are supported if CPUID[6].EAX[16] is set.

- Bits 63:6 are reserved and must be zero.

The status bits of IA32_HWP_STATUS must be cleared (0) by software so that a new status condition change will cause the hardware to set the bit again and issue the notification. Status bits are not set for “normal” excursions, e.g., running below Minimum Performance for short durations during C-state exit. Changes to Guaranteed_Performance, Highest_Performance, excursions to Minimum_Performance, or PECI Override entry/exit will occur no more than once per second.

The OS can determine the specific reasons for a Guaranteed_Performance change or an excursion to Minimum_Performance in IA32_HWP_REQUEST by examining the associated status and log bits reported in the IA32_THERM_STATUS MSR. The layout of the IA32_HWP_STATUS MSR that HWP uses to support software query of HWP feedback is shown in Figure 15-10. The bit fields of IA32_THERM_STATUS associated with HWP feedback are described below (Bit fields of IA32_THERM_STATUS unrelated to HWP can be found in Section 15.8.5.2).
• Bits 11:0, See Section 15.8.5.2.

• **Current Limit Status (bit 12, RO)** — If set (1), indicates an electrical current limit (e.g., Electrical Design Point/IccMax) is being exceeded and is adversely impacting energy efficiency optimizations.

• **Current Limit Log (bit 13, RWC0)** — If set (1), an electrical current limit has been exceeded that has adversely impacted energy efficiency optimizations since the last clearing of this bit or a reset. This bit is sticky, software may clear this bit by writing a zero (0).

• **Cross-domain Limit Status (bit 14, RO)** — If set (1), indicates another hardware domain (e.g., processor graphics) is currently limiting energy efficiency optimizations in the processor core domain.

• **Cross-domain Limit Log (bit 15, RWC0)** — If set (1), indicates another hardware domain (e.g., processor graphics) has limited energy efficiency optimizations in the processor core domain since the last clearing of this bit or a reset. This bit is sticky, software may clear this bit by writing a zero (0).

• Bits 63:16, See Section 15.8.5.2.

### 15.4.5.1 Non-Architectural HWP Feedback

The Productive Performance (MSR_PPERF) MSR (non-architectural) provides hardware’s view of workload scalability, which is a rough assessment of the relationship between frequency and workload performance, to software. The layout of the MSR_PPERF is shown in Figure 15-11.

• **PCNT (bits 63:0, RO)** — Similar to IA32_APERF but only counts cycles perceived by hardware as contributing to instruction execution (e.g., unhalted and unstalled cycles). This counter increments at the same rate as IA32_APERF, where the ratio of \( \frac{\Delta \text{PCNT}}{\Delta \text{ACNT}} \) is an indicator of workload scalability (0% to 100%). Note that values in this register are valid even when HWP is not enabled.
15.4.6 HWP Notifications

Processors may support interrupt-based notification of changes to HWP status as indicated by CPUID. If supported, the IA32_HWP_INTERRUPT MSR is used to enable interrupt-based notifications. Notification events, when enabled, are delivered using the existing thermal LVT entry. The layout of the IA32_HWP_INTERRUPT is shown in Figure 15-12. The bit fields are described below:

- **EN_Guaranteed_Performance_Change (bit 0, RW)** — When set (1), an HWP Interrupt will be generated whenever a change to the IA32_HWP_CAPABILITIES.Guaranteed_Performance occurs. The default value is 0 (Interrupt generation is disabled).
- **EN_Excursion_Minimum (bit 1, RW)** — When set (1), an HWP Interrupt will be generated whenever the HWP hardware is unable to meet the IA32_HWP_REQUEST.Minimum_Performance setting. The default value is 0 (Interrupt generation is disabled).
- **EN_Highest_Change (bit 2, RW)** — When set (1), an HWP Interrupt will be generated whenever a change to the IA32_HWP_CAPABILITIES.Highest_Performance occurs. The default value is 0 (interrupt generation is disabled). Interrupts upon Highest Performance change are supported if CPUID[6].EAX[15] is set.
- **EN_PECI_OVERRIDE (bit 3, RW)** — When set (1), an HWP Interrupt will be generated whenever PECI starts or stops overriding any of the three HWP fields described in Section 15.4.4.3. The default value is 0 (interrupt generation is disabled). See Section 15.4.5 and Section 15.4.4.3 for details on how the OS learns what is the current set of HWP fields that are overridden by PECI. Interrupts upon PECI override change are supported if CPUID[6].EAX[16] is set.
- Bits 63:4 are reserved and must be zero.

Figure 15-12. IA32_HWP_INTERRUPT MSR

15.4.7 Idle Logical Processor Impact on Core Frequency

Intel processors use one of two schemes for setting core frequency:

1. All cores share same frequency.
2. Each physical core is set to a frequency of its own.

In both cases the two logical processors that share a single physical core are set to the same frequency, so the processor accounts for the IA32_HWP_REQUEST MSR fields of both logical processors when defining the core frequency or the whole package frequency.

When CPUID[6].EAX[20] is set and only one logical processor of the two is active, while the other is idle (in any C1 sub-state or in a deeper sleep state), only the active logical processor’s IA32_HWP_REQUEST MSR fields are considered, i.e., the HWP Request fields of a logical processor in the C1E sub-state or in a deeper sleep state are ignored.

**Note:** when a logical processor is in **C1 state** its HWP Request fields are accounted for.
15.4.8 Fast Write of Uncore MSR (Model Specific Feature)

There are a few logical processor scope MSRs whose values need to be observed outside the logical processor. The WRMSR instruction takes over 1000 cycles to complete (retire) for those MSRs. This overhead forces operating systems to avoid writing them too often whereas in many cases it is preferable that the OS writes them quite frequently for optimal power/performance operation of the processor.

The model specific “Fast Write MSR” feature reduces this overhead by an order of magnitude to a level of 100 cycles for a selected subset of MSRs.

Note: Writes to Fast Write MSRs are posted, i.e., when the WRMSR instruction completes, the data may still be “in transit” within the processor. Software can check the status by querying the processor to ensure data is already visible outside the logical processor (see Section 15.4.8.3 for additional details). Once the data is visible outside the logical processor, software is ensured that later writes by the same logical processor to the same MSR will be visible later (will not bypass the earlier writes).

MSRs that are selected for Fast Write are specified in a special capability MSR (see Section 15.4.8.1). Architectural MSRs that existed prior to the introduction of this feature and are selected for Fast Write, thus turning from slow to fast write MSRs, will be noted as such via a new CPUID bit. New MSRs that are fast upon introduction will be documented as such without an additional CPUID bit.

Three model specific MSRs are associated with the feature itself. They enable enumerating, controlling, and monitoring it. All three are logical processor scope.

15.4.8.1 FAST_UNCORE_MSRS_CAPABILITY (Address: 0x65F, Logical Processor Scope)

Operating systems or BIOS can read the FAST_UNCORE_MSRS_CAPABILITY MSR to enumerate those MSRs that are Fast Write MSRs.

- **FAST_IA32_HWP_REQUEST MSR (bit 0, RO)** — When set (1), indicates that the IA32_HWP_REQUEST MSR is supported as a Fast Write MSR. A value of 0 indicates the IA32_HWP_REQUEST MSR is not supported as a Fast Write MSR.
- Bits 63:1 are reserved and must be zero.

Figure 15-13. FAST_UNCORE_MSRS_CAPABILITY MSR

15.4.8.2 FAST_UNCORE_MSRS_CTL (Address: 0x657, Logical Processor Scope)

Operating Systems or BIOS can use the FAST_UNCORE_MSRS_CTL MSR to opt-in or opt-out for fast write of specific MSRs that are enabled for Fast Write by the processor.

Note: Not all MSRs that are selected for this feature will necessarily have this opt-in/opt-out option. They may be supported in fast write mode only.
• **FAST_IA32_HWP_REQUEST_MSR_ENABLE (bit 0, RW)** — When set (1), enables fast access mode for the IA32_HWP_REQUEST MSR and sets the low latency, posted IA32_HWP_REQUEST MSR' CPUID[6].EAX[18]. The default value is 0. Note that this bit can only be enabled once from the default value. Once set, writes to this bit are ignored. Only RESET will clear this bit.

• Bits 63:1 are reserved and must be zero.

### 15.4.8.3 FAST_UNCORE_MSRS_STATUS (Address: 0x65E, Logical Processor Scope)

Software that executes the WRMSR instruction of a Fast Write MSR can check whether the data is already visible outside the logical processor by reading the FAST_UNCORE_MSRS_STATUS MSR. For each Fast Write MSR there is a status bit that indicates whether the data is already visible outside the logical processor or is still in “transit”.

• **FAST_IA32_HWP_REQUEST_WRITE_STATUS (bit 0, RO)** — Indicates whether the CPU is still in the middle of writing IA32_HWP_REQUEST MSR, even after the WRMSR instruction has retired. A value of 1 indicates the last write of IA32_HWP_REQUEST is still ongoing. A value of 0 indicates the last write of IA32_HWP_REQUEST is visible outside the logical processor.

• Bits 63:1 are reserved and must be zero.

### 15.4.9 Fast_IA32_Hwp_Request CPUID

IA32_HWP_REQUEST is an architectural MSR that exists in processors whose CPUID[6].EAX[7] is set (HWP Base is enabled). This MSR has logical processor scope, but after its contents are written the contents become visible outside the logical processor. When the FAST_IA32_HWP_REQUEST CPUID[6].EAX[18] bit is set, writes to the IA32_HWP_REQUEST MSR are visible outside the logical processor via the “Fast Write” feature described in Section 15.4.8.

### 15.4.10 Recommendations for OS use of HWP Controls

**Common Cases of Using HWP**

The default HWP control field values are expected to be suitable for many applications. The OS can enable autonomous HWP for these common cases by
• Setting IA32_HWP_REQUEST.Desired_Performance = 0 (hardware autonomous selection determines the performance target). Set IA32_HWP_REQUEST.Activity_Window = 0 (enable HW dynamic selection of window size).

To maximize HWP benefit for the common cases, the OS should set
• IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_CAPABILITIES.Lowest_Performance and
• IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Highest_Performance.

Setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance is functionally equivalent to using of the IA32_PERF_CTL interface and is therefore not recommended (bypassing HWP).

Calibrating HWP for Application-Specific HWP Optimization
In some applications, the OS may have Quality of Service requirements that may not be met by the default values. The OS can characterize HWP by:
• keeping IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance to prevent non-linearity in the characterization process,
• utilizing the range values enumerated from the IA32_HWP_CAPABILITIES MSR to program IA32_HWP_REQUEST while executing workloads of interest and observing the power and performance result.

The power and performance result of characterization is also influenced by the IA32_HWP_REQUEST.Energy_Performance_Preference field, which must also be characterized.

Characterization can be used to set IA32_HWP_REQUEST.Minimum_Performance to achieve the required QOS in terms of performance. If IA32_HWP_REQUEST.Minimum_Performance is set higher than IA32_HWP_CAPABILITIES.Guaranteed_Performance then notification of excursions to Minimum Performance may be continuous.

If autonomous selection does not deliver the required workload performance, the OS should assess the current delivered effective frequency and for the duration of the specific performance requirement set IA32_HWP_REQUEST.Desired_Performance ≠ 0 and adjust IA32_HWP_REQUEST.Energy_Performance_Preference as necessary to achieve the required workload performance. The MSR_PPERF.PCNT value can be used to better comprehend the potential performance result from adjustments to IA32_HWP_REQUEST.Desired_Performance. The OS should set IA32_HWP_REQUEST.Desired_Performance = 0 to re-enable autonomous selection.

Tuning for Maximum Performance or Lowest Power Consumption
Maximum performance will be delivered by setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Highest_Performance and setting IA32_HWP_REQUEST.Energy_Performance_Preference = 0 (performance preference).

Lowest power will be achieved by setting IA32_HWP_REQUEST.Minimum_Performance = IA32_HWP_REQUEST.Maximum_Performance = IA32_HWP_CAPABILITIES.Lowest_Performance and setting IA32_HWP_REQUEST.Energy_Performance_Preference = 0FFH (energy efficiency preference).

Mixing Logical Processor and Package Level HWP Field Settings
Using the IA32_HWP_REQUEST Package_Control bit and the five valid bits in that MSR, the OS can mix and match between selecting the Logical Processor scope fields and the Package level fields. For example, the OS can set all logical cores' IA32_HWP_REQUEST.Package_Control bit to ‘1’, and for those logical processors if it prefers a different EPP value than the one set in the IA32_HWP_REQUEST_PKG MSR, the OS can set the desired EPP value and the EPP valid bit. This overrides the package EPP value for only a subset of the logical processors in the package.

Additional Guidelines
Set IA32_HWP_REQUEST.Energy_Performance_Preference as appropriate for the platform's current mode of operation. For example, a mobile platforms' setting may be towards performance preference when on AC power and more towards energy efficiency when on DC power.

The use of the Running Average Power Limit (RAPL) processor capability (see section 14.7.1) is highly recommended when HWP is enabled. Use of IA32_HWP_Request.Maximum_Performance for thermal control is subject to limitations and can adversely impact the performance of other processor components, e.g., graphics.
If default values deliver undesirable performance latency in response to events, the OS should set IA32_HWP_REQUEST. Activity_Window to a low (non-zero) value and IA32_HWP_REQUEST.Energy_Performance_Preference towards performance (0) for the event duration.

Similarly, for "real-time" threads, set IA32_HWP_REQUEST.Energy_Performance_Preference towards performance (0) and IA32_HWP_REQUEST. Activity_Window to a low value, e.g., 01H, for the duration of their execution.

When executing low priority work that may otherwise cause the hardware to deliver high performance, set IA32_HWP_REQUEST. Activity_Window to a longer value and reduce the IA32_HWP_Request.Maximum_Performance value as appropriate to control energy efficiency. Adjustments to IA32_HWP_REQUEST.Energy_Performance_Preference may also be necessary.

15.5 HARDWARE DUTY CYCLING (HDC)

Intel processors may contain support for Hardware Duty Cycling (HDC), which enables the processor to autonomously force its components inside the physical package into idle state. For example, the processor may selectively force only the processor cores into an idle state.

HDC is disabled by default on processors that support it. System software can dynamically enable or disable HDC to force one or more components into an idle state or wake up those components previously forced into an idle state. Forced Idling (and waking up) of multiple components in a physical package can be done with one WRMSR to a packaged-scope MSR from any logical processor within the same package.

HDC does not delay events such as timer expiration, but it may affect the latency of short (less than 1 msec) software threads, e.g., if a thread is forced to idle state just before completion and entering a “natural idle”.

HDC forced idle operation can be thought of as operating at a lower effective frequency. The effective average frequency computed by software will include the impact of HDC forced idle.

The primary use of HDC is enable system software to manage low active workloads to increase the package level C6 residency. Additionally, HDC can lower the effective average frequency in case of power or thermal limitation.

When HDC forces a logical processor, a processor core or a physical package to enter an idle state, its C-State is set to C3 or deeper. The deep “C-states” referred to in this section are processor-specific C-states.

15.5.1 Hardware Duty Cycling Programming Interfaces

The programming interfaces provided by HDC include the following:

- The CPUID instruction allows software to discover the presence of HDC support in an Intel processor. Specifically, execute CPUID instruction with EAX=06H as input, bit 13 of EAX indicates the processor’s support of the following aspects of HDC.
  - Availability of HDC baseline resource, CPUID.06H:EAX[bit 13]: If this bit is set, HDC provides the following architectural MSRs: IA32_PKG_HDC_CTL, IA32_PM_CTL1, and the IA32_THREAD_STALL MSRs.

- Additionally, HDC may provide several non-architectural MSR.
15.5.2 Package level Enabling HDC

The layout of the IA32_PKG_HDC_CTL MSR is shown in Figure 15-16. IA32_PKG_HDC_CTL is a writable MSR from any logical processor in a package. The bit fields are described below:

- **HDC_PKG_Enable (bit 0, R/W)** — Software sets this bit to enable HDC operation by allowing the processor to force to idle all “HDC-allowed” (see Figure 15.5.3) logical processors in the package. Clearing this bit disables HDC operation in the package by waking up all the processor cores that were forced into idle by a previous ‘0’-to-’1’ transition in IA32_PKG_HDC_CTL.HDC_PKG_Enable. This bit is writable only if CPUID.06H:EAX[bit 13] = 1. Default = zero (0).
- **Bits 63:1 are reserved and must be zero.**

After processor support is determined via CPUID, system software can enable HDC operation by setting IA32_PKG_HDC_CTL.HDC_PKG_Enable to 1. At reset, IA32_PKG_HDC_CTL.HDC_PKG_Enable is cleared to 0. A '0'-to-'1' transition in HDC_PKG_Enable allows the processor to force to idle all HDC-allowed (indicated by the non-zero state of IA32_PM_CTL1[bit 0]) logical processors in the package. A '1'-to-'0' transition wakes up those HDC force-idled logical processors.

Software can enable or disable HDC using this package level control multiple times from any logical processor in the package. Note the latency of writing a value to the package-visible IA32_PKG_HDC_CTL.HDC_PKG_Enable is longer than the latency of a WRMSR operation to a Logical Processor MSR (as opposed to package level MSR) such as: IA32_PM_CTL1 (described in Section 15.5.3). Propagation of the change in IA32_PKG_HDC_CTL.HDC_PKG_Enable and reaching all HDC idled logical processor to be woken up may take on the order of core C6 exit latency.

### Table 15-3. Architectural and non-Architecture MSRs Related to HDC

<table>
<thead>
<tr>
<th>Address</th>
<th>Architectural</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB0H</td>
<td>Y</td>
<td>IA32_PKG_HDC_CTL</td>
<td>Package Enable/Disable HDC.</td>
</tr>
<tr>
<td>DB1H</td>
<td>Y</td>
<td>IA32_PM_CTL1</td>
<td>Per-logical-processor select control to allow/block HDC forced idling.</td>
</tr>
<tr>
<td>DB2H</td>
<td>Y</td>
<td>IA32_THREADSTALL</td>
<td>Accumulate stalled cycles on this logical processor due to HDC forced idling.</td>
</tr>
<tr>
<td>653H</td>
<td>N</td>
<td>MSR_CORE_HDC_RESIDENCY</td>
<td>Core level stalled cycle counter due to HDC forced idling on one or more logical processor.</td>
</tr>
<tr>
<td>655H</td>
<td>N</td>
<td>MSR_PKG_HDC_SHALLOW_RESIDENCY</td>
<td>Accumulate the cycles the package was in C2^1 state and at least one logical processor was in forced idle</td>
</tr>
<tr>
<td>656H</td>
<td>N</td>
<td>MSR_PKG_HDC_DEEP_RESIDENCY</td>
<td>Accumulate the cycles the package was in the software specified Cx^1 state and at least one logical processor was in forced idle. Cx is specified in MSR_PKG_HDC_CONFIG_CTL.</td>
</tr>
<tr>
<td>652H</td>
<td>N</td>
<td>MSR_PKG_HDC_CONFIG_CTL</td>
<td>HDC configuration controls</td>
</tr>
</tbody>
</table>

### NOTES:

1. The package “C-states” referred to in this section are processor-specific C-states.

![Figure 15-16. IA32_PKG_HDC_CTL MSR](image-url)
15.5.3 Logical-Processor Level HDC Control

The layout of the IA32_PM_CTL1 MSR is shown in Figure 15-17. Each logical processor in a package has its own IA32_PM_CTL1 MSR. The bit fields are described below:

- **HDC_Allow_Block (bit 0, R/W)** — Software sets this bit to allow this logical processors to honor the package-level IA32_PKG_HDC_CTL.HDC_PKG_Enable control. Clearing this bit prevents this logical processor from using the HDC. This bit is writable only if CPUID.06H:EAX[bit 13] = 1. Default = one (1).

- Bits 63:1 are reserved and must be zero.

Fine-grain OS control of HDC operation at the granularity of per-logical-processor is provided by IA32_PM_CTL1. At RESET, all logical processors are allowed to participate in HDC operation such that OS can manage HDC using the package-level IA32_PKG_HDC_CTL.

Writes to IA32_PM_CTL1 complete with the latency that is typical to WRMSR to a Logical Processor level MSR. When the OS chooses to manage HDC operation at per-logical-processor granularity, it can write to IA32_PM_CTL1 on one or more logical processors as desired. Each write to IA32_PM_CTL1 must be done by code that executes on the logical processor targeted to be allowed into or blocked from HDC operation.

Blocking one logical processor for HDC operation may have package level impact. For example, the processor may decide to stop duty cycling of all other Logical Processors as well.

The propagation of IA32_PKG_HDC_CTL.HDC_PKG_Enable in a package takes longer than a WRMSR to IA32_PM_CTL1. The last completed write to IA32_PM_CTL1 on a logical processor will be honored when a ‘0’-to-’1’ transition of IA32_PKG_HDC_CTL.HDC_PKG_Enable arrives to a logical processor.

15.5.4 HDC Residency Counters

There is a collection of counters available for software to track various residency metrics related to HDC operation. In general, HDC residency time is defined as the time in HDC forced idle state at the granularity of per-logical-processor, per-core, or package. At the granularity of per-core/package-level HDC residency, at least one of the logical processor in a core/package must be in the HDC forced idle state.

15.5.4.1 IA32_THREAD_STALL

Software can track per-logical-processor HDC residency using the architectural MSR IA32_THREAD_STALL. The layout of the IA32_THREAD_STALL MSR is shown in Figure 15-18. Each logical processor in a package has its own IA32_THREAD_STALL MSR. The bit fields are described below:
- **Stall_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after the logical processor exits from the forced idled C-state. At each update, the number of cycles that the logical processor was stalled due to forced-idle will be added to the counter. This counter is available only if CPUID.06H:EAX[bit 13] = 1. Default = zero (0).

A value of zero in IA32 THREAD_STALL indicates either HDC is not supported or the logical processor never serviced any forced HDC idle. A non-zero value in IA32 THREAD_STALL indicates the HDC forced-idle residency times of the logical processor. It also indicates the forced-idle cycles due to HDC that could appear as C0 time to traditional OS accounting mechanisms (e.g., time-stamping OS idle/exit events).

Software can read IA32 THREAD_STALL irrespective of the state of IA32_PKG_HDC_CTL and IA32_PM_CTL1, as long as CPUID.06H:EAX[bit 13] = 1.

### 15.5.4.2 Non-Architectural HDC Residency Counters

Processors that support HDC operation may provide the following model-specific HDC residency counters.

**MSR_CORE_HDC_RESIDENCY**

Software can track per-core HDC residency using the counter MSR_CORE_HDC_RESIDENCY. This counter increments when the core is in C3 state or deeper (all logical processors in this core are idle due to either HDC or other mechanisms) and at least one of the logical processors is in HDC forced idle state. The layout of the MSR_CORE_HDC_RESIDENCY is shown in Figure 15-19. Each processor core in a package has its own MSR_CORE_HDC_RESIDENCY MSR. The bit fields are described below:

- **Core_Cx_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after core C-state exit from a forced idled C-state. At each update, the increment counts cycles when the core is in a Cx state (all its logical processor are idle) and at least one logical processor in this core was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR will cause a #GP fault. Default = zero (0).

A value of zero in MSR_CORE_HDC_RESIDENCY indicates either HDC is not supported or this processor core never serviced any forced HDC idle.

**MSR_PKG_HDC_SHALLOW_RESIDENCY**

The counter MSR_PKG_HDC_SHALLOW_RESIDENCY allows software to track HDC residency time when the package is in C2 state, all processor cores in the package are not active and at least one logical processor was forced into idle state due to HDC. The layout of the MSR_PKG_HDC_SHALLOW_RESIDENCY is shown in Figure 15-20. There is one MSR_PKG_HDC_SHALLOW_RESIDENCY per package. The bit fields are described below:
• **Pkg_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. Package shallow residency may be implementation specific. In the initial implementation, the threshold is package C2-state. The count is updated only after package C2-state exit from a forced idled C-state. At each update, the increment counts cycles when the package is in C2 state and at least one processor core in this package was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

A value of zero in MSR_PKG_HDC_SHALLOW_RESIDENCY indicates either HDC is not supported or this processor package never serviced any forced HDC idle.

**MSR_PKG_HDC_DEEP_RESIDENCY**

The counter MSR_PKG_HDC_DEEP_RESIDENCY allows software to track HDC residency time when the package is in a software-specified package Cx state, all processor cores in the package are not active and at least one logical processor was forced into idle state due to HDC. Selection of a specific package Cx state can be configured using MSR_PKG_HDC_CONFIG. The layout of the MSR_PKG_HDC_DEEP_RESIDENCY is shown in Figure 15-21. There is one MSR_PKG_HDC_DEEP_RESIDENCY per package. The bit fields are described below:

- **Pkg_Cx_Duty_Cycle_Cnt (bits 63:0, R/O)** — Stores accumulated HDC forced-idle cycle count of this processor core since last RESET. This counter increments at the same rate of the TSC. The count is updated only after package C-state exit from a forced idle state. At each update, the increment counts cycles when the package is in the software-configured Cx state and at least one processor core in this package was forced into idle state due to HDC. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

A value of zero in MSR_PKG_HDC_SHALLOW_RESIDENCY indicates either HDC is not supported or this processor package never serviced any forced HDC idle.

**MSR_PKG_HDC.CONFIG**

MSR_PKG_HDC_CONFIG allows software to configure the package Cx state that the counter MSR_PKG_HDC_DEEP_RESIDENCY monitors. The layout of the MSR_PKG_HDC_CONFIG is shown in Figure 15-22. There is one MSR_PKG_HDC_CONFIG per package. The bit fields are described below:

- **Pkg_Cx_Monitor (bits 2:0, R/W)** — Selects which package C-state the MSR_HDC_DEEP_RESIDENCY counter will monitor. The encoding of the HDC_Cx_Monitor field are: 0: no-counting; 1: count package C2 only, 2: count package C3 and deeper; 3: count package C6 and deeper; 4: count package C7 and deeper; other encodings are reserved. If CPUID.06H:EAX[bit 13] = 0, attempt to access this MSR may cause a #GP fault. Default = zero (0).

- Bits 63:3 are reserved and must be zero.
15.5.5 MPERF and APERF Counters Under HDC

HDC operation can be thought of as an average effective frequency drop due to all or some of the Logical Processors enter an idle state period.

By default, the IA32_MPERF counter counts during forced idle periods as if the logical processor was active. The IA32_APERF counter does not count during forced idle state. This counting convention allows the OS to compute the average effective frequency of the Logical Processor between the last MWAIT exit and the next MWAIT entry (OS visible C0) by \( \frac{\Delta \text{ACNT}}{\Delta \text{MCNT}} \times \text{TSC Frequency} \).

15.6 HARDWARE FEEDBACK INTERFACE AND INTEL® THREAD DIRECTOR

Intel processors that enumerate CPUID.06H.0H:EAX.HW_FEEDBACK[bit 19] as 1 support Hardware Feedback Interface (HFI). Hardware provides guidance to the Operating System (OS) scheduler to perform optimal workload scheduling through a hardware feedback interface structure in memory. Details on this table structure are described in Section 15.6.1.

Intel processors that enumerate CPUID.06H.0H:EAX[bit 23] as 1 support Intel® Thread Director. Hardware provides guidance to the Operating System (OS) scheduler to perform optimal workload scheduling through a memory resident table and software thread specific index (Class ID) that points into that table and selects which data to use for that software thread. Details on this table structure are described in Section 15.6.2.

15.6.1 Hardware Feedback Interface Table Structure

This structure has a global header that is 16 bytes in size. Following this global header, there is one 8 byte entry per logical processor in the socket. The structure is designed as follows.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>Global Header</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>Per Logical Processor Entry</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>Per Logical Processor Entry</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16 + n*8</td>
<td>8</td>
<td>Per Logical Processor Entry</td>
</tr>
</tbody>
</table>
The global header is structured as shown in Table 15-5.

### Table 15-5. Hardware Feedback Interface Global Header Structure

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size (Bytes)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>Timestamp</td>
<td>Timestamp of when the table was last updated by hardware. This is a timestamp in crystal clock units. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>Performance Capability Flags</td>
<td>If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>Energy Efficiency Capability Changed</td>
<td>If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>Reserved</td>
<td>Initialized by the OS to 0.</td>
</tr>
</tbody>
</table>

The per logical processor scheduler feedback entry is structured as follows. The operating system can determine the index of the logical processor feedback entry for a logical processor using CPUID.06H:EDX[31:16] by executing CPUID on that logical processor.

### Table 15-6. Hardware Feedback Interface Logical Processor Entry Structure

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size (Bytes)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Performance Capability</td>
<td>Performance capability is an 8-bit value (0 .. 255) specifying the relative performance level of a logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. The OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback. CPUID.06H:EDX[0] enumerates support for Performance capability reporting.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Energy Efficiency Capability</td>
<td>Energy Efficiency capability is an 8-bit value (0 .. 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. The OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback. CPUID.06H:EDX[1] enumerates support for Energy Efficiency capability reporting.</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>Reserved</td>
<td>The OS scheduler is expected to initialize the Hardware Feedback Interface Structure to 0 prior to enabling Hardware Feedback.</td>
</tr>
</tbody>
</table>
15.6.2 Intel® Thread Director Table Structure

This structure has a global header that is at least 16 bytes in size. Its size depends on the number of classes and capabilities enumerated by the CPUID instruction (see notes below Table 15-7). Following this global header there are multiple Logical Processor related entries. The structure is designed as follows.

**Table 15-7. Intel® Thread Director Table Structure**

<table>
<thead>
<tr>
<th>Byte Offset1,2,3</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 + CP*CL + R85</td>
<td>Global Header</td>
</tr>
<tr>
<td>8 + CP*CL + R8</td>
<td>CL*CP + R8</td>
<td>Per Logical Processor Entry06</td>
</tr>
<tr>
<td>8 + 2*(CP*CL + R8)</td>
<td>CL*CP + R8</td>
<td>Per Logical Processor Entry1</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 + (N^2 -1)<em>(CP</em>CL + R8)</td>
<td>CL*CP +R8</td>
<td>Logical Processor EntryN-1</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Byte offset of Capability\_\_\_\_\_\_ of Class\_\_\_ of Change indication: 8 + CP * cl + cp.
2. Byte offset of LP Entry\_\_\_ of Class\_ of Change indication: 8 + (i+1) * (CP * CL + R8).
3. Byte offset of capability\_\_\_\_\_\_ of class\_\_ of LP Entry\_\_\_; 8 + (i+1) * (CP * CL + R8) + CP * cl + cp.
4. Both upper case CL and CP denote total number of classes and capabilities defined for the processor. Lower case cl and cp denote one instance of a class or capability. cl and cp are counted starting at zero. See “CPUID—CPU Identification” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A for the number of classes (CL) and the number of supported capabilities (CP). CP (# of capabilities): number of enumerated bits in CPUID.06H.0H.EDX[7:0] and CL (# of classes): CPUID.06H.0H.ECX[15:8].
5. R8 is the number of bytes necessary to round up the Capability Change Indication array and the Logical Processor Entry to whole multiple of 8 bytes.
6. Table size: 8 + (N+1)* (CP * CL + R8).
7. N is the number of Logical Processor Entries in the table. It is not greater than the number of Logical Processors on the socket, but may be lower.
8. The Operating System can determine the index for the Logical Processor Entry within the Intel Thread Director table using CPUID.06H.0H.EDX[31:16] by executing the CPUID instruction on that Logical Processor.
9. The Operating System should allocate space to accommodate for one such structure per socket in the system.
10. The Intel Thread Director table structure extends the Hardware Feedback Interface table structure without breaking backward compatibility. The Hardware Feedback Interface can be viewed as having two capabilities and a single class.

The global header is structured as shown in Table 15-8.
<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>Time-stamp of when the table was last updated by hardware. This is a time-stamp in crystal clock units. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>Class 0 Performance Capability Flags If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>8 + 1</td>
<td>1</td>
<td>Class 0 Energy Efficiency Capability Flags If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 + CP - 1</td>
<td>1</td>
<td>Class 0 change indication for Capability #(CP-1) if exists Unavailable for capabilities that are not enumerated.</td>
</tr>
<tr>
<td>8 + CP</td>
<td>1</td>
<td>Class 1 Performance Capability Flags If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>8 + CP + 1</td>
<td>1</td>
<td>Class 1 Energy Efficiency Capability Flags If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 + 2*CP - 1</td>
<td>1</td>
<td>Class 1 change indication for Capability #(CP-1) if exists Unavailable for capabilities that are not enumerated.</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 + (CL-1)*CP</td>
<td>1</td>
<td>Class #(CL-1) Performance Capability Flags If bit 0 is set to 1, indicates the performance capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>8 + (CL-1)*CP + 1</td>
<td>1</td>
<td>Class #(CL-1) Energy Efficiency Capability Flags If bit 0 is set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table and/or another bit in this field is updated. If bit 1 is set to 1, indicates a force idle/inject idle request to the OS for one or more logical processors. Initialized by the OS to 0.</td>
</tr>
</tbody>
</table>
The logical processor capability entry in the Intel Thread Director table is structured as follows.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Size (Bytes)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Performance Capability</strong></td>
<td>Class 0 Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a single logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td><strong>Energy Efficiency Capability</strong></td>
<td>Class 0 Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>CP - 1</td>
<td>1</td>
<td>Capability #(CP-1)</td>
<td>Class 0 Capability #(CP-1) if exists. If the capability does not exist (is not enumerated in the CPUID), the entry is unavailable (no space is reserved for future use here).</td>
</tr>
<tr>
<td>CP</td>
<td>R8</td>
<td>Padding</td>
<td>Padding to nearest multiple of 8 bytes. Initialized by the OS to 0 prior to enabling Intel Thread Director.</td>
</tr>
<tr>
<td>CP + R8</td>
<td>1</td>
<td>Performance Capability</td>
<td>Class 1 Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a single logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. Initialized by the OS to 0.</td>
</tr>
<tr>
<td>CP + 1</td>
<td>1</td>
<td>Energy Efficiency Capability</td>
<td>Class 1 Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. An Energy Efficiency capability of 255 indicates which logical processors have the highest relative energy efficiency capability. In addition, the value 255 is an explicit recommendation for the OS to consolidate work on those logical processors for energy efficiency reasons. Initialized by the OS to 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Capability #(CP-1)</strong></td>
<td>Class 1 Capability #(CP-1) if exists. If the capability does not exist (is not enumerated in the CPUID), the entry is unavailable (no space is reserved for future use here).</td>
</tr>
</tbody>
</table>
15.6.3 Intel® Thread Director Usage Model

When the OS Scheduler needs to decide which one of multiple free logical processors to assign to a software thread that is ready to execute, it can choose one of the following options:

1. The free logical processor with the highest performance value of that software thread class, if the system is scheduling for performance.

2. The free logical processor with the highest energy efficiency value of that software thread class, if the system is scheduling for energy efficiency.

When the OS Scheduler needs to decide which of two logical processors \((i,j)\) to assign to which of two software threads whose Class IDs are \(k1\) and \(k2\), it can compute the two performance ratios: \(\text{Perf Ratio}_1 = \frac{\text{Perf}_{ik1}}{\text{Perf}_{jk1}}\) and \(\text{Perf Ratio}_2 = \frac{\text{Perf}_{ik2}}{\text{Perf}_{jk2}}\), or two energy efficiency ratios: \(\text{Energy Eff. Ratio}_1 = \frac{\text{Energy Eff}_{ik1}}{\text{Energy Eff}_{jk1}}\) and \(\text{Energy Eff. Ratio}_2 = \frac{\text{Energy Eff}_{ik2}}{\text{Energy Eff}_{jk2}}\) between the two logical processors for each of the two classes, depending on whether the OS is scheduling for performance or for energy efficiency.

For example, assume that the system is scheduling for performance and that \(\text{Perf Ratio}_1 > \text{Perf Ratio}_2\). The OS Scheduler will assign the software thread whose Class ID is \(k1\) to logical processor \(i\), and the one whose Class ID is \(k2\) to logical processor \(j\).

When the two software threads in question belong to the same Class ID, the OS Scheduler can schedule to higher performance logical processors within that class when scheduling for performance and to higher energy efficiency logical processors within that class when scheduling for energy efficiency.

The highest to lowest ordering may be different between classes across cores and between the performance column and the energy efficiency column of the same class across cores.
15.6.4 Hardware Feedback Interface Pointer

The physical address of the HFI/Intel Thread Director structure is programmed by the OS into a package scoped MSR named IA32_HW_FEEDBACK_PTR. The MSR is structured as follows:

- Bits 63:MAXPHYADDR1 – Reserved.
- Bits MAXPHYADDR-1:12 – ADDR. This is the physical address of the page frame of the first page of this structure.
- Bits 11:1 – Reserved.
- Bit 0 – Valid. When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.

The address of this MSR is 17D0H. This MSR is cleared on processor reset to its default value of 0. It retains its value upon INIT.

CPUID.06H.0H:EDX[11:8] enumerates the size of memory that must be allocated by the OS for this structure.

15.6.5 Hardware Feedback Interface Configuration

The operating system enables HFI/Intel Thread Director using a package scoped MSR named IA32_HW_FEEDBACK_CONFIG (address 17D1H). This MSR is cleared on processor reset to its default value of 0. It retains its value upon INIT.

The MSR is structured as follows:

- Bits 63:2 – Reserved.
- Bit 1 – Enable Intel Thread Director (or multi-class support). Both bits 0 and 1 must be set for Intel Thread Director to be enabled. The extra class columns in the Intel Thread Director table are updated by hardware immediately following setting those two bits, as well as during run time as necessary.
- Bit 0 – Enable. When set to 1, enables HFI.

Before enabling HFI, the OS must set a valid hardware feedback interface structure using IA32_HW_FEEDBACK_PTR.

When the OS sets bit 0 only, the hardware populates class 0 capabilities only in the HFI structure. When bit 1 is set after or together with bit 0, the Intel Thread Director multi-class structure is populated.

When either the HFI structure or the Intel Thread Director structure are ready to use by the OS, the hardware sets IA32_PACKAGE_THERM_STATUS[bit 26]. An interrupt is generated by the hardware if IA32_PACKAGE_THERM_INTERRUPT[bit 25] is set.

When the OS clears bit 1 but leaves bit 0 set, Intel Thread Director is disabled, but HFI is kept operational. IA32_PACKAGE_THERM_STATUS[bit 26] is NOT set in this case.

Clearing bit 0 disables both HFI and Intel Thread Director, independent of the bit 1 state. Setting bit 1 to '1' while keeping bit 0 at '0' is an invalid combination which is quietly ignored.

When the OS clears bit 0, hardware sets the IA32_PACKAGE_THERM_STATUS[bit 26] to 1 to acknowledge disabling of the interface. The OS should wait for this bit to be set to 1 to reclaim the memory of the Intel Thread Director structure, as by setting IA32_PACKAGE_THERM_STATUS[bit 26] hardware guarantees not to write into the Intel Thread Director structure anymore.

The OS may clear bit 0 only after receiving an indication from the hardware that the structure initialization is complete via the same IA32_PACKAGE_THERM_STATUS[bit 26], following enabling of HFI/Intel Thread Director, thus avoiding a race condition between OS and hardware.

Bit 1 is valid only if CPUID[6].EAX[bit 23] is set. When setting this bit while support is not enumerated, the hardware generates #GP.

Table 15-10 summarizes the control options described above.

See Section 15.6.9 for details on scenarios where IA32_HW_FEEDBACK_CONFIG bits are implicitly reset by the hardware.

1. MAXPHYADDR is reported in CPUID.80000008H:EAX[7:0].
15.6.6 Hardware Feedback Interface Notifications

The IA32_PACKAGE_THERM_STATUS MSR is extended with a new bit, hardware feedback interface structure change status (bit 26, R/W/C0), to indicate that the hardware has updated the HFI/Intel Thread Director structure. This is a sticky bit and once set, indicates that the OS should read the structure to determine the change and adjust its scheduling decisions. Once set, the hardware will not generate any further updates to this structure until the OS clears this bit by writing 0.

The OS can enable interrupt-based notifications when the structure is updated by hardware through a new enable bit, hardware feedback interrupt enable (bit 25, R/W), in the IA32_PACKAGE_THERM_INTERRUPT MSR. When this bit is set to 1, it enables the generation of an interrupt when the HFI/Intel Thread Director structure is updated by hardware. When the enable bit transitions from 0 to 1, hardware will generate an initial notify, with the IA32_PACKAGE_THERM_STATUS bit 26 set to 1, to indicate that the OS should read the current HFI/Intel Thread Director structure.

<table>
<thead>
<tr>
<th>Pre-Bit 1</th>
<th>Pre-Bit 0</th>
<th>Post-Bit 1</th>
<th>Post-Bit 0</th>
<th>Action</th>
<th>IA32_PACKAGE_THERM_STATUS [bit 26] and Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Reset value. Both Hardware Feedback Interface and Intel Thread Director are disabled, no status bit set, no interrupt is generated.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Enable HFI structure. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Invalid option; quietly ignored by the hardware. No action (no update in the table).</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Enable HFI and Intel Thread Director. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Disable HFI support. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Disable HFI and Intel Thread Director. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Enable Intel Thread Director. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No action; keeps HFI and Intel Thread Director disabled. No action (no update in the table).</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Enable HFI. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Enable HFI and Intel Thread Director. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Disable HFI and Intel Thread Director. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Disable Intel Thread Director; keep HFI enabled. No action (no update in the table).</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Disable HFI and Intel Thread Director. Set the status bit and generate interrupt if enabled.</td>
<td></td>
</tr>
</tbody>
</table>
15.6.7 **Hardware Feedback Interface and Intel® Thread Director Structure Dynamic Update**

The HFI/Intel Thread Director structure can be updated dynamically during run time. Changes to the structure may occur to one or more of its cells. Such changes may occur for one or more logical processors. The hardware sets a non-zero value in the “capability change” field of the HFI/Intel Thread Director structure as an indication for the OS to read that capability for all logical processors. A thermal interrupt is delivered to indicate to the OS that the structure has just changed. Section 15.6.6 contains more details on this notification mechanism. The hardware clears all “capability change” fields after the OS resets `IA32_PACKAGE_THERM_STATUS[bit 26]`.

Zeroing a performance or energy efficiency cell hints to the OS that it is beneficial not to schedule software threads of that class on the associated logical processor for performance or energy efficiency reasons, respectively. If SMT is supported, it may be the case that the hardware zeroes one of the core’s logical processors only. Zeroing the performance and energy efficiency cells of all classes for a logical processor implies that the hardware provides a hint to the OS to completely avoid scheduling work on that logical processor.

Zeroing a performance and energy efficiency cell hint of a logical processor across all classes along with Capability Flag bit 1 set to 1 across all capabilities and classes, indicates to the OS to force idle logical processor(s), and if affinitized activity occurs on those logical processor(s), the OS should inject idle periods such that overall utilization of those idled cores has a minimal-to-no impact to power. Capability Flag bit 1 will be set to 1 while this hint persists.

When EE=255 is set on one or more logical processors, it represents a request that the OS attempt to consolidate work to those logical processors with EE=255. These requests are made when the SOC has knowledge that consolidating the work to a subset of cores will result in significantly better platform energy efficiency. Examples of consolidating work would include, but not limited to, delaying less important work as needed to provide compute bandwidth for more important work, and routing interrupts to the logical processors with EE=255. When the cumulative workload requires performance greater than that which is available on the subset of cores with EE=255, it is expected that the OS will scale the work out to additional logical processors.

A few example reasons for runtime changes in the HGS/Intel Thread Director Table:

- Over clocking run time update that changes the capability values.
- Change in run time physical constraints.
- Run time performance or energy efficiency optimization.
- Change in core frequency, voltage, or power budget.

15.6.8 **Logical Processor Scope Intel® Thread Director Configuration**

The operating system enables Intel Thread Director at the logical processor scope using a logical processor scope MSR named `IA32_HW_FEEDBACK_THREAD_CONFIG` (address 17D4H).

The MSR is read/write and is structured as follows:

- Bits 63:1 – Reserved.
- Bit 0 – Enables Intel Thread Director. When set to 1, logical processor scope Intel Thread Director is enabled. Default is 0 (disabled).

Bit 0 of the logical processor scope configuration MSR can be cleared or set regardless of the state of the HFI/Intel Thread Director package configuration MSR state. Even when bit 0 of all logical processor configuration MSRs is clear, the processor can still update the Intel Thread Director structure if it is still enabled in the `IA32_HW_FEEDBACK_CONFIG` package scope MSR. When the operating system clears `IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0]`, hardware clears the history accumulated on that logical processor which otherwise drives assigning the Class ID to the software thread that executes on that logical processor. As long as `IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0]` is set, the Class ID is available for the operating system to read, independent of the state of the package scope `IA32_HW_FEEDBACK_CONFIG[1:0]` bits.

See Section 15.6.9 for details on scenarios where `IA32_HW_FEEDBACK_CONFIG` bits are implicitly reset by the hardware.
15.6.9 Implicit Reset of Package and Logical Processor Scope Configuration MSRs

HFI/Intel Thread Director enable bits are reset by hardware in the following scenarios:

1. When GETSEC[SENDER] is executed:
   a. The processor implicitly resets the HFI/Intel Thread Director enable bits in the IA32_HW_FEEDBACK_-CONFIG MSR on all sockets (packages) in the system.
   b. The processor implicitly resets the Intel Thread Director enable bit in the IA32_HW_FEEDBACK_THREAD_-CONFIG MSR on all logical processors in the system across all sockets.
   c. The processor implicitly clears the HFI/Intel Thread Director table structure pointer in the IA32_HW_FEEDBACK_PTR package MSR across all sockets.

2. When GETSEC[ENTERACCS] is executed:
   a. The processor implicitly resets the HFI/Intel Thread Director enable bits in the IA32_HW_FEEDBACK_-CONFIG MSR on the socket where the GETSEC[ENTERACCS] instruction was executed.
   b. The processor implicitly resets the Intel Thread Director enable bit in the IA32_HW_FEEDBACK_THREAD_-CONFIG MSR on all logical processors on the socket where the GETSEC[ENTERACCS] instruction was executed.
   c. The processor implicitly clears the HFI/Intel Thread Director table structure pointer in the IA32_HW_FEEDBACK_PTR package MSR on the socket where the GETSEC[ENTERACCS] instruction was executed.

3. When an INIT or a wait-for-SIPI state are processed by a logical processor:
   a. The processor implicitly resets the Intel Thread Director enable bit in the IA32_HW_FEEDBACK_THREAD_-CONFIG MSR on that logical processor, whether the signal was in the context of GETSEC[ENTERACCS] or not.

If the OS requires HFI/Intel Thread Director to be active after exiting the measured environment or when processing a SIPI event, it should re-enable HFI/Intel Thread Director.

15.6.10 Logical Processor Scope Intel® Thread Director Run Time Characteristics

The processor provides the operating system with run time feedback about the execution characteristics of the software thread executing on logical processors whose IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0] is set.

The run time feedback is communicated via a read-only MSR named IA32_THREAD_FEEDBACK_CHAR. This is a logical processor scope MSR whose address is 17D2H. This MSR is structured as follows:

- Bit 63 – Valid bit. When set to 1 the OS Scheduler can use the Class ID (in bits 7:0) for its scheduling decisions. If this bit is 0, the Class ID field should be ignored. It is recommended that the OS uses the last known Class ID of the software thread for its scheduling decisions.
- Bits 62:8 – Reserved.
- Bits 7:0 – Application Class ID, pointing into the Intel Thread Director structure described in Table 15-8.

This MSR is valid only if CPUID.06H:EAX[bit 23] is set.

The valid bit is cleared by the hardware in the following cases:

- The hardware does not have enough information to provide the operating system with a reliable Class ID.
- The operating system cleared the logical processor’s IA32_HW_FEEDBACK_THREAD_CONFIG[bit 0] bit.

The HRESET instruction is executed while configured to reset the Intel Thread Director history.

15.6.11 Logical Processor Scope History

The operating system can reset the Intel Thread Director related history accumulated on the current logical processor it is executing on by issuing the HRESET instruction. See “CPUID—CPU Identification” in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for enumeration of the HRESET
15.6.11.1 Enabling Intel® Thread Director History Reset

The IA32_HRESET_ENABLE MSR is a read/write MSR and is structured as follows:

- Bits 63:32 – Reserved.
- Bits 31:1 – Reserved for other capabilities that can be reset by the HRESET instruction.
- Bit 0 – Enable reset of the Intel Thread Director history.

The operating system should set IA32_HRESET_ENABLE[bit 0] to enable Intel Thread Director history reset via the HRESET instruction.

15.6.11.2 Implicit Intel® Thread Director History Reset

The Intel Thread Director history is implicitly reset in the following scenarios:

1. When the processor enters or exits SMM mode and IA32_DEBUGCTL MSR.FREEZE_WHILE_SMM (bit 14) is set, the Intel Thread Director history is implicitly reset by the processor.
2. When GETSEC[SENTER] is executed, the processor resets the Intel Thread Director history on all logical processors in the system, including logical processors on other sockets (other than the one GETSEC[SENTER] is executed).
3. When GETSEC[ENTERACCS] is executed, the processor resets the Intel Thread Director history on the logical processor it is executed on.
4. When an INIT or a wait-for-SIPI state are processed by a logical processor, the Intel Thread Director history is reset whether the signal was a result of GETSEC[ENTERACCS] or not.

If the operating system requires HFI/Intel Thread Director to be active after exiting the measured environment or when processing a SIPI event, it should re-enable HFI/Intel Thread Director.

15.7 MWAIT EXTENSIONS FOR ADVANCED POWER MANAGEMENT

IA-32 processors may support a number of C-states that reduce power consumption for inactive states. Intel Core Solo and Intel Core Duo processors support both deeper C-state and MWAIT extensions that can be used by OS to implement power management policy.

Software should use CPUID to discover if a target processor supports the enumeration of MWAIT extensions. If CPUID.05H.ECX[Bit 0] = 1, the target processor supports MWAIT extensions and their enumeration (see Chapter 4, “Instruction Set Reference, M-U,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B).

If CPUID.05H.ECX[Bit 1] = 1, the target processor supports using interrupts as break-events for MWAIT, even when interrupts are disabled. Use this feature to measure C-state residency as follows:

- Software can write to bit 0 in the MWAIT Extensions register (ECX) when issuing an MWAIT to enter into a processor-specific C-state or sub C-state.
- When a processor comes out of an inactive C-state or sub C-state, software can read a timestamp before an interrupt service routine (ISR) is potentially executed.

CPUID.05H.EDX allows software to enumerate processor-specific C-states and sub C-states available for use with MWAIT extensions. IA-32 processors may support more than one C-state of a given C-state type. These are called sub C-states. Numerically higher C-state have higher power savings and latency (upon entering and exiting) than lower-numbered C-state.

2. The processor-specific C-states defined in MWAIT extensions can map to ACPI defined C-state types (C0, C1, C2, C3). The mapping relationship depends on the definition of a C-state by processor implementation and is exposed to OSPM by the BIOS using the ACPI defined _CST table.
At CPL = 0, system software can specify desired C-state and sub C-state by using the MWAIT hints register (EAX). Processors will not go to C-state and sub C-state deeper than what is specified by the hint register. If CPL > 0 and if MONITOR/MWAIT is supported at CPL > 0, the processor will only enter C1-state (regardless of the C-state request in the hints register).

Executing MWAIT generates an exception on processors operating at a privilege level where MONITOR/MWAIT are not supported.

**NOTE**

If MWAIT is used to enter a C-state (including sub C-state) that is numerically higher than C1, a store to the address range armed by MONITOR instruction will cause the processor to exit MWAIT if the store was originated by other processor agents. A store from non-processor agent may not cause the processor to exit MWAIT.

### 15.8 THERMAL MONITORING AND PROTECTION

The IA-32 architecture provides the following mechanisms for monitoring temperature and controlling thermal power:

1. The **catastrophic shutdown detector** forces processor execution to stop if the processor’s core temperature rises above a preset limit.
2. **Automatic and adaptive thermal monitoring mechanisms** force the processor to reduce its power consumption in order to operate within predetermined temperature limits.
3. The **software controlled clock modulation mechanism** permits operating systems to implement power management policies that reduce power consumption; this is in addition to the reduction offered by automatic thermal monitoring mechanisms.
4. **On-die digital thermal sensor and interrupt mechanisms** permit the OS to manage thermal conditions natively without relying on BIOS or other system board components.

The first mechanism is not visible to software. The other three mechanisms are visible to software using processor feature information returned by executing CPUID with EAX = 1.

The second mechanism includes:

- **Automatic thermal monitoring** provides two modes of operation. One mode modulates the clock duty cycle; the second mode changes the processor’s frequency. Both modes are used to control the core temperature of the processor.
- **Adaptive thermal monitoring** can provide flexible thermal management on processors made of multiple cores.

The third mechanism modulates the clock duty cycle of the processor. As shown in Figure 15-24, the phrase ‘duty cycle’ does not refer to the actual duty cycle of the clock signal. Instead it refers to the time period during which the clock signal is allowed to drive the processor chip. By using the stop clock mechanism to control how often the processor is clocked, processor power consumption can be modulated.

![Figure 15-24. Processor Modulation Through Stop-Clock Mechanism](image-url)
For previous automatic thermal monitoring mechanisms, software controlled mechanisms that changed processor operating parameters to impact changes in thermal conditions. Software did not have native access to the native thermal condition of the processor; nor could software alter the trigger condition that initiated software program control.

The fourth mechanism (listed above) provides access to an on-die digital thermal sensor using a model-specific register and uses an interrupt mechanism to alert software to initiate digital thermal monitoring.

**15.8.1 Catastrophic Shutdown Detector**

P6 family processors introduced a thermal sensor that acts as a catastrophic shutdown detector. This catastrophic shutdown detector was also implemented in Pentium 4, Intel Xeon and Pentium M processors. It is always enabled. When processor core temperature reaches a factory preset level, the sensor trips and processor execution is halted until after the next reset cycle.

**15.8.2 Thermal Monitor**

Pentium 4, Intel Xeon and Pentium M processors introduced a second temperature sensor that is factory-calibrated to trip when the processor’s core temperature crosses a level corresponding to the recommended thermal design envelop. The trip-temperature of the second sensor is calibrated below the temperature assigned to the catastrophic shutdown detector.

**15.8.2.1 Thermal Monitor 1**

The Pentium 4 processor uses the second temperature sensor in conjunction with a mechanism called Thermal Monitor 1 (TM1) to control the core temperature of the processor. TM1 controls the processor’s temperature by modulating the duty cycle of the processor clock. Modulation of duty cycles is processor model specific. Note that the processors STPCLK# pin is not used here; the stop-clock circuitry is controlled internally.

Support for TM1 is indicated by CPUID.1:EDX.TM.bit 29 = 1.

TM1 is enabled by setting the thermal-monitor enable flag (bit 3) in IA32_MISC_ENABLE; see Chapter 2, "Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. Following a power-up or reset, the flag is cleared, disabling TM1. BIOS is required to enable only one automatic thermal monitoring modes. Operating systems and applications must not disable the operation of these mechanisms.

**15.8.2.2 Thermal Monitor 2**

An additional automatic thermal protection mechanism, called Thermal Monitor 2 (TM2), was introduced in the Intel Pentium M processor and also incorporated in newer models of the Pentium 4 processor family. Intel Core Duo and Solo processors, and Intel Core 2 Duo processor family all support TM1 and TM2. TM2 controls the core temperature of the processor by reducing the operating frequency and voltage of the processor and offers a higher performance level for a given level of power reduction than TM1.

TM2 is triggered by the same temperature sensor as TM1. The mechanism to enable TM2 may be implemented differently across various IA-32 processor families with different CPUID signatures in the family encoding value, but will be uniform within an IA-32 processor family.

Support for TM2 is indicated by CPUID.1:ECX.TM2.bit 8 = 1.

**15.8.2.3 Two Methods for Enabling TM2**

On processors with CPUID family/model/stepping signature encoded as 0x69n or 0x6Dn (early Pentium M processors), TM2 is enabled if the TM_SELECT flag (bit 16) of the MSR_THERM2_CTL register is set to 1 (Figure 15-25) and bit 3 of the IA32_MISC_ENABLE register is set to 1.
Following a power-up or reset, the TM_SELECT flag may be cleared. BIOS is required to enable either TM1 or TM2. Operating systems and applications must not disable mechanisms that enable TM1 or TM2. If bit 3 of the IA32_MISC_ENABLE register is set and TM_SELECT flag of the MSR_THERM2_CTL register is cleared, TM1 is enabled.

On processors introduced after the Pentium 4 processor (this includes most Pentium M processors), the method used to enable TM2 is different. TM2 is enable by setting bit 13 of IA32_MISC_ENABLE register to 1. This applies to Intel Core Duo, Core Solo, and Intel Core 2 processor family.

The target operating frequency and voltage for the TM2 transition after TM2 is triggered is specified by the value written to MSR_THERM2_CTL, bits 15:0 (Figure 15-26). Following a power-up or reset, BIOS is required to enable at least one of these two thermal monitoring mechanisms. If both TM1 and TM2 are supported, BIOS may choose to enable TM2 instead of TM1. Operating systems and applications must not disable the mechanisms that enable TM1 or TM2; and they must not alter the value in bits 15:0 of the MSR_THERM2_CTL register.

**15.8.2.4 Performance State Transitions and Thermal Monitoring**

If the thermal control circuitry (TCC) for thermal monitor (TM1/TM2) is active, writes to the IA32_PERF_CTL will effect a new target operating point as follows:

- If TM1 is enabled and the TCC is engaged, the performance state transition can commence before the TCC is disengaged.
- If TM2 is enabled and the TCC is engaged, the performance state transition specified by a write to the IA32_PERF_CTL will commence after the TCC has disengaged.

**15.8.2.5 Thermal Status Information**

The status of the temperature sensor that triggers the thermal monitor (TM1/TM2) is indicated through the thermal status flag and thermal status log flag in the IA32_THERM_STATUS MSR (see Figure 15-27).

The functions of these flags are:

- **Thermal Status flag, bit 0** — When set, indicates that the processor core temperature is currently at the trip temperature of the thermal monitor and that the processor power consumption is being reduced via either TM1 or TM2, depending on which is enabled. When clear, the flag indicates that the core temperature is below the thermal monitor trip temperature. This flag is read only.
• **Thermal Status Log flag, bit 1** — When set, indicates that the thermal sensor has tripped since the last power-up or reset or since the last time that software cleared this flag. This flag is a sticky bit; once set it remains set until cleared by software or until a power-up or reset of the processor. The default state is clear.

![Figure 15-27. IA32_THERM_STATUS MSR](image)

After the second temperature sensor has been tripped, the thermal monitor (TM1/TM2) will remain engaged for a minimum time period (on the order of 1 ms). The thermal monitor will remain engaged until the processor core temperature drops below the preset trip temperature of the temperature sensor, taking hysteresis into account.

While the processor is in a stop-clock state, interrupts will be blocked from interrupting the processor. This holding off of interrupts increases the interrupt latency, but does not cause interrupts to be lost. Outstanding interrupts remain pending until clock modulation is complete.

The thermal monitor can be programmed to generate an interrupt to the processor when the thermal sensor is tripped; this is called a thermal interrupt. The delivery mode, mask, and vector for this interrupt can be programmed through the thermal entry in the local APIC’s LVT (see Section 11.5.1, "Local Vector Table"). The low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR (see Figure 15-28) control when the interrupt is generated; that is, on a transition from a temperature below the trip point to above and/or vice-versa.

![Figure 15-28. IA32_THERM_INTERRUPT MSR](image)

• **High-Temperature Interrupt Enable flag, bit 0** — Enables an interrupt to be generated on the transition from a low-temperature to a high-temperature when set; disables the interrupt when clear. (R/W).

• **Low-Temperature Interrupt Enable flag, bit 1** — Enables an interrupt to be generated on the transition from a high-temperature to a low-temperature when set; disables the interrupt when clear.

The thermal interrupt can be masked by the thermal LVT entry. After a power-up or reset, the low-temperature interrupt enable and high-temperature interrupt enable flags in the IA32_THERM_INTERRUPT MSR are cleared (interrupts are disabled) and the thermal LVT entry is set to mask interrupts. This interrupt should be handled either by the operating system or system management mode (SMM) code.

Note that the operation of the thermal monitoring mechanism has no effect upon the clock rate of the processor’s internal high-resolution timer (time stamp counter).

**15.8.2.6 Adaptive Thermal Monitor**

The Intel Core 2 Duo processor family supports enhanced thermal management mechanism, referred to as Adaptive Thermal Monitor (Adaptive TM).

Unlike TM2, Adaptive TM is not limited to one TM2 transition target. During a thermal trip event, Adaptive TM (if enabled) selects an optimal target operating point based on whether or not the current operating point has effectively cooled the processor.
Similar to TM2, Adaptive TM is enable by BIOS. The BIOS is required to test the TM1 and TM2 feature flags and enable all available thermal control mechanisms (including Adaptive TM) at platform initiation.

Adaptive TM is available only to a subset of processors that support TM2.

In each chip-multiprocessing (CMP) silicon die, each core has a unique thermal sensor that triggers independently. These thermal sensor can trigger TM1 or TM2 transitions in the same manner as described in Section 15.8.2.1 and Section 15.8.2.2. The trip point of the thermal sensor is not programmable by software since it is set during the fabrication of the processor.

Each thermal sensor in a processor core may be triggered independently to engage thermal management features. In Adaptive TM, both cores will transition to a lower frequency and/or lower voltage level if one sensor is triggered. Triggering of this sensor is visible to software via the thermal interrupt LVT entry in the local APIC of a given core.

### 15.8.3 Software Controlled Clock Modulation

Pentium 4, Intel Xeon and Pentium M processors also support software-controlled clock modulation. This provides a means for operating systems to implement a power management policy to reduce the power consumption of the processor. Here, the stop-clock duty cycle is controlled by software through the IA32_CLOCK_MODULATION MSR (see Figure 15-29).

The IA32_CLOCK_MODULATION MSR contains the following flag and field used to enable software-controlled clock modulation and to select the clock modulation duty cycle:

- **On-Demand Clock Modulation Enable, bit 4** — Enables on-demand software controlled clock modulation when set; disables software-controlled clock modulation when clear.
- **On-Demand Clock Modulation Duty Cycle, bits 1 through 3** — Selects the on-demand clock modulation duty cycle (see Table 15-11). This field is only active when the on-demand clock modulation enable flag is set.

Note that the on-demand clock modulation mechanism (like the thermal monitor) controls the processor’s stop-clock circuitry internally to modulate the clock signal. The STPCLK# pin is not used in this mechanism.

#### Table 15-11. On-Demand Clock Modulation Duty Cycle Field Encoding

<table>
<thead>
<tr>
<th>Duty Cycle Field Encoding</th>
<th>Duty Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>000B</td>
<td>Reserved</td>
</tr>
<tr>
<td>001B</td>
<td>12.5% (Default)</td>
</tr>
<tr>
<td>010B</td>
<td>25.0%</td>
</tr>
<tr>
<td>011B</td>
<td>37.5%</td>
</tr>
<tr>
<td>100B</td>
<td>50.0%</td>
</tr>
<tr>
<td>101B</td>
<td>63.5%</td>
</tr>
<tr>
<td>110B</td>
<td>75%</td>
</tr>
<tr>
<td>111B</td>
<td>87.5%</td>
</tr>
</tbody>
</table>
The on-demand clock modulation mechanism can be used to control processor power consumption. Power management software can write to the IA32_CLOCK_MODULATION MSR to enable clock modulation and to select a modulation duty cycle. If on-demand clock modulation and TM1 are both enabled and the thermal status of the processor is hot (bit 0 of the IA32_THERM_STATUS MSR is set), clock modulation at the duty cycle specified by TM1 takes precedence, regardless of the setting of the on-demand clock modulation duty cycle.

For Hyper-Threading Technology enabled processors, the IA32_CLOCK_MODULATION register is duplicated for each logical processor. In order for the On-demand clock modulation feature to work properly, the feature must be enabled on all the logical processors within a physical processor. If the programmed duty cycle is not identical for all the logical processors, the processor core clock will modulate to the highest duty cycle programmed for processors with any of the following CPUID DisplayFamily_DisplayModel signatures (see CPUID instruction in Chapter3, “Instruction Set Reference, A-L” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A): 06_1A, 06_1C, 06_1E, 06_1F, 06_25, 06_26, 06_27, 06_2C, 06_2E, 06_2F, 06_35, 06_36, and 0F_xx. For all other processors, if the programmed duty cycle is not identical for all logical processors in the same core, the processor core will modulate at the lowest programmed duty cycle.

For multiple processor cores in a physical package, each processor core can modulate to a programmed duty cycle independently.

For the P6 family processors, on-demand clock modulation was implemented through the chipset, which controlled clock modulation through the processor’s STPCLK# pin.

15.8.3.1 Extension of Software Controlled Clock Modulation

Extension of the software controlled clock modulation facility supports on-demand clock modulation duty cycle with 4-bit dynamic range (increased from 3-bit range). Granularity of clock modulation duty cycle is increased to 6.25% (compared to 12.5%).

Four bit dynamic range control is provided by using bit 0 in conjunction with bits 3:1 of the IA32_CLOCK_MODULATION MSR (see Figure 15-30).

![IA32_CLOCK_MODULATION MSR with Clock Modulation Extension](image-url)

Extension to software controlled clock modulation is supported only if CPUID.06H:EAX[Bit 5] = 1. If CPUID.06H:EAX[Bit 5] = 0, then bit 0 of IA32_CLOCK_MODULATION is reserved.

15.8.4 Detection of Thermal Monitor and Software Controlled Clock Modulation Facilities

The ACPI flag (bit 22) of the CPUID feature flags indicates the presence of the IA32_THERM_STATUS, IA32_THERM_INTERRUPT, IA32_CLOCK_MODULATION MSRs, and the xAPIC thermal LVT entry.

The TM1 flag (bit 29) of the CPUID feature flags indicates the presence of the automatic thermal monitoring facilities that modulate clock duty cycles.

15.8.4.1 Detection of Software Controlled Clock Modulation Extension

Processor’s support of software controlled clock modulation extension is indicated by CPUID.06H:EAX[Bit 5] = 1.
15.8.5 On Die Digital Thermal Sensors

On die digital thermal sensor can be read using an MSR (no I/O interface). In Intel Core Duo processors, each core has a unique digital sensor whose temperature is accessible using an MSR. The digital thermal sensor is the preferred method for reading the die temperature because (a) it is located closer to the hottest portions of the die, (b) it enables software to accurately track the die temperature and the potential activation of thermal throttling.

15.8.5.1 Digital Thermal Sensor Enumeration

The processor supports a digital thermal sensor if CPUID.06H.EAX[0] = 1. If the processor supports digital thermal sensor, EBX[bits 3:0] determine the number of thermal thresholds that are available for use.

Software sets thermal thresholds by using the IA32_THERM_INTERRUPT MSR. Software reads output of the digital thermal sensor using the IA32_THERM_STATUS MSR.

15.8.5.2 Reading the Digital Sensor

Unlike traditional analog thermal devices, the output of the digital thermal sensor is a temperature relative to the maximum supported operating temperature of the processor.

Temperature measurements returned by digital thermal sensors are always at or below TCC activation temperature. Critical temperature conditions are detected using the “Critical Temperature Status” bit. When this bit is set, the processor is operating at a critical temperature and immediate shutdown of the system should occur. Once the “Critical Temperature Status” bit is set, reliable operation is not guaranteed.

See Figure 15-31 for the layout of IA32_THERM_STATUS MSR. Bit fields include:

- **Thermal Status (bit 0, RO)** — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.
- **Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#). Bit 1 = 1 if PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.
- **PROCHOT# or FORCEPR# Event (bit 2, RO)** — Indicates whether PROCHOT# or FORCEPR# is being asserted by another agent on the platform.
• **PROCHOT# or FORCEPR# Log (bit 3, R/WC0)** — Sticky bit that indicates whether PROCHOT# or FORCEPR# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, PROCHOT# or FORCEPR# has been externally asserted. Software may clear this bit by writing a zero. External PROCHOT# assertions are only acknowledged if the Bidirectional Prochot feature is enabled.

• **Critical Temperature Status (bit 4, RO)** — Indicates whether the critical temperature detector output signal is currently active. If bit 4 = 1, the critical temperature detector output signal is currently active.

• **Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.

• **Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual temperature is currently higher than or equal to the value set in Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to TT#1. Quantitative information of actual temperature can be inferred from Digital Readout, bits 22:16.

• **Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Threshold #1 has been reached. Software may clear this bit by writing a zero.

• **Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual temperature is currently higher than or equal to the value set in Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to TT#2. Quantitative information of actual temperature can be inferred from Digital Readout, bits 22:16.

• **Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.

• **Power Limitation Status (bit 10, RO)** — Indicates whether the processor is currently operating below OS-requested P-state (specified in IA32_PERF_CTL) or OS-requested clock modulation duty cycle (specified in IA32_CLOCK_MODULATION). This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be delivered independently to IA32_PACKAGE_THERM_STATUS MSR.

• **Power Notification Log (bit 11, R/WC0)** — Sticky bit that indicates the processor went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification is indicated independently in IA32_PACKAGE_THERM_STATUS MSR.

• **Digital Readout (bits 22:16, RO)** — Digital temperature reading in 1 degree Celsius relative to the TCC activation temperature.
  0: TCC Activation temperature,
  1: (TCC Activation - 1), etc. See the processor’s data sheet for details regarding TCC activation.
  A lower reading in the Digital Readout field (bits 22:16) indicates a higher actual temperature.

• **Resolution in Degrees Celsius (bits 30:27, RO)** — Specifies the resolution (or tolerance) of the digital thermal sensor. The value is in degrees Celsius. It is recommended that new threshold values be offset from the current temperature by at least the resolution + 1 in order to avoid hysteresis of interrupt generation.

• **Reading Valid (bit 31, RO)** — Indicates if the digital readout in bits 22:16 is valid. The readout is valid if bit 31 = 1.

Changes to temperature can be detected using two thresholds (see Figure 15-32); one is set above and the other below the current temperature. These thresholds have the capability of generating interrupts using the core’s local APIC which software must then service. Note that the local APIC entries used by these thresholds are also used by the Intel® Thermal Monitor; it is up to software to determine the source of a specific interrupt.
See Figure 15-32 for the layout of IA32_THERM_INTERRUPT MSR. Bit fields include:

- **High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.

- **Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.

- **PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.

- **FORCEPR# Interrupt Enable (bit 3, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when FORCEPR# has been asserted by another agent on the platform. Bit 3 = 0 disables the interrupt; bit 3 = 1 enables the interrupt.

- **Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.

- **Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Digital Readout and is used to generate the Thermal Threshold #1 Status and Log bits as well as the Threshold #1 thermal interrupt delivery.

- **Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Threshold #1 setting in any direction. Bit 15 = 1 enables the interrupt; bit 15 = 0 disables the interrupt.

- **Threshold #2 Value (bits 22:16, R/W)** — A temperature threshold, encoded relative to the TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Digital Readout and is used to generate the Thermal Threshold #2 Status and Log bits as well as the Threshold #2 thermal interrupt delivery.

- **Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Threshold #2 setting in any direction. Bit 23 = 1 enables the interrupt; bit 23 = 0 disables the interrupt.

- **Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of power notification events when the processor went below OS-requested P-state or OS-requested clock modulation duty cycle. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be enabled independently by IA32_PACKAGE_THERM_INTERRUPT MSR.
15.8.6 Power Limit Notification

Platform firmware may be capable of specifying a power limit to restrict power delivered to a platform component, such as a physical processor package. This constraint imposed by platform firmware may occasionally cause the processor to operate below OS-requested P or T-state. A power limit notification event can be delivered using the existing thermal LVT entry in the local APIC.

Software can enumerate the presence of the processor’s support for power limit notification by verifying CPUID.06H:EAX[bit 4] = 1.

If CPUID.06H:EAX[bit 4] = 1, then IA32_THERM_INTERRUPT and IA32_THERM_STATUS provides the following facility to manage power limit notification:

- Bits 10 and 11 in IA32_THERM_STATUS informs software of the occurrence of processor operating below OS-requested P-state or clock modulation duty cycle setting (see Figure 15-31).
- Bit 24 in IA32_THERM_INTERRUPT enables the local APIC to deliver a thermal event when the processor went below OS-requested P-state or clock modulation duty cycle setting (see Figure 15-32).

15.9 PACKAGE LEVEL THERMAL MANAGEMENT

The thermal management facilities like IA32_THERM_INTERRUPT and IA32_THERM_STATUS are often implemented with a processor core granularity. To facilitate software manage thermal events from a package level granularity, two architectural MSR is provided for package level thermal management. The IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT MSRs use similar interfaces as IA32_THERM_STATUS and IA32_THERM_INTERRUPT, but are shared in each physical processor package.

Software can enumerate the presence of the processor’s support for package level thermal management facility (IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT) by verifying CPUID.06H:EAX[bit 6] = 1.

The layout of IA32_PACKAGE_THERM_STATUS MSR is shown in Figure 15-33.

|       | 63 | 32 | 31 | 27 | 23 | 22 | 16 | 15 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
|       |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| Reserved         |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Digital Readout |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Power Limit Notification Log |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Power Limit Notification Status |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Thermal Threshold #2 Log |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Thermal Threshold #2 Status |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Thermal Threshold #1 Log |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Thermal Threshold #1 Status |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Critical Temperature Log |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Critical Temperature Status |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG PROCHOT# or FORCEPR# Log |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG PROCHOT# or FORCEPR# Event |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Thermal Status Log |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |
| PKG Thermal Status |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |

*Package Thermal Status (bit 0, RO) — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) for the package is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.*
• **Package Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#) of the package. Bit 1 = 1 if package PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.

• **Package PROCHOT# Event (bit 2, RO)** — Indicates whether package PROCHOT# is being asserted by another agent on the platform.

• **Package PROCHOT# Log (bit 3, R/WC0)** — Sticky bit that indicates whether package PROCHOT# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, package PROCHOT# has been externally asserted. Software may clear this bit by writing a zero.

• **Package Critical Temperature Status (bit 4, RO)** — Indicates whether the package critical temperature detector output signal is currently active. If bit 4 = 1, the package critical temperature detector output signal is currently active.

• **Package Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the package critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.

• **Package Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to PTT#1. Quantitative information of actual package temperature can be inferred from Package Digital Readout, bits 22:16.

• **Package Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Package Threshold #1 has been reached. Software may clear this bit by writing a zero.

• **Package Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to PTT#2. Quantitative information of actual temperature can be inferred from Package Digital Readout, bits 22:16.

• **Package Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Package Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.

• **Package Power Limitation Status (bit 10, RO)** — Indicates package power limit is forcing one or more processors to operate below OS-requested P-state. Note that package power limit violation may be caused by processor cores or by devices residing in the uncore. Software can examine IA32_THERM_STATUS to determine if the cause originates from a processor core (see Figure 15-31).

• **Package Power Notification Log (bit 11, R/WC0)** — Sticky bit that indicates any processor in the package went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET.

• **Package Digital Readout (bits 22:16, RO)** — Package digital temperature reading in 1 degree Celsius relative to the package TCC activation temperature.
  0: Package TCC Activation temperature,
  1: (PTCC Activation - 1), etc. See the processor’s data sheet for details regarding PTCC activation. A lower reading in the Package Digital Readout field (bits 22:16) indicates a higher actual temperature.

The layout of IA32_PACKAGE_THERM_INTERRUPT MSR is shown in Figure 15-34.
- **Package High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a package high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.

- **Package Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.

- **Package PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when Package PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.

- **Package Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Package Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.

- **Package Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the Package TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #1 Status and Log bits as well as the Package Threshold #1 thermal interrupt delivery.

- **Package Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #1 setting in any direction. Bit 15 = 1 enables the interrupt; bit 15 = 0 disables the interrupt.

- **Package Threshold #2 Value (bits 22:16, R/W)** — A temperature threshold, encoded relative to the PTCC Activation temperature (using the same format as the Package Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #2 Status and Log bits as well as the Package Threshold #2 thermal interrupt delivery.

- **Package Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #2 setting in any direction. Bit 23 = 1 enables the interrupt; bit 23 = 0 disables the interrupt.

- **Package Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of package power notification events.

### 15.9.1 Support for Passive and Active cooling

Passive and active cooling may be controlled by the OS power management agent through ACPI control methods. On platforms providing package level thermal management facility described in the previous section, it is recommended that active cooling (FAN control) should be driven by measuring the package temperature using the `IA32_PACKAGE_THERM_INTERRUPT MSR`. 

---

**Figure 15-34. IA32_PACKAGE_THERM_INTERRUPT Register**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Pkg Power Limit Notification Enable</td>
</tr>
<tr>
<td>16</td>
<td>Pkg Threshold #2 Interrupt Enable</td>
</tr>
<tr>
<td>18</td>
<td>Pkg Threshold #2 Value</td>
</tr>
<tr>
<td>19</td>
<td>Pkg Threshold #1 Interrupt Enable</td>
</tr>
<tr>
<td>21</td>
<td>Pkg Threshold #1 Value</td>
</tr>
<tr>
<td>22</td>
<td>Pkg Overheat Interrupt Enable</td>
</tr>
<tr>
<td>23</td>
<td>Pkg PROCHOT# Interrupt Enable</td>
</tr>
<tr>
<td>24</td>
<td>Pkg Low Temp. Interrupt Enable</td>
</tr>
<tr>
<td>25</td>
<td>Pkg High Temp. Interrupt Enable</td>
</tr>
</tbody>
</table>
Passive cooling (frequency throttling) should be driven by measuring (a) the core and package temperatures, or (b) only the package temperature. If measured package temperature led the power management agent to choose which core to execute passive cooling, then all cores need to execute passive cooling. Core temperature is measured using the IA32_THERMAL_STATUS and IA32_THERMAL_INTERRUPT MSRs. The exact implementation details depend on the platform firmware and possible solutions include defining two different thermal zones (one for core temperature and passive cooling and the other for package temperature and active cooling).

15.10 PLATFORM SPECIFIC POWER MANAGEMENT SUPPORT

This section covers power management interfaces that are not architectural but addresses the power management needs of several platform specific components. Specifically, RAPL (Running Average Power Limit) interfaces provide mechanisms to enforce power consumption limit. Power limiting usages have specific usages in client and server platforms.

For client platform power limit control and for server platforms used in a data center, the following power and thermal related usages are desirable:

- Platform Thermal Management: Robust mechanisms to manage component, platform, and group-level thermals, either proactively or reactively (e.g., in response to a platform-level thermal trip point).
- Platform Power Limiting: More deterministic control over the system's power consumption, for example to meet battery life targets on rack-level or container-level power consumption goals within a datacenter.
- Power/Performance Budgeting: Efficient means to control the power consumed (and therefore the sustained performance delivered) within and across platforms.

The server and client usage models are addressed by RAPL interfaces, which expose multiple domains of power rationing within each processor socket. Generally, these RAPL domains may be viewed to include hierarchically:

- Package domain is the processor die.
- Memory domain includes the directly-attached DRAM; an additional power plane may constitute a separate domain.

In order to manage the power consumed across multiple sockets via RAPL, individual limits must be programmed for each processor complex. Programming specific RAPL domain across multiple sockets is not supported.

15.10.1 RAPL Interfaces

RAPL interfaces consist of non-architectural MSRs. Each RAPL domain supports the following set of capabilities, some of which are optional as stated below.

- Power limit - MSR interfaces to specify power limit, time window; lock bit, clamp bit etc.
- Energy Status - Power metering interface providing energy consumption information.
- Perf Status (Optional) - Interface providing information on the performance effects (regression) due to power limits. It is defined as a duration metric that measures the power limit effect in the respective domain. The meaning of duration is domain specific.
- Power Info (Optional) - Interface providing information on the range of parameters for a given domain, minimum power, maximum power etc.
- Policy (Optional) - 4-bit priority information that is a hint to hardware for dividing budget between sub-domains in a parent domain.

Each of the above capabilities requires specific units in order to describe them. Power is expressed in Watts, Time is expressed in Seconds, and Energy is expressed in Joules. Scaling factors are supplied to each unit to make the information presented meaningful in a finite number of bits. Units for power, energy, and time are exposed in the read-only MSR_RAPL_POWER_UNIT MSR.
POWER AND THERMAL MANAGEMENT

MSR_RAPL_POWER_UNIT (Figure 15-35) provides the following information across all RAPL domains:

- **Power Units** (bits 3:0): Power related information (in Watts) is based on the multiplier, $1/2^PU$; where PU is an unsigned integer represented by bits 3:0. Default value is 0011b, indicating power unit is in 1/8 Watts increment.

- **Energy Status Units** (bits 12:8): Energy related information (in Joules) is based on the multiplier, $1/2^ESU$; where ESU is an unsigned integer represented by bits 12:8. Default value is 10000b, indicating energy status unit is in 15.3 micro-Joules increment.

- **Time Units** (bits 19:16): Time related information (in Seconds) is based on the multiplier, $1/2^TU$; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating time unit is in 976 micro-seconds increment.

15.10.2 RAPL Domains and Platform Specificity

The specific RAPL domains available in a platform vary across product segments. Platforms targeting the client segment support the following RAPL domain hierarchy:

- Package
- Two power planes: PP0 and PP1 (PP1 may reflect to uncore devices)

Platforms targeting the server segment support the following RAPL domain hierarchy:

- Package
- Power plane: PP0
- DRAM

Each level of the RAPL hierarchy provides a respective set of RAPL interface MSRs. Table 15-12 lists the RAPL MSR interfaces available for each RAPL domain. The power limit MSR of each RAPL domain is located at offset 0 relative to an MSR base address which is non-architectural; see Chapter 2, "Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The energy status MSR of each domain is located at offset 1 relative to the MSR base address of respective domain.
The presence of the optional MSR interfaces (the three right-most columns of Table 15-12) may be model-specific. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for details.

### 15.10.3 Package RAPL Domain

The MSR interfaces defined for the package RAPL domain are:

- **MSR_PKG_POWER_LIMIT** allows software to set power limits for the package and measurement attributes associated with each limit,
- **MSR_PKG_ENERGY_STATUS** reports measured actual energy usage,
- **MSR_PKG_POWER_INFO** reports the package power range information for RAPL usage.

**MSR_PKG_PERF_STATUS** can report the performance impact of power limiting, but its availability may be model-specific.

**MSR_PKG_PERF_STATUS** can report the performance impact of power limiting, but its availability may be model-specific.

**MSR_PKG_POWER_LIMIT** allows a software agent to define power limitation for the package domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_PKG_POWER_LIMIT. Two power limits can be specified, corresponding to time windows of different sizes. Each power limit provides independent clamping control that would permit the processor cores to go below OS-requested state to meet the power limits. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_PKG_POWER_LIMIT (Figure 15-36) are:

- **Package Power Limit #1** (bits 14:0): Sets the average power usage limit of the package domain corresponding to time window # 1. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.
• **Enable Power Limit #1** (bit 15): 0 = disabled; 1 = enabled.

• **Package Clamping Limitation #1** (bit 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.

• **Time Window for Power Limit #1** (bits 23:17): Indicates the time window for power limit #1

  \[ \text{Time limit} = 2^Y \times (1.0 + Z/4.0) \times \text{Time Unit} \]

  Here “Y” is the unsigned integer value represented by bits 21:17, “Z” is an unsigned integer represented by bits 23:22. “Time Unit” is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

• **Package Power Limit #2** (bits 46:32): Sets the average power usage limit of the package domain corresponding to time window # 2. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.

• **Enable Power Limit #2** (bit 47): 0 = disabled; 1 = enabled.

• **Package Clamping Limitation #2** (bit 48): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.

• **Time Window for Power Limit #2** (bits 55:49): Indicates the time window for power limit #2

  \[ \text{Time limit} = 2^Y \times (1.0 + Z/4.0) \times \text{Time Unit} \]

  Here “Y” is the unsigned integer value represented by bits 53:49, “Z” is an unsigned integer represented by bits 55:54. “Time Unit” is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT. This field may have a hard-coded value in hardware and ignores values written by software.

• **Lock** (bit 63): If set, all write attempts to this MSR are ignored until next RESET.

MSR_PKG_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the package domain. This MSR is updated every ~1msec. It has a wraparound time of around 60 secs when power consumption is high, and may be longer otherwise.

**Figure 15-37. MSR_PKG_ENERGY_STATUS MSR**

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PKG_POWER_INFO is a read-only MSR. It reports the package power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the package domain. It also provides the largest possible time window for software to program the RAPL interface.

**Figure 15-38. MSR_PKG_POWER_INFO Register**

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
• **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the package domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.

• **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the package domain. The unit of this field is specified by the “Power Units” field of MSR_RAPL_POWER_UNIT.

• **Maximum Time Window** (bits 53:48): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_PKG_POWER_LIMIT. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

MSR_PKG_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific; see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

Accumulated Package Throttled Time (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the package has throttled. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

15.10.4 PP0/PP1 RAPL Domains

The MSR interfaces defined for the PP0 and PP1 domains are identical in layout. Generally, PP0 refers to the processor cores. The availability of PP1 RAPL domain interface is platform-specific. For a client platform, the PP1 domain refers to the power plane of a specific device in the uncore. For server platforms, the PP1 domain is not supported, but its PP0 domain supports the MSR_PP0_PERF_STATUS interface.

• MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow software to set power limits for the respective power plane domain.

• MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS report actual energy usage on a power plane.

• MSR_PP0_POLICY/MSR_PP1_POLICY allow software to adjust balance for respective power plane.

MSR_PP0_PERF_STATUS can report the performance impact of power limiting, but it is not available in client platforms.
MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow a software agent to define power limitation for the respective power plane domain. A lock mechanism in each power plane domain allows the software agent to enforce power limit settings independently. Once a lock bit is set, the power limit settings in that power plane are static and un-modifiable until next RESET.

The bit fields of MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT (Figure 15-40) are:

- **Power Limit** (bits 14:0): Sets the average power usage limit of the respective power plane domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.

- **Enable Power Limit** (bit 15): 0 = disabled; 1 = enabled.

- **Clamping Limitation** (bit 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.

- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit will be used by the processor. The numeric value encoded by bits 23:17 is represented by the product of $2^Y \times F$; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

- **Lock** (bit 31): If set, all write attempts to the MSR and corresponding policy MSRs are ignored until next RESET.

MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS are read-only MSRs. They report the actual energy use for the respective power plane domains. These MSRs are updated every ~1msec.

**Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since the last time this register was cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PP0_POLICY/MSR_PP1_POLICY provide balance power policy control for each power plane by providing inputs to the power budgeting management algorithm. On platforms that support PP0 (IA cores) and PP1 (uncore graphic device), the default values give priority to the non-IA power plane. These MSRs enable the PCU to balance power consumption between the IA cores and uncore graphic device.

**Priority Level** (bits 4:0): Priority level input to the PCU for respective power plane. PP0 covers the IA processor cores, PP1 covers the uncore graphic device. The value 31 is considered highest priority.

MSR_PP0_PERF_STATUS is a read-only MSR. It reports the total time for which the PP0 domain was throttled due to the power limits. This MSR is supported only in server platform. Throttling in this context is defined as going below the OS-requested P-state or T-state.
• **Accumulated PP0 Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the PP0 domain has throttled. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.

### 15.10.5 DRAM RAPL Domain

The MSR interfaces defined for the DRAM domains are supported only in the server platform. The MSR interfaces are:

- MSR_DRAM_POWER_LIMIT allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.
- MSR_DRAM_ENERGY_STATUS reports measured actual energy usage.
- MSR_DRAM_POWER_INFO reports the DRAM domain power range information for RAPL usage.
- MSR_DRAM_PERF_STATUS can report the performance impact of power limiting.

**MSR_DRAM_POWER_LIMIT** allows a software agent to define power limitation for the DRAM domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_DRAM_POWER_LIMIT. A power limit can be specified along with a time window. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_DRAM_POWER_LIMIT (Figure 15-44) are:

- **DRAM Power Limit #1** (bits 14:0): Sets the average power usage limit of the DRAM domain corresponding to time window # 1. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #1** (bit 15): 0 = disabled; 1 = enabled.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit will be used by the processor. The numeric value encoded by bits 23:17 is represented by the product of $2^Y \times F$; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.
- **Lock** (bit 31): If set, all write attempts to this MSR are ignored until next RESET.
MSR_DRAM_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the DRAM domain. This MSR is updated every ~1msec.

![Figure 15-45. MSR_DRAM_ENERGY_STATUS MSR](image)

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_POWER_INFO is a read-only MSR. It reports the DRAM power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the DRAM domain. It also provides the largest possible time window for software to program the RAPL interface.

![Figure 15-46. MSR_DRAM_POWER_INFO Register](image)

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 53:48): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_DRAM_POWER_LIMIT. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific; see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

![Figure 15-47. MSR_DRAM_PERF_STATUS MSR](image)
• **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the DRAM domain has throttled. The unit of this field is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.
This chapter describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. See Chapter 6, “Interrupt 18—Machine-Check Exception (#MC),” for more information on machine-check exceptions. A brief description of the Pentium processor’s machine check capability is also given.

Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

16.1 MACHINE-CHECK ARCHITECTURE

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors implement a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected.

The processor signals the detection of an uncorrected machine-check error by generating a machine-check exception (#MC), which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

Starting with 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH; see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. The processor can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors, referred to as corrected machine-check error interrupt (CMCI). See Section 16.5 for details.

Intel 64 processors supporting machine-check architecture and CMCI may also support an additional enhancement, namely, support for software recovery from certain uncorrected recoverable machine check errors. See Section 16.6 for details.

16.2 COMPATIBILITY WITH PENTIUM PROCESSOR

The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support and extend the machine-check exception mechanism introduced in the Pentium processor. The Pentium processor reports the following machine-check errors:

• Data parity errors during read cycles.
• Unsuccessful completion of a bus cycle.

The above errors are reported using the P5_MC_TYPE and P5_MC_ADDR MSRs (implementation specific for the Pentium processor). Use the RDMSR instruction to read these MSRs. See Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for the addresses.

The machine-check error reporting mechanism that Pentium processors use is similar to that used in Pentium 4, Intel Xeon, Intel Atom, and P6 family processors. When an error is detected, it is recorded in P5_MC_TYPE and P5_MC_ADDR; the processor then generates a machine-check exception (#MC).

16.3 MACHINE-CHECK MSRS

Machine check MSRs in the Pentium 4, Intel Atom, Intel Xeon, and P6 family processors consist of a set of global control and status registers and several error-reporting register banks. See Figure 16-1.

Each error-reporting bank is associated with a specific hardware unit (or group of hardware units) in the processor. Use RDMSR and WRMSR to read and to write these registers.

16.3.1 Machine-Check Global Control MSRs

The machine-check global control MSRs include the IA32_MCG_CAP, IA32_MCG_STATUS, and optionally IA32_MCG_CTL and IA32_MCG_EXT_CTL. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for the addresses of these registers.

16.3.1.1 IA32_MCG_CAP MSR

The IA32_MCG_CAP MSR is a read-only register that provides information about the machine-check architecture of the processor. Figure 16-2 shows the layout of the register.
Where:

- **Count field, bits 7:0** — Indicates the number of hardware unit error-reporting banks available in a particular processor implementation.

- **MCG_CTL_P (control MSR present) flag, bit 8** — Indicates that the processor implements the IA32_MCG_CTL MSR when set; this register is absent when clear.

- **MCG_EXT_P (extended MSRs present) flag, bit 9** — Indicates that the processor implements the extended machine-check state registers found starting at MSR address 180H; these registers are absent when clear.

- **MCG_CMCI_P (Corrected MC error counting/signaling extension present) flag, bit 10** — Indicates (when set) that extended state and associated MSRs necessary to support the reporting of an interrupt on a corrected MC error event and/or count threshold of corrected MC errors, is present. When this bit is set, it does not imply this feature is supported across all banks. Software should check the availability of the necessary logic on a bank by bank basis when using this signaling capability (i.e., bit 30 settable in individual IA32_MCi_CTL2 register).

- **MCG_TES_P (threshold-based error status present) flag, bit 11** — Indicates (when set) that bits 56:53 of the IA32_MCi_STATUS MSR are part of the architectural space. Bits 56:55 are reserved, and bits 54:53 are used to report threshold-based error status. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific.

- **MCG_EXT_CNT, bits 23:16** — Indicates the number of extended machine-check state registers present. This field is meaningful only when the MCG_EXT_P flag is set.

- **MCG_SER_P (software error recovery support present) flag, bit 24** — Indicates (when set) that the processor supports software error recovery (see Section 16.6), and IA32_MCi_STATUS MSR bits 56:55 are used to report the signaling of uncorrected recoverable errors and whether software must take recovery actions for uncorrected errors. Note that when MCG_TES_P is not set, bits 56:53 of the IA32_MCi_STATUS MSR are model-specific. If MCG_TES_P is set but MCG_SER_P is not set, bits 56:55 are reserved.

- **MCG_EMC_P (Enhanced Machine Check Capability) flag, bit 25** — Indicates (when set) that the processor supports enhanced machine check capabilities for firmware first signaling.

- **MCG_ELOG_P (extended error logging) flag, bit 26** — Indicates (when set) that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format “Generic Error Data Entry” that augments the data included in machine check bank registers.

For additional information about extended error logging interface, see https://cdrdv2.intel.com/v1/dl/getContent/671064.

- **MCG_LMCE_P (local machine check exception) flag, bit 27** — Indicates (when set) that the following interfaces are present:
— an extended state LMCE_S (located in bit 3 of IA32_MCG_STATUS), and
— the IA32_MCG_EXT_CTL MSR, necessary to support Local Machine Check Exception (LMCE).

A non-zero MCG_LMCE_P indicates that, when LMCE is enabled as described in Section 16.3.1.5, some machine check errors may be delivered to only a single logical processor.

The effect of writing to the IA32_MCG_CAP MSR is undefined.

### 16.3.1.2 IA32_MCG_STATUS MSR

The IA32_MCG_STATUS MSR describes the current state of the processor after a machine-check exception has occurred (see Figure 16-3).

![Figure 16-3. IA32_MCG_STATUS Register](image)

Where:

- **RIPV (restart IP valid) flag, bit 0** — Indicates (when set) that program execution can be restarted reliably at the instruction pointed to by the instruction pointer pushed on the stack when the machine-check exception is generated. When clear, the program cannot be reliably restarted at the pushed instruction pointer.

- **EIPV (error IP valid) flag, bit 1** — Indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.

- **MCIP (machine check in progress) flag, bit 2** — Indicates (when set) that a machine-check exception was generated. Software can set or clear this flag. The occurrence of a second Machine-Check Event while MCIP is set will cause the processor to enter a shutdown state. For information on processor behavior in the shutdown state, please refer to the description in Chapter 6, “Interrupt and Exception Handling”: “Interrupt 8—Double Fault Exception (#DF)”.

- **LMCE_S (local machine check exception signaled), bit 3** — Indicates (when set) that a local machine-check exception was generated. This indicates that the current machine-check event was delivered to only this logical processor.

Bits 63:04 in the IA32_MCG_STATUS MSR are reserved. An attempt to write to the IA32_MCG_STATUS MSR’s reserved bits with any value other than 0 results in #GP.

### 16.3.1.3 IA32_MCG_CTL MSR

The IA32_MCG_CTL MSR is present if the capability flag MCG_CTL_P is set in the IA32_MCG_CAP MSR. IA32_MCG_CTL controls the reporting of machine-check exceptions. If present, writing 1s to this register enables machine-check features and writing all 0s disables machine-check features. All other values are undefined and/or implementation specific.

### 16.3.1.4 IA32_MCG_EXT_CTL MSR

The IA32_MCG_EXT_CTL MSR is present if the capability flag MCG_LMCE_P is set in the IA32_MCG_CAP MSR.
IA32_MCG_EXT_CTL::LMCE_EN (bit 0) allows the processor to signal some MCEs to only a single logical processor in the system.

If MCG_LMCE_P is not set in IA32_MCG_CAP, or platform software has not enabled LMCE by setting IA32_FEATURE_CONTROL::LMCE_ENABLED (bit 20), any attempt to write or read IA32_MCG_EXT_CTL will result in #GP. The IA32_MCG_EXT_CTL MSR is cleared on RESET.

Figure 16-4 shows the layout of the IA32_MCG_EXT_CTL register

![Figure 16-4. IA32_MCG_EXT_CTL Register](image)

where

- **LMCE_EN (local machine check exception enable) flag, bit 0** - System software sets this to allow hardware to signal some MCEs to only a single logical processor. System software can set LMCE_EN only if the platform software has configured IA32_FEATURE_CONTROL as described in Section 16.3.1.5.

### 16.3.1.5 Enabling Local Machine Check

The intended usage of LMCE requires proper configuration by both platform software and system software. Platform software can turn LMCE on by setting bit 20 (LMCE_ENABLED) in IA32_FEATURE_CONTROL MSR (MSR address 3AH).

System software must ensure that both IA32_FEATURE_CONTROL::Lock (bit 0) and IA32_FEATURE_CONTROL::LMCE_ENABLED (bit 20) are set before attempting to set IA32_MCG_EXT_CTL::LMCE_EN (bit 0). When system software has enabled LMCE, then hardware will determine if a particular error can be delivered only to a single logical processor. Software should make no assumptions about the type of error that hardware can choose to deliver as LMCE. The severity and override rules stay the same as described in Table 16-8 to determine the recovery actions.

### 16.3.2 Error-Reporting Register Banks

Each error-reporting register bank can contain the IA32_MCi_CTL, IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC MSRs. The number of reporting banks is indicated by bits [7:0] of IA32_MCG_CAP MSR (address 0179H). The first error-reporting register (IA32_MC0_CTL) always starts at address 400H.


### 16.3.2.1 IA32_MCi_CTL MSRs

The IA32_MCi_CTL MSR controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EE)[j] represents a potential error. Setting an EE[j] flag enables signaling #MC of the associated error and clearing it disables signaling of the error. Error logging happens regardless of the setting of these bits. The processor drops writes to bits that are not implemented. Figure 16-5 shows the bit fields of IA32_MCi_CTL.
For P6 family processors, processors based on Intel Core microarchitecture (excluding those on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH and onward): the operating system or executive software must not modify the contents of the IA32_MC0_CTL MSR. This MSR is internally aliased to the EBL_CR_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the IA32_MC0_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the IA32_MC0_CTL MSR.

### 16.3.2.2 IA32_MCi_STATUS MSRS

Each IA32_MCi_STATUS MSR contains information related to a machine-check error if its VAL (valid) flag is set (see Figure 16-6). Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.

**NOTE**


Where:

- **MCA (machine-check architecture) error code field, bits 15:0** — Specifies the machine-check architecture-defined error code for the machine-check error condition detected. The machine-check architecture-defined error codes are guaranteed to be the same for all IA-32 processors that implement the machine-check architecture. See Section 16.9, “Interpreting the MCA Error Codes,” and Chapter 17, “Interpreting Machine Check Error Codes,” for information on machine-check error codes.

- **Model-specific error code field, bits 31:16** — Specifies the model-specific error code that uniquely identifies the machine-check error condition detected. The model-specific error codes may differ among IA-32 processors for the same machine-check error condition. See Chapter 17, “Interpreting Machine Check Error Codes,” for information on model-specific error codes.

- **Reserved, Error Status, and Other Information fields, bits 56:32** —
  - If IA32_MCG_CAP.MCG_EMC_P[bit 25] is 0, bits 37:32 contain “Other Information” that is implementation-specific and is not part of the machine-check architecture.
  - If IA32_MCG_CAP.MCG_EMC_P is 1, “Other Information” is in bits 36:32. If bit 37 is 0, system firmware has not changed the contents of IA32_MCi_STATUS. If bit 37 is 1, system firmware may have edited the contents of IA32_MCi_STATUS.
  - If IA32_MCG_CAP.MCG_CMCI_P[bit 10] is 0, bits 52:38 also contain “Other Information” (in the same sense as bits 37:32).
• If IA32_MCG_CAP[10] is 1, bits 52:38 are architectural (not model-specific). In this case, bits 52:38 reports the value of a 15 bit counter that increments each time a corrected error is observed by the MCA recording bank. This count value will continue to increment until cleared by software. The most significant bit, 52, is a sticky count overflow bit.

• If IA32_MCG_CAP[11] is 0, bits 56:53 also contain “Other Information” (in the same sense).

• If IA32_MCG_CAP[11] is 1, bits 56:53 are architectural (not model-specific). In this case, bits 56:53 have the following functionality:
  • If IA32_MCG_CAP[24] is 0, bits 56:55 are reserved.
  • If IA32_MCG_CAP[24] is 1, bits 56:55 are defined as follows:
    • S (Signaling) flag, bit 56 - Signals the reporting of UCR errors in this MC bank. See Section 16.6.2 for additional details.
    • AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. See Section 16.6.2 for additional details.
    • If the UC bit (Figure 16-6) is 1, bits 54:53 are undefined.
    • If the UC bit (Figure 16-6) is 0, bits 54:53 indicate the status of the hardware structure that reported the threshold-based error. See Table 16-1.

Table 16-1. Bits 54:53 in IA32_MCI_STATUS MSRs when IA32_MCG_CAP[11] = 1 and UC = 0

<table>
<thead>
<tr>
<th>Bits 54:53</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>No tracking - No hardware status tracking is provided for the structure reporting this event.</td>
</tr>
<tr>
<td>01</td>
<td>Green - Status tracking is provided for the structure posting the event; the current status is green (below threshold). For more information, see Section 16.4, “Enhanced Cache Error reporting.”</td>
</tr>
<tr>
<td>10</td>
<td>Yellow - Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). For more information, see Section 16.4, “Enhanced Cache Error reporting.”</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
• **PCC (processor context corrupt) flag, bit 57** — Indicates (when set) that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor’s state, and software may be able to restart. When system software supports recovery, consult Section 16.10.4, “Machine-Check Software Handler Guidelines for Error Recovery,” for additional rules that apply.

• **ADDRV (IA32_MCI_ADDR register valid) flag, bit 58** — Indicates (when set) that the IA32_MCI_ADDR register contains the address where the error occurred (see Section 16.3.2.3, “IA32_MCI_ADDR MSRs”). When clear, this flag indicates that the IA32_MCI_ADDR register is either not implemented or does not contain the address where the error occurred. Do not read these registers if they are not implemented in the processor.

• **MISCV (IA32_MCI_MISC register valid) flag, bit 59** — Indicates (when set) that the IA32_MCI_MISC register contains additional information regarding the error. When clear, this flag indicates that the IA32_MCI_MISC register is either not implemented or does not contain additional information regarding the error. Do not read these registers if they are not implemented in the processor.

• **EN (error enabled) flag, bit 60** — Indicates (when set) that the error was enabled by the associated EEj bit of the IA32_MCI_CTL register.

• **UC (error uncorrected) flag, bit 61** — Indicates (when set) that the processor did not or was not able to correct the error condition. When clear, this flag indicates that the processor was able to correct the error condition.

• **OVER (machine check overflow) flag, bit 62** — Indicates (when set) that a machine-check error occurred while the results of a previous error were still in the error-reporting register bank (that is, the VAL bit was already set in the IA32_MCI_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. In general, enabled errors are written over disabled errors, and uncorrected errors are written over corrected errors. Uncorrected errors are not written over previous valid uncorrected errors. When MCG_CMCI_P is set, corrected errors may not set the OVER flag. Software can rely on corrected error count in IA32_MCI_Status[52:38] to determine if any additional corrected errors may have occurred. For more information, see Section 16.3.2.2.1, “Overwrite Rules for Machine Check Overflow.”

• **VAL (IA32_MCI_STATUS register valid) flag, bit 63** — Indicates (when set) that the information within the IA32_MCI_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the IA32_MCI_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.

### 16.3.2.2.1 Overwrite Rules for Machine Check Overflow

Table 16-2 shows the overwrite rules for how to treat a second event if the MC bank already contains a valid log from an earlier event – that is, what to do if the valid bit for an MC bank already is set to 1. When more than one structure posts events in a given bank, these rules specify whether a new event will overwrite a previous posting or not. These rules define a priority for uncorrected (highest priority), yellow, and green/unmonitored (lowest priority) status.

In Table 16-2, the values in the two left-most columns are IA32_MCI_STATUS[54:53].

<table>
<thead>
<tr>
<th>First Event</th>
<th>Second Event</th>
<th>UC bit</th>
<th>Color</th>
<th>MCA Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>00/green</td>
<td>00/green</td>
<td>0</td>
<td>00/green</td>
<td>either</td>
</tr>
<tr>
<td>00/green</td>
<td>yellow</td>
<td>0</td>
<td>yellow</td>
<td>second error</td>
</tr>
<tr>
<td>yellow</td>
<td>00/green</td>
<td>0</td>
<td>yellow</td>
<td>first error</td>
</tr>
<tr>
<td>yellow</td>
<td>yellow</td>
<td>0</td>
<td>yellow</td>
<td>either</td>
</tr>
<tr>
<td>00/green/yellow</td>
<td>UC</td>
<td>1</td>
<td>undefined</td>
<td>second</td>
</tr>
<tr>
<td>UC</td>
<td>00/green/yellow</td>
<td>1</td>
<td>undefined</td>
<td>first</td>
</tr>
</tbody>
</table>

If a second event overwrites a previously posted event, the information (as guarded by individual valid bits) in the MCI bank is entirely from the second event. Similarly, if a first event is retained, all of the information previously posted for that event is retained. In general, when the logged error or the recent error is a corrected error, the OVER bit (MCI_Status[62]) may be set to indicate an overflow. When MCG_CMCI_P is set in IA32_MCG_CAP, system software should consult IA32_MCI_STATUS[52:38] to determine if additional corrected errors may have
occurred. Software may re-read IA32_MCi_STATUS, IA32_MCi_ADDR, and IA32_MCi_MISC appropriately to ensure data collected represent the last error logged.

After software polls a posting and clears the register, the valid bit is no longer set and therefore the meaning of the rest of the bits, including the yellow/green/00 status field in bits 54:53, is undefined. The yellow/green indication will only be posted for events associated with monitored structures – otherwise the unmonitored (00) code will be posted in IA32_MCi_STATUS[54:53].

16.3.2.3 IA32_MCi_ADDR MSRs

The IA32_MCi_ADDR MSR contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set (see Section 16-7, "IA32_MCi_ADDR MSR"). The IA32_MCi_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MCi_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception.

The address returned is an offset into a segment, linear address, or physical address. This depends on the error encountered. When these registers are implemented, these registers can be cleared by explicitly writing 0s to these registers. Writing 1s to these registers will cause a general-protection exception. See Figure 16-7.

<table>
<thead>
<tr>
<th>Processor Without Support For Intel 64 Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor With Support for Intel 64 Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
</tr>
<tr>
<td>Address*</td>
</tr>
</tbody>
</table>

* Useful bits in this field depend on the address methodology in use when the the register state is saved.

Figure 16-7. IA32_MCi_ADDR MSR

16.3.2.4 IA32_MCi_MISC MSRs

The IA32_MCi_MISC MSR contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set. The IA32_MCi_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MCi_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general protection exception. When implemented in a processor, these registers can be cleared by explicitly writing all 0s to them; writing 1s to them causes a general-protection exception to be generated. This register is not implemented in any of the error-reporting register banks for the P6 or Intel Atom family processors.

If both MISCV and IA32_MCG_CAP[24] are set, the IA32_MCi_MISC MSR is defined according to Figure 16-8 to support software recovery of uncorrected errors (see Section 16.6).
MACHINE-CHECK ARCHITECTURE

• Recoverable Address LSB (bits 5:0): The lowest valid recoverable address bit. Indicates the position of the least significant bit (LSB) of the recoverable error address. For example, if the processor logs bits [43:9] of the address, the LSB sub-field in IA32_MCi_MISC is 01001b (9 decimal). For this example, bits [8:0] of the recoverable error address in IA32_MCi_ADDR should be ignored.

• Address Mode (bits 8:6): Address mode for the address logged in IA32_MCi_ADDR. The supported address modes are given in Table 16-3.

Table 16-3. Address Mode in IA32_MCi_MISC[8:6]

<table>
<thead>
<tr>
<th>IA32_MCi_MISC[8:6] Encoding</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Segment Offset</td>
</tr>
<tr>
<td>001</td>
<td>Linear Address</td>
</tr>
<tr>
<td>010</td>
<td>Physical Address</td>
</tr>
<tr>
<td>011</td>
<td>Memory Address</td>
</tr>
<tr>
<td>100 to 110</td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td>Generic</td>
</tr>
</tbody>
</table>

• Model Specific Information (bits 63:9): Not architecturally defined.

16.3.2.4.2 IOMCA

Logging and Signaling of errors from PCI Express domain is governed by PCI Express Advanced Error Reporting (AER) architecture. PCI Express architecture divides errors in two categories: Uncorrectable errors and Correctable errors. Uncorrectable errors can further be classified as Fatal or Non-Fatal. Uncorrected IO errors are signaled to the system software either as AER Message Signaled Interrupt (MSI) or via platform specific mechanisms such as NMI. Generally, the signaling mechanism is controlled by BIOS and/or platform firmware. Certain processors support an error handling mode, called IOMCA mode, where Uncorrected PCI Express errors are signaled in the form of machine check exception and logged in machine check banks.

When a processor is in this mode, Uncorrected PCI Express errors are logged in the MCACOD field of the IA32_MCi_STATUS register as Generic I/O error. The corresponding MCA error code is defined in Table 15-8. IA32_MCi_Status[15:0] Simple Error Code Encoding. Machine check logging complements and does not replace AER logging that occurs inside the PCI Express hierarchy. The PCI Express Root Complex and Endpoints continue to log the error in accordance with PCI Express AER mechanism. In IOMCA mode, MCI_MISC register in the bank that logged IOMCA can optionally contain information that link the Machine Check logs with the AER logs or proprietary logs. In such a scenario, the machine check handler can utilize the contents of MCI_MISC to locate the next level of error logs corresponding to the same error. Specifically, if MCI_Status.MISCV is 1 and MCACOD is 0xE0B, MCI_MISC contains the PCI Express address of the Root Complex device containing the AER Logs. Software can consult the header type and class code registers in the Root Complex device's PCIe Configuration space to determine what type of device it is. This Root Complex device can either be a PCI Express Root Port, PCI Express Root Complex Event Collector or a proprietary device.
Errors that originate from PCI Express or Legacy Endpoints are logged in the corresponding Root Port in addition to the generating device. If MISCV=1 and MCi_MISC contains the address of the Root Port or a Root Complex Event collector, software can parse the AER logs to learn more about the error.

If MISCV=1 and MCi_MISC points to a device that is neither a Root Complex Event Collector not a Root Port, software must consult the Vendor ID/Device ID and use device specific knowledge to locate and interpret the error log registers. In some cases, the Root Complex device configuration space may not be accessible to the software and both the Vendor and Device ID read as 0xFFFF.

- The format of MCi_MISC for IOMCA errors is shown in Table 16-4.

### Table 16-4. Address Mode in IA32_MCi_MISC[8:6]

<table>
<thead>
<tr>
<th>63:40</th>
<th>39:32</th>
<th>31:16</th>
<th>15:9</th>
<th>8:6</th>
<th>5:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSVD</td>
<td>PCI Express Segment number</td>
<td>PCI Express Requestor ID</td>
<td>RSVD</td>
<td>ADDR MODE(^1)</td>
<td>RECOV ADDR LSB(^1)</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Not Applicable if ADDRV=0.

Refer to PCI Express Specification 3.0 for definition of PCI Express Requestor ID and AER architecture. Refer to PCI Firmware Specification 3.0 for an explanation of PCI Express Segment number and how software can access configuration space of a PCI Express device given the segment number and Requestor ID.

#### 16.3.2.5 IA32_MCi_CTL2 MSRs

The IA32_MCi_CTL2 MSR provides the programming interface to use corrected MC error signaling capability that is indicated by IA32_MCG_CAP[10] = 1. Software must check for the presence of IA32_MCi_CTL2 on a per-bank basis.

When IA32_MCG_CAP[10] = 1, the IA32_MCi_CTL2 MSR for each bank exists, i.e., reads and writes to these MSR are supported. However, signaling interface for corrected MC errors may not be supported in all banks.

The layout of IA32_MCi_CTL2 is shown in Figure 16-9.

- **Corrected error count threshold, bits 14:0** — Software must initialize this field. The value is compared with the corrected error count field in IA32_MCi_STATUS, bits 38 through 52. An overflow event is signaled to the CMCI LVT entry (see Table 11-1) in the APIC when the count value equals the threshold value. The new LVT entry in the APIC is at 02F0H offset from the APIC_BASE. If CMCI interface is not supported for a particular bank (but IA32_MCG_CAP[10] = 1), this field will always read 0.

- **CMCI_EN (Corrected error interrupt enable/disable/indicator), bits 30** — Software sets this bit to enable the generation of corrected machine-check error interrupt (CMCI). If CMCI interface is not supported for a particular bank (but IA32_MCG_CAP[10] = 1), this bit is writeable but will always return 0 for that bank. This bit also indicates CMCI is supported or not supported in the corresponding bank. See Section 16.5 for details of software detection of CMCI facility.
Some microarchitectural sub-systems that are the source of corrected MC errors may be shared by more than one logical processors. Consequently, the facilities for reporting MC errors and controlling mechanisms may be shared by more than one logical processors. For example, the IA32_MCi_CTL2 MSR is shared between logical processors sharing a processor core. Software is responsible to program IA32_MCi_CTL2 MSR in a consistent manner with CMCI delivery and usage.

After processor reset, IA32_MCi_CTL2 MSRs are zeroed.

### 16.3.2.6 IA32_MCG Extended Machine Check State MSRs

The Pentium 4 and Intel Xeon processors implement a variable number of extended machine-check state MSRs. The MCG EXT P flag in the IA32_MCG_CAP MSR indicates the presence of these extended registers, and the MCG EXT_CNT field indicates the number of these registers actually implemented. See Section 16.3.1.1, “IA32_MCG_CAP MSR.” Also see Table 16-5.

#### Table 16-5. Extended Machine Check State MSRs in Processors Without Support for Intel® 64 Architecture

<table>
<thead>
<tr>
<th>MSR</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MCG_EAX</td>
<td>180H</td>
<td>Contains state of the EAX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_EBX</td>
<td>181H</td>
<td>Contains state of the EBX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG ECX</td>
<td>182H</td>
<td>Contains state of the ECX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_EDX</td>
<td>183H</td>
<td>Contains state of the EDX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_ESI</td>
<td>184H</td>
<td>Contains state of the ESI register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG EDI</td>
<td>185H</td>
<td>Contains state of the EDI register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_EBP</td>
<td>186H</td>
<td>Contains state of the EBP register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_ESP</td>
<td>187H</td>
<td>Contains state of the ESP register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_MISC</td>
<td>18AH</td>
<td>Contains state of the EFLAGS register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG.Mask</td>
<td>18AH</td>
<td>When set, indicates that a page assist or page fault occurred during DS normal operation.</td>
</tr>
</tbody>
</table>

In processors with support for Intel 64 architecture, 64-bit machine check state MSRs are aliased to the legacy MSRs. In addition, there may be registers beyond IA32_MCG_MISC. These may include up to five reserved MSRs (IA32_MCG_RESERVED[1:5]) and save-state MSRs for registers introduced in 64-bit mode. See Table 16-6.

#### Table 16-6. Extended Machine Check State MSRs In Processors With Support for Intel® 64 Architecture

<table>
<thead>
<tr>
<th>MSR</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MCG_RAX</td>
<td>180H</td>
<td>Contains state of the RAX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RBX</td>
<td>181H</td>
<td>Contains state of the RBX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG RCX</td>
<td>182H</td>
<td>Contains state of the RCX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RDX</td>
<td>183H</td>
<td>Contains state of the RDX register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RSI</td>
<td>184H</td>
<td>Contains state of the RSI register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RDI</td>
<td>185H</td>
<td>Contains state of the RDI register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RBP</td>
<td>186H</td>
<td>Contains state of the RBP register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RSP</td>
<td>187H</td>
<td>Contains state of the RSP register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RFLAGS</td>
<td>188H</td>
<td>Contains state of the RFLAGS register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_RIP</td>
<td>189H</td>
<td>Contains state of the RIP register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_MISC</td>
<td>18AH</td>
<td>When set, indicates that a page assist or page fault occurred during DS normal operation.</td>
</tr>
</tbody>
</table>
When a machine-check error is detected on a Pentium 4 or Intel Xeon processor, the processor saves the state of the general-purpose registers, the R/EFLAGS register, and the R/EIP in these extended machine-check state MSRs. This information can be used by a debugger to analyze the error.

These registers are read/write to zero registers. This means software can read them; but if software writes to them, only all zeros is allowed. If software attempts to write a non-zero value into one of these registers, a general-protection (#GP) exception is generated. These registers are cleared on a hardware reset (power-up or RESET), but maintain their contents following a soft reset (INIT reset).

### 16.3.3 Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture

The Pentium processor reports machine-check errors using two registers: P5_MC_TYPE and P5_MC_ADDR. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors map these registers to the IA32_MCi_STATUS and IA32_MCiADDR in the error-reporting register bank. This bank reports on the same type of external bus errors reported in P5_MC_TYPE and P5_MC_ADDR.

The information in these registers can then be accessed in two ways:

- By reading the IA32_MCi_STATUS and IA32_MCiADDR registers as part of a general machine-check exception handler written for Pentium 4, Intel Atom and P6 family processors.
- By reading the P5_MC_TYPE and P5_MC_ADDR registers using the RDMSR instruction.

The second capability permits a machine-check exception handler written to run on a Pentium processor to be run on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor. There is a limitation in that information returned by the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors is encoded differently than information returned by the Pentium processor. To run a Pentium processor machine-check exception handler on a Pentium 4, Intel Xeon, Intel Atom, or P6 family processor; the handler must be written to interpret P5_MC_TYPE encodings correctly.

### 16.4 ENHANCED CACHE ERROR REPORTING

Starting with Intel Core Duo processors, cache error reporting was enhanced. In earlier Intel processors, cache status was based on the number of correction events that occurred in a cache. In the new paradigm, called “threshold-based error status”, cache status is based on the number of lines (ECC blocks) in a cache that incur repeated corrections. The threshold is chosen by Intel, based on various factors. If a processor supports threshold-based error status, it sets IA32_MCG_CAP[11] (MCG_TES_P) to 1; if not, to 0.

A processor that supports enhanced cache error reporting contains hardware that tracks the operating status of certain caches and provides an indicator of their “health”. The hardware reports a “green” status when the number of lines that incur repeated corrections is at or below a pre-defined threshold, and a “yellow” status when the

<table>
<thead>
<tr>
<th>MSR Address Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MCG_RSERVED[1:5] 18BH-18FH These registers, if present, are reserved.</td>
</tr>
<tr>
<td>IA32_MCG_R8 190H Contains state of the R8 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R9 191H Contains state of the R9 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R10 192H Contains state of the R10 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R11 193H Contains state of the R11 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R12 194H Contains state of the R12 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R13 195H Contains state of the R13 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R14 196H Contains state of the R14 register at the time of the machine-check error.</td>
</tr>
<tr>
<td>IA32_MCG_R15 197H Contains state of the R15 register at the time of the machine-check error.</td>
</tr>
</tbody>
</table>
number of affected lines exceeds the threshold. Yellow status means that the cache reporting the event is operating correctly, but you should schedule the system for servicing within a few weeks.

Intel recommends that you rely on this mechanism for structures supported by threshold-base error reporting. The CPU/system/platform response to a yellow event should be less severe than its response to an uncorrected error. An uncorrected error means that a serious error has actually occurred, whereas the yellow condition is a warning that the number of affected lines has exceeded the threshold but is not, in itself, a serious event: the error was corrected and system state was not compromised.

The green/yellow status indicator is not a foolproof early warning for an uncorrected error resulting from the failure of two bits in the same ECC block. Such a failure can occur and cause an uncorrected error before the yellow threshold is reached. However, the chance of an uncorrected error increases as the number of affected lines increases.

16.5 CORRECTED MACHINE CHECK ERROR INTERRUPT

Corrected machine-check error interrupt (CMCI) is an architectural enhancement to the machine-check architecture. It provides capabilities beyond those of threshold-based error reporting (Section 16.4). With threshold-based error reporting, software is limited to use periodic polling to query the status of hardware corrected MC errors. CMCI provides a signaling mechanism to deliver a local interrupt based on threshold values that software can program using the IA32_MCI_CTL2 MSRs.

CMCI is disabled by default. System software is required to enable CMCI for each IA32_MCi bank that support the reporting of hardware corrected errors if IA32_MCG_CAP[10] = 1.

System software use IA32_MCI_CTL2 MSR to enable/disable the CMCI capability for each bank and program threshold values into IA32_MCI_CTL2 MSR. CMCI is not affected by the CR4.MCE bit, and it is not affected by the IA32_MCI_CTL MSRs.

To detect the existence of thresholding for a given bank, software writes only bits 14:0 with the threshold value. If the bits persist, then thresholding is available (and CMCI is available). If the bits are all 0’s, then no thresholding exists. To detect that CMCI signaling exists, software writes a 1 to bit 30 of the MCI_CTL2 register. Upon subsequent read, if bit 30 = 0, no CMCI is available for this bank and no corrected or UCNA errors will be reported on this bank. If bit 30 = 1, then CMCI is available and enabled.

16.5.1 CMCI Local APIC Interface

The operation of CMCI is depicted in Figure 16-10.

![Figure 16-10. CMCI Behavior](image)

CMCI interrupt delivery is configured by writing to the LVT CMCI register entry in the local APIC register space at default address of APIC_BASE + 2F0H. A CMCI interrupt can be delivered to more than one logical processors if multiple logical processors are affected by the associated MC errors. For example, if a corrected bit error in a cache shared by two logical processors caused a CMCI, the interrupt will be delivered to both logical processors sharing...
that microarchitectural sub-system. Similarly, package level errors may cause CMCI to be delivered to all logical processors within the package. However, system level errors will not be handled by CMCI.

See Section 11.5.1, "Local Vector Table," for details regarding the LVT CMCI register.

### 16.5.2 System Software Recommendation for Managing CMCI and Machine Check Resources

System software must enable and manage CMCI, set up interrupt handlers to service CMCI interrupts delivered to affected logical processors, program CMCI LVT entry, and query machine check banks that are shared by more than one logical processors.

This section describes techniques system software can implement to manage CMCI initialization, service CMCI interrupts in a efficient manner to minimize contentions to access shared MSR resources.

#### 16.5.2.1 CMCI Initialization

Although a CMCI interrupt may be delivered to more than one logical processors depending on the nature of the corrected MC error, only one instance of the interrupt service routine needs to perform the necessary service and make queries to the machine-check banks. The following steps describes a technique that limits the amount of work the system has to do in response to a CMCI.

1. To provide maximum flexibility, system software should define per-thread data structure for each logical processor to allow equal-opportunity and efficient response to interrupt delivery. Specifically, the per-thread data structure should include a set of per-bank fields to track which machine check bank it needs to access in response to a delivered CMCI interrupt. The number of banks that needs to be tracked is determined by IA32_MCG_CAP[7:0].
2. Initialization of per-thread data structure. The initialization of per-thread data structure must be done serially on each logical processor in the system. The sequencing order to start the per-thread initialization between different logical processor is arbitrary. But it must observe the following specific detail to satisfy the shared nature of specific MSR resources:
   a. Each thread initializes its data structure to indicate that it does not own any MC bank registers.
   b. Each thread examines IA32_MCI_CTL2[30] indicator for each bank to determine if another thread has already claimed ownership of that bank.
      • If IA32_MCI_CTL2[30] had been set by another thread. This thread can not own bank / and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
      • If IA32_MCI_CTL2[30] = 0, proceed to step c.
   c. Check whether writing a 1 into IA32_MCI_CTL2[30] can return with 1 on a subsequent read to determine this bank can support CMCI.
      • If IA32_MCI_CTL2[30] = 0, this bank does not support CMCI. This thread can not own bank / and should proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
      • If IA32_MCI_CTL2[30] = 1, modify the per-thread data structure to indicate this thread claims ownership to the MC bank; proceed to initialize the error threshold count (bits 15:0) of that bank as described in Chapter 16, "CMCI Threshold Management". Then proceed to step b. and examine the next machine check bank until all of the machine check banks are exhausted.
3. After the thread has examined all of the machine check banks, it sees if it owns any MC banks to service CMCI. If any bank has been claimed by this thread:
   — Ensure that the CMCI interrupt handler has been set up as described in Chapter 16, “CMCI Interrupt Handler”.
   — Initialize the CMCI LVT entry, as described in Section 16.5.1, "CMCI Local APIC Interface."
   — Log and clear all of IA32_MCI_Status registers for the banks that this thread owns. This will allow new errors to be logged.
16.5.2.2 CMCI Threshold Management

The Corrected MC error threshold field, IA32_MCi_CTL2[14:0], is architecturally defined. Specifically, all these bits are writable by software, but different processor implementations may choose to implement less than 15 bits as threshold for the overflow comparison with IA32_MCi_STATUS[52:38]. The following describes techniques that software can manage CMCI threshold to be compatible with changes in implementation characteristics:

- Software can set the initial threshold value to 1 by writing 1 to IA32_MCi_CTL2[14:0]. This will cause overflow condition on every corrected MC error and generates a CMCI interrupt.
- To increase the threshold and reduce the frequency of CMCI servicing:
  - Find the maximum threshold value a given processor implementation supports. The steps are:
    a. Write 7FFFH to IA32_MCi_CTL2[14:0],
    b. Read back IA32_MCi_CTL2[14:0]; these 15 bits (14:0) contain the maximum threshold supported by the processor.
  - Increase the threshold to a value below the maximum value discovered using step a.

16.5.2.3 CMCI Interrupt Handler

The following describes techniques system software may consider to implement a CMCI service routine:

- The service routine examines its private per-thread data structure to check which set of MC banks it has ownership. If the thread does not have ownership of a given MC bank, proceed to the next MC bank. Ownership is determined at initialization time which is described in Section 16.5.2.1.

If the thread had claimed ownership to an MC bank, this technique will allow each logical processors to handle corrected MC errors independently and requires no synchronization to access shared MSR resources. Consult Example 16-5 for guidelines on logging when processing CMCI.

16.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS

Recovery of uncorrected recoverable machine check errors is an enhancement in machine-check architecture. The first processor that supports this feature is 45 nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_2EH; see the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. This allows system software to perform recovery action on a certain class of uncorrected errors and continue execution.

16.6.1 Detection of Software Error Recovery Support

Software must use bit 24 of IA32_MCG_CAP (MCG_SER_P) to detect the presence of software error recovery support (see Figure 16-2). When IA32_MCG_CAP[24] is set, this indicates that the processor supports software error recovery. When this bit is clear, this indicates that there is no support for error recovery from the processor and the primary responsibility of the machine check handler is logging the machine check error information and shutting down the system.

The new class of architectural MCA errors from which system software can attempt recovery is called Uncorrected Recoverable (UCR) Errors. UCR errors are uncorrected errors that have been detected and signaled but have not corrupted the processor context. For certain UCR errors, this means that once system software has performed a certain recovery action, it is possible to continue execution on this processor. UCR error reporting provides an error containment mechanism for data poisoning. The machine check handler will use the error log information from the error reporting registers to analyze and implement specific error recovery actions for UCR errors.

16.6.2 UCR Error Reporting and Logging

IA32_MCI_STATUS MSR is used for reporting UCR errors and existing corrected or uncorrected errors. The definitions of IA32_MCI_STATUS, including bit fields to identify UCR errors, is shown in Figure 16-6. UCR errors can be
signaled through either the corrected machine check interrupt (CMCI) or machine check exception (MCE) path depending on the type of the UCR error.

When IA32_MCG_CAP[24] is set, a UCR error is indicated by the following bit settings in the IA32_MCI_STATUS register:

- Valid (bit 63) = 1
- UC (bit 61) = 1
- PCC (bit 57) = 0

Additional information from the IA32_MCI_MISC and the IA32_MCI_ADDR registers for the UCR error are available when the ADDRV and the MISCV flags in the IA32_MCI_STATUS register are set (see Section 16.3.2.4). The MCA error code field of the IA32_MCI_STATUS register indicates the type of UCR error. System software can interpret the MCA error code field to analyze and identify the necessary recovery action for the given UCR error.

In addition, the IA32_MCI_STATUS register bit fields, bits 56:55, are defined (see Figure 16-6) to provide additional information to help system software to properly identify the necessary recovery action for the UCR error:

- S (Signaling) flag, bit 56 - Indicates (when set) that a machine check exception was generated for the UCR error reported in this MC bank and system software needs to check the AR flag and the MCA error code fields in the IA32_MCI_STATUS register to identify the necessary recovery action for this error. When the S flag in the IA32_MCI_STATUS register is clear, this UCR error was not signaled via a machine check exception and instead was reported as a corrected machine check (CMC). System software is not required to take any recovery action when the S flag in the IA32_MCI_STATUS register is clear.
- AR (Action Required) flag, bit 55 - Indicates (when set) that MCA error code specific recovery action must be performed by system software at the time this error was signaled. This recovery action must be completed successfully before any additional work is scheduled for this processor. When the RIPV flag in the IA32_MCG_STATUS is clear, an alternative execution stream needs to be provided; when the MCA error code specific recovery specific recovery action cannot be successfully completed, system software must shut down the system. When the AR flag in the IA32_MCI_STATUS register is clear, system software may still take MCA error code specific recovery action but this is optional; system software can safely resume program execution at the instruction pointer saved on the stack from the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.

Both the S and the AR flags in the IA32_MCI_STATUS register are defined to be sticky bits, which mean that once set, the processor does not clear them. Only software and good power-on reset can clear the S and the AR-flags. Both the S and the AR flags are only set when the processor reports the UCR errors (MCG_CAP[24] is set).

### 16.6.3 UCR Error Classification

With the S and AR flag encoding in the IA32_MCI_STATUS register, UCR errors can be classified as:

- Uncorrected no action required (UCNA) - is a UCR error that is not signaled via a machine check exception and, instead, is reported to system software as a corrected machine check error. UCNA errors indicate that some data in the system is corrupted, but the data has not been consumed and the processor state is valid and you may continue execution on this processor. UCNA errors require no action from system software to continue execution. A UCNA error is indicated with UC=1, PCC=0, S=0 and AR=0 in the IA32_MCI_STATUS register.

- Software recoverable action optional (SRAO) - a UCR error is signaled either via a machine check exception or CMCI. System software recovery action is optional and not required to continue execution from this machine check exception. SRAO errors indicate that some data in the system is corrupt, but the data has not been consumed and the processor state is valid. SRAO errors provide the additional error information for system software to perform a recovery action. An SRAO error when signaled as a machine check is indicated with UC=1, PCC=0, S=1, EN=1 and AR=0 in the IA32_MCI_STATUS register. In cases when SRAO is signaled via CMCI the error signature is indicated via UC=1, PCC=0, S=0. Recovery actions for SRAO errors are MCA error code specific. The MISCV and the ADDRV flags in the IA32_MCI_STATUS register are set when the additional error information is available from the IA32_MCI_MISC and the IA32_MCI_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MCI_STATUS register to identify the specific recovery action for a given SRAO error. If MISCV and ADDRV are not set, it is recommended that no system software error recovery be performed however, system software can resume execution.

- Software recoverable action required (SRAR) - a UCR error that requires system software to take a recovery action on this processor before scheduling another stream of execution on this processor. SRAR errors indicate
that the error was detected and raised at the point of the consumption in the execution flow. An SRAR error is indicated with UC=1, PCC=0, S=1, EN=1 and AR=1 in the IA32_MCi_STATUS register. Recovery actions are MCA error code specific. The MISCV and the ADDRV flags in the IA32_MCi_STATUS register are set when the additional error information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. System software needs to inspect the MCA error code fields in the IA32_MCi_STATUS register to identify the specific recovery action for a given SRAR error. If MISCV and ADDRV are not set, it is recommended that system software shutdown the system.

Table 16-7 summarizes UCR, corrected, and uncorrected errors.

<table>
<thead>
<tr>
<th>Type of Error</th>
<th>UC</th>
<th>EN</th>
<th>PCC</th>
<th>S</th>
<th>AR</th>
<th>Signaling</th>
<th>Software Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncorrected Error (UC)</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>MCE</td>
<td>If EN=1, reset the system, else log and OK to keep the system running.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRAR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>MCE</td>
<td>For known MCACOD, take specific recovery action; For unknown MCACOD, must bugcheck. If OVER=1, reset system, else take specific recovery action.</td>
<td>Cache to processor load error.</td>
</tr>
<tr>
<td>SRAO</td>
<td>1</td>
<td>x²</td>
<td>0</td>
<td>x²</td>
<td>0</td>
<td>MCE/CMC</td>
<td>For known MCACOD, take specific recovery action; For unknown MCACOD, OK to keep the system running.</td>
<td>Patrol scrub and explicit writeback poison errors.</td>
</tr>
<tr>
<td>UCNA</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CMC</td>
<td>Log the error and OK to keep the system running.</td>
<td>Poison detection error.</td>
</tr>
<tr>
<td>Corrected Error (CE)</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>CMC</td>
<td>Log the error and no corrective action required.</td>
<td>ECC in caches and memory.</td>
</tr>
</tbody>
</table>

NOTES:
1. SRAR, SRAO and UCNA errors are supported by the processor only when IA32_MCG_CAP[24] (MCG_SER_P) is set.
2. EN=1, S=1 when signaled via MCE. EN=x, S=0 when signaled via CMC.

### 16.6.4 UCR Error Overwrite Rules

In general, the overwrite rules are as follows:
- UCR errors will overwrite corrected errors.
- Uncorrected (PCC=1) errors overwrite UCR (PCC=0) errors.
- UCR errors are not written over previous UCR errors.
- Corrected errors do not write over previous UCR errors.

Regardless of whether the 1st error is retained or the 2nd error is overwritten over the 1st error, the OVER flag in the IA32_MCi_STATUS register will be set to indicate an overflow condition. As the S flag and AR flag in the IA32_MCi_STATUS register are defined to be sticky flags, a second event cannot clear these 2 flags once set, however the MC bank information may be filled in for the 2nd error. The table below shows the overwrite rules and how to treat a second error if the first event is already logged in a MC bank along with the resulting bit setting of the UC, PCC, and AR flags in the IA32_MCi_STATUS register. As UCNA and SRA0 errors do not require recovery action from system software to continue program execution, a system reset by system software is not required unless the AR flag or PCC flag is set for the UCR overflow case (OVER=1, VAL=1, UC=1, PCC=0).

Table 16-8 lists overwrite rules for uncorrected errors, corrected errors, and uncorrected recoverable errors.

<table>
<thead>
<tr>
<th>First Event</th>
<th>Second Event</th>
<th>UC</th>
<th>PCC</th>
<th>S</th>
<th>AR</th>
<th>MCA Bank</th>
<th>Reset System</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE</td>
<td>UCR</td>
<td>1</td>
<td>0</td>
<td>0 if UCNA, else 1</td>
<td>1 if SRAR, else 0</td>
<td>second</td>
<td>yes, if AR=1</td>
</tr>
<tr>
<td>UCR</td>
<td>CE</td>
<td>1</td>
<td>0</td>
<td>0 if UCNA, else 1</td>
<td>1 if SRAR, else 0</td>
<td>first</td>
<td>yes, if AR=1</td>
</tr>
</tbody>
</table>
16.7 MACHINE-CHECK AVAILABILITY

The machine-check architecture and machine-check exception (#MC) are model-specific features. Software can execute the CPUID instruction to determine whether a processor implements these features. Following the execution of the CPUID instruction, the settings of the MCA flag (bit 14) and MCE flag (bit 7) in EDX indicate whether the processor implements the machine-check architecture and machine-check exception.

16.8 MACHINE-CHECK INITIALIZATION

To use the processors machine-check architecture, software must initialize the processor to activate the machine-check exception and the error-reporting mechanism.

Example 16-1 gives pseudocode for performing this initialization. This pseudocode checks for the existence of the machine-check architecture and exception; it then enables machine-check exception and the error-reporting register banks. The pseudocode shown is compatible with the Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors.

Following power up or power cycling, IA32_MCi_STATUS registers are not guaranteed to have valid data until after they are initially cleared to zero by software (as shown in the initialization pseudocode in Example 16-1).

Example 16-1. Machine-Check Initialization Pseudocode

Check CPUID Feature Flags for MCE and MCA support
IF CPU supports MCE
THEN
  IF CPU supports MCA
  THEN
    IF (IA32_MCG_CAP.MCG_CTL_P = 1)
(* IA32_MCG_CTL register is present *)
    THEN
      IA32_MCG_CTL ← FFFFFFFF00000000H;
(* enables all MCA features *)
    FI
  IF (IA32_MCG_CAP.MCG_LMCE_P = 1 and IA32_FEATURE_CONTROL.LOCK = 1 and IA32_FEATURE_CONTROL.LMCE_ENABLED = 1)
(* IA32_MCG_EXT_CTL register is present and platform has enabled LMCE to permit system software to use LMCE *)
    THEN
      IA32_MCG_EXT_CTL ← IA32_MCG_EXT_CTL | 01H;
(* System software enables LMCE capability for hardware to signal MCE to a single logical processor*)
  FI
(* Determine number of error-reporting banks supported *)
COUNT ← IA32_MCG_CAP.Count;
MAX_BANK_NUMBER ← COUNT - 1;

IF (Processor Family is 6H and Processor EXTMODEL:MODEL is less than 1AH)
THEN
   (* Enable logging of all errors except for MC0_CTL register *)
   FOR error-reporting banks (1 through MAX_BANK_NUMBER)
       DO
           IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFFH;
       OD
ELSE
   (* Enable logging of all errors including MC0_CTL register *)
   FOR error-reporting banks (0 through MAX_BANK_NUMBER)
       DO
           IA32_MCi_CTL ← 0FFFFFFFFFFFFFFFFH;
       OD
FI

(* BIOS clears all errors only on power-on reset *)
IF (BIOS detects Power-on reset)
THEN
   FOR error-reporting banks (0 through MAX_BANK_NUMBER)
       DO
           IA32_MCi_STATUS ← 0;
       OD
ELSE
   FOR error-reporting banks (0 through MAX_BANK_NUMBER)
       DO
           (Optional for BIOS and OS) Log valid errors
           (OS only) IA32_MCi.STATUS ← 0;
       OD
FI

Setup the Machine Check Exception (#MC) handler for vector 18 in IDT

Set the MCE bit (bit 6) in CR4 register to enable Machine-Check Exceptions

16.9 INTERPRETING THE MCA ERROR CODES

When the processor detects a machine-check error condition, it writes a 16-bit error code to the MCA error code field of one of the IA32_MCi.STATUS registers and sets the VAL (valid) flag in that register. The processor may also write a 16-bit model-specific error code in the IA32_MCi.STATUS register depending on the implementation of the machine-check architecture of the processor.

The MCA error codes are architecturally defined for Intel 64 and IA-32 processors. To determine the cause of a machine-check exception, the machine-check exception handler must read the VAL flag for each IA32_MCI.STATUS register. If the flag is set, the machine check-exception handler must then read the MCA error code field of the register. It is the encoding of the MCA error code field [15:0] that determines the type of error being reported and not the register bank reporting it.

There are two types of MCA error codes: simple error codes and compound error codes.

16.9.1 Simple Error Codes

Table 16-9 shows the simple error codes. These unique codes indicate global error information.
**16.9.2 Compound Error Codes**

Compound error codes describe errors related to the TLBs, memory, caches, bus and interconnect logic, and internal timer. A set of sub-fields is common to all of compound errors. These sub-fields describe the type of access, level in the cache hierarchy, and type of request. Table 16-10 shows the general form of the compound error codes.

### Table 16-10. IA32_MCC_Status [15:0] Compound Error Code Encoding

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Cache Hierarchy</td>
<td>000F 0000 0000 11LL</td>
<td>Generic cache hierarchy error</td>
</tr>
<tr>
<td>TLB Errors</td>
<td>000F 0000 0000 01TT</td>
<td>(TT)TLB(LL) ERR</td>
</tr>
<tr>
<td>Memory Controller Errors</td>
<td>000F 0000 11MMM CCCC</td>
<td>{MMM}_CHANNEL{CCCC}_ERR</td>
</tr>
<tr>
<td>Cache Hierarchy Errors</td>
<td>000F 0001 00RRR TTLL</td>
<td>(TT)CACHE(LL)_{RRRR}_ERR</td>
</tr>
<tr>
<td>Extended Memory Errors</td>
<td>000F 0010 11MMM CCCC</td>
<td>{MMM}_CHANNEL{CCCC}_ERR</td>
</tr>
<tr>
<td>Bus and Interconnect Errors</td>
<td>000F 1PPR RRRR ILL</td>
<td>BUS(LL)<em>{PP}</em>(RRRR)_{II}(T) ERR</td>
</tr>
</tbody>
</table>

The “Interpretation” column in the table indicates the name of a compound error. The name is constructed by substituting mnemonics for the sub-field names given within curly braces. For example, the error code ICACHE1_RD_ERR is constructed from the form:

(TT)CACHE(LL)_{RRRR}_ERR,

where {TT} is replaced by I, {LL} is replaced by L1, and {RRRR} is replaced by RD.

For more information on the “Form” and “Interpretation” columns, see Section 16.9.2.1, “Correction Report Filtering (F) Bit,” through Section 16.9.2.5, “Bus and Interconnect Errors.”

### 16.9.2.1 Correction Report Filtering (F) Bit

Starting with Intel Core Duo processors, bit 12 in the “Form” column in Table 16-10 is used to indicate that a particular posting to a log may be the last posting for corrections in that line/entry, at least for some time.
• 0 in bit 12 indicates “normal” filtering (original P6/Pentium4/Atom/Xeon processor meaning).
• 1 in bit 12 indicates “corrected” filtering (filtering is activated for the line/entry in the posting). Filtering means that some or all of the subsequent corrections to this entry (in this structure) will not be posted. The enhanced error reporting introduced with the Intel Core Duo processors is based on tracking the lines affected by repeated corrections (see Section 16.4, “Enhanced Cache Error reporting”). This capability is indicated by IA32_MCG_CAP[11]. Only the first few correction events for a line are posted; subsequent redundant correction events to the same line are not posted. Uncorrected events are always posted.

The behavior of error filtering after crossing the yellow threshold is model-specific. Filtering has meaning only for corrected errors (UC=0 in IA32_MCi_STATUS MSR). System software must ignore filtering bit (12) for uncorrected errors.

16.9.2.2 Transaction Type (TT) Sub-Field
The 2-bit TT sub-field (Table 16-11) indicates the type of transaction (data, instruction, or generic). The sub-field applies to the TLB, cache, and interconnect error conditions. Note that interconnect error conditions are primarily associated with P6 family and Pentium processors, which utilize an external APIC bus separate from the system bus. The generic type is reported when the processor cannot determine the transaction type.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Mnemonic</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>I</td>
<td>00</td>
</tr>
<tr>
<td>Data</td>
<td>D</td>
<td>01</td>
</tr>
<tr>
<td>Generic</td>
<td>G</td>
<td>10</td>
</tr>
</tbody>
</table>

16.9.2.3 Level (LL) Sub-Field
The 2-bit LL sub-field (see Table 16-12) indicates the level in the memory hierarchy where the error occurred (level 0, level 1, level 2, or generic). The LL sub-field also applies to the TLB, cache, and interconnect error conditions. The Pentium 4, Intel Xeon, Intel Atom, and P6 family processors support two levels in the cache hierarchy and one level in the TLBs. Again, the generic type is reported when the processor cannot determine the hierarchy level.

<table>
<thead>
<tr>
<th>Hierarchy Level</th>
<th>Mnemonic</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>L0</td>
<td>00</td>
</tr>
<tr>
<td>Level 1</td>
<td>L1</td>
<td>01</td>
</tr>
<tr>
<td>Level 2</td>
<td>L2</td>
<td>10</td>
</tr>
<tr>
<td>Generic</td>
<td>LG</td>
<td>11</td>
</tr>
</tbody>
</table>

16.9.2.4 Request (RRRR) Sub-Field
The 4-bit RRRR sub-field (see Table 16-13) indicates the type of action associated with the error. Actions include read and write operations, prefetches, cache evictions, and snoops. Generic error is returned when the type of error cannot be determined. Generic read and generic write are returned when the processor cannot determine the type of instruction or data request that caused the error. Eviction and snoop requests apply only to the caches. All of the other requests apply to TLBs, caches, and interconnects.

<table>
<thead>
<tr>
<th>Hierarchy Level</th>
<th>Mnemonic</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>L0</td>
<td>00</td>
</tr>
<tr>
<td>Level 1</td>
<td>L1</td>
<td>01</td>
</tr>
<tr>
<td>Level 2</td>
<td>L2</td>
<td>10</td>
</tr>
<tr>
<td>Generic</td>
<td>LG</td>
<td>11</td>
</tr>
</tbody>
</table>
16.9.2.5 Bus and Interconnect Errors

The bus and interconnect errors are defined with the 2-bit PP (participation), 1-bit T (time-out), and 2-bit II (memory or I/O) sub-fields, in addition to the LL and RRRR sub-fields (see Table 16-14). The bus error conditions are implementation dependent and related to the type of bus implemented by the processor. Likewise, the interconnect error conditions are predicated on a specific implementation-dependent interconnect model that describes the connections between the different levels of the storage hierarchy. The type of bus is implementation dependent, and as such is not specified in this document. A bus or interconnect transaction consists of a request involving an address and a response.

Table 16-13. Encoding of Request (RRRR) Sub-Field

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Mnemonic</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Error</td>
<td>ERR</td>
<td>0000</td>
</tr>
<tr>
<td>Generic Read</td>
<td>RD</td>
<td>0001</td>
</tr>
<tr>
<td>Generic Write</td>
<td>WR</td>
<td>0010</td>
</tr>
<tr>
<td>Data Read</td>
<td>DRD</td>
<td>0011</td>
</tr>
<tr>
<td>Data Write</td>
<td>DWR</td>
<td>0100</td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>IRD</td>
<td>0101</td>
</tr>
<tr>
<td>Prefetch</td>
<td>PREFETCH</td>
<td>0110</td>
</tr>
<tr>
<td>Eviction</td>
<td>EVICT</td>
<td>0111</td>
</tr>
<tr>
<td>Snoop</td>
<td>SNOOP</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 16-14. Encodings of PP, T, and II Sub-Fields

<table>
<thead>
<tr>
<th>Sub-Field</th>
<th>Transaction</th>
<th>Mnemonic</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP (Participation)</td>
<td>Local processor* originated request</td>
<td>SRC</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>Local processor* responded to request</td>
<td>RES</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>Local processor* observed error as third party</td>
<td>OBS</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Generic</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>T (Time-out)</td>
<td>Request timed out</td>
<td>TIMEOUT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Request did not time out</td>
<td>NOTIMEOUT</td>
<td>0</td>
</tr>
<tr>
<td>II (Memory or I/O)</td>
<td>Memory Access</td>
<td>M</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>I/O</td>
<td>IO</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Other transaction</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

NOTE:
* Local processor differentiates the processor reporting the error from other system components (including the APIC, other processors, etc.).
16.9.2.6 Memory Controller and Extended Memory Errors

The memory controller errors are defined with the 3-bit MMM (memory transaction type), and 4-bit CCCC (channel) sub-fields. The encodings for MMM and CCCC are defined in Table 16-15. Extended Memory errors use the same encodings and are used to report errors in memory used as a cache.

<table>
<thead>
<tr>
<th>Sub-Field</th>
<th>Transaction Description</th>
<th>Mnemonic</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMM</td>
<td>Generic undefined request</td>
<td>GEN</td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>Memory read error</td>
<td>RD</td>
<td>001</td>
</tr>
<tr>
<td></td>
<td>Memory write error</td>
<td>WR</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td>Address/Command Error</td>
<td>AC</td>
<td>011</td>
</tr>
<tr>
<td></td>
<td>Memory Scrubbing Error</td>
<td>MS</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td></td>
<td>101-111</td>
</tr>
<tr>
<td>CCCC</td>
<td>Channel number</td>
<td>CHN</td>
<td>0000-1110</td>
</tr>
<tr>
<td></td>
<td>Channel not specified</td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>

Note that the CCCC channel number may be enumerated from zero separately by each memory controller on a system. On a multi-socket system, or a system with multiple memory controllers per socket, it is necessary to also consider which machine check bank logged the error. See Chapter 17 for details on specific implementations.

16.9.3 Architecturally Defined UCR Errors

Software recoverable compound error code are defined in this section.

16.9.3.1 Architecturally Defined SRAO Errors

The following two SRAO errors are architecturally defined.

- UCR Errors detected by memory controller scrubbing; and
- UCR Errors detected during L3 cache (L3) explicit writebacks.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 16-10). Their values and compound encoding format are given in Table 16-16.

<table>
<thead>
<tr>
<th>Type</th>
<th>MCACOD Value</th>
<th>MCA Error Code Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Scrubbing</td>
<td>C0H - CFH</td>
<td>0000_0000_1100_CCCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000F 0000 1MMM CCCC (Memory Controller Error), where</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory subfield MMM = 100B (memory scrubbing)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Channel subfield CCCC = channel # or generic</td>
</tr>
<tr>
<td>L3 Explicit Writeback</td>
<td>17AH</td>
<td>0000_0001_0111_1010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000F 0001 RRRR TTLL (Cache Hierarchy Error) where</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request subfields RRRR = 0111B (Eviction)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transaction Type subfields TT = 10B (Generic)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level subfields LL = 10B</td>
</tr>
</tbody>
</table>

NOTES:
1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.
Table 16-17 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAO errors.

**Table 16-17. IA32_MCi_STATUS Values for SRAO Errors**

<table>
<thead>
<tr>
<th>SRAO Error</th>
<th>Valid</th>
<th>OVER</th>
<th>UC</th>
<th>EN</th>
<th>MISCV</th>
<th>ADDRV</th>
<th>PCC</th>
<th>S</th>
<th>AR</th>
<th>MCACOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Scrubbing</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x'</td>
<td>0</td>
<td>C0H-CFH</td>
</tr>
<tr>
<td>L3 Explicit Writeback</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x'</td>
<td>0</td>
<td>17AH</td>
</tr>
</tbody>
</table>

**NOTES:**
1. When signaled as MCE, EN=1 and S=1. If error was signaled via CMC, then EN=x, and S=0.

For both the memory scrubbing and L3 explicit writeback errors, the ADDRV and MISCV flags in the IA32_MCi_MISC and the IA32_MCi_ADDR registers are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the memory scrubbing and L3 explicit writeback errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors as outlined in Section 16.10.4.1. If LMCE is supported and enabled, some errors (not limited to UCR errors) may be delivered to only a single logical processor. System software should consult IA32_MCG_STATUS.LMCE_S to determine if the MCE signaled is only to this logical processor.

IA32_MCi_STATUS banks can be shared by logical processors within a core or within the same package. So several logical processors may find an SRAO error in the shared IA32_MCi_STATUS bank but other processors do not find it in any of the IA32_MCi_STATUS banks. Table 16-18 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the memory scrubbing and L3 explicit writeback errors on both the reporting and non-reporting logical processors.

**Table 16-18. IA32_MCG_STATUS Flag Indication for SRAO Errors**

<table>
<thead>
<tr>
<th>SRAO Type</th>
<th>Reporting Logical Processors</th>
<th>Non-reporting Logical Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RIPV</td>
<td>EIPV</td>
</tr>
<tr>
<td>Memory Scrubbing</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>L3 Explicit Writeback</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**16.9.3.2 Architecturally Defined SRAR Errors**

The following two SRAR errors are architecturally defined.

- UCR Errors detected on data load; and
- UCR Errors detected on instruction fetch.

The MCA error code encodings for these two architecturally-defined UCR errors corresponds to sub-classes of compound MCA error codes (see Table 16-10). Their values and compound encoding format are given in Table 16-19.

**Table 16-19. MCA Compound Error Code Encoding for SRAR Errors**

<table>
<thead>
<tr>
<th>Type</th>
<th>MCACOD Value</th>
<th>MCA Error Code Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Load</td>
<td>134H</td>
<td>0000_0001_0011_0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000F 0001 RRRR TTLL (Cache Hierarchy Error), where</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request subfield RRRR = 0011B (Data Load)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transaction Type subfield TT= 01B (Data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level subfield LL = 00B (Level 0)</td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>150H</td>
<td>0000_0001_0101_0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000F 0001 RRRR TTLL (Cache Hierarchy Error), where</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request subfield RRRR = 0101B (Instruction Fetch)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transaction Type subfield TT= 00B (Instruction)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Level subfield LL = 00B (Level 0)</td>
</tr>
</tbody>
</table>
MACHINE-CHECK ARCHITECTURE

NOTES:
1. Note that for both of these errors the correction report filtering (F) bit (bit 12) of the MCA error must be ignored.

Table 16-20 lists values of relevant bit fields of IA32_MCi_STATUS for architecturally defined SRAR errors.

**Table 16-20. IA32_MCi_STATUS Values for SRAR Errors**

<table>
<thead>
<tr>
<th>SRAR Error</th>
<th>Valid</th>
<th>OVER</th>
<th>UC</th>
<th>EN</th>
<th>MISCV</th>
<th>ADDRV</th>
<th>PCC</th>
<th>S</th>
<th>AR</th>
<th>MCACOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Load</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>134H</td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>150H</td>
</tr>
</tbody>
</table>

For both the data load and instruction fetch errors, the ADDRV and MISCV flags in the IA32_MCi_STATUS register are set to indicate that the offending physical address information is available from the IA32_MCi_MISC and the IA32_MCi_ADDR registers. For the data load and instruction fetch errors, the address mode in the IA32_MCi_MISC register should be set as physical address mode (010b) and the address LSB information in the IA32_MCi_MISC register should indicate the lowest valid address bit in the address information provided from the IA32_MCi_ADDR register.

MCE signal is broadcast to all logical processors on the system on which the UCR errors are supported, except when the processor supports LMCE and LMCE is enabled by system software (see Section 16.3.1.5). The IA32_MC-G_STATUS MSR allows system software to distinguish the affected logical processor of an SRAR error amongst logical processors that observed SRAR via MCI_STATUS bank.

Table 16-21 shows the RIPV and EIPV flag indication in the IA32_MCG_STATUS register for the data load and instruction fetch errors on both the reporting and non-reporting logical processors. The recoverable SRAR error reported by a processor may be continuable, where the system software can interpret the context of continuable as follows: the error was isolated, contained. If software can rectify the error condition in the current instruction stream, the execution context on that logical processor can be continued without loss of information.

**Table 16-21. IA32_MCG_STATUS Flag Indication for SRAR Errors**

<table>
<thead>
<tr>
<th>SRAR Type</th>
<th>Affected Logical Processor</th>
<th>Non-Affected Logical Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RIPV</td>
<td>EIPV</td>
</tr>
<tr>
<td>Recoverable-continuable</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recoverable-not-continuable</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

NOTES:
1. see the definition of the context of “continuable” above and additional detail below.

**SRAR Error And Affected Logical Processors**

The affected logical processor is the one that has detected and raised an SRAR error at the point of the consumption in the execution flow. The affected logical processor should find the Data Load or the Instruction Fetch error information in the IA32_MCi_STATUS register that is reporting the SRAR error.

Table 16-21 list the actionable scenarios that system software can respond to an SRAR error on an affected logical processor according to RIPV and EIPV values:

- Recoverable-Continuable SRAR Error (RIPV=1, EIPV=1):
  - For Recoverable-Continuable SRAR errors, the affected logical processor should find that both the IA32_MC-G_STATUS.RIPV and the IA32_MCG_STATUS.EIPV flags are set, indicating that system software may be able to restart execution from the interrupted context if it is able to rectify the error condition. If system software cannot rectify the error condition then it must treat the error as a recoverable error where restarting execution with the interrupted context is not possible. Restarting without rectifying the error condition will result in most cases with another SRAR error on the same instruction.
- Recoverable-not-continuable SRAR Error \((\text{RIPV}=0, \text{EIPV}=x)\):
  
  For Recoverable-not-continuable errors, the affected logical processor should find that either
  - \(\text{IA32_MCG_STATUS.RIPV}= 0, \text{IA32_MCG_STATUS.EIPV}=1\), or
  - \(\text{IA32_MCG_STATUS.RIPV}= 0, \text{IA32_MCG_STATUS.EIPV}=0\).

  In either case, this indicates that the error is detected at the instruction pointer saved on the stack for this machine check exception and restarting execution with the interrupted context is not possible. System software may take the following recovery actions for the affected logical processor:
  - The current executing thread cannot be continued. System software must terminate the interrupted stream of execution and provide a new stream of execution on return from the machine check handler for the affected logical processor.

**SRAR Error And Non-Affected Logical Processors**

The logical processors that observed but not affected by an SRAR error should find that the RIPV flag in the \(\text{IA32_MCG_STATUS}\) register is set and the EIPV flag in the \(\text{IA32_MCG_STATUS}\) register is cleared, indicating that it is safe to restart the execution at the instruction saved on the stack for the machine check exception on these processors after the recovery action is successfully taken by system software.

### 16.9.4 Multiple MCA Errors

When multiple MCA errors are detected within a certain detection window, the processor may aggregate the reporting of these errors together as a single event, i.e., a single machine exception condition. If this occurs, system software may find multiple MCA errors logged in different MC banks on one logical processor or find multiple MCA errors logged across different processors for a single machine check broadcast event. In order to handle multiple UCR errors reported from a single machine check event and possibly recover from multiple errors, system software may consider the following:

- Whether it can recover from multiple errors is determined by the most severe error reported on the system. If the most severe error is found to be an unrecoverable error \((\text{VAL}=1, \text{UC}=1, \text{PCC}=1 \text{ and } \text{EN}=1)\) after system software examines the MC banks of all processors to which the MCA signal is broadcast, recovery from the multiple errors is not possible and system software needs to reset the system.

- When multiple recoverable errors are reported and no other fatal condition (e.g., overflowed condition for SRAR error) is found for the reported recoverable errors, it is possible for system software to recover from the multiple recoverable errors by taking necessary recovery action for each individual recoverable error. However, system software can no longer expect one to one relationship with the error information recorded in the \(\text{IA32_MCi_STATUS}\) register and the states of the RIPV and EIPV flags in the \(\text{IA32_MCG_STATUS}\) register as the states of the RIPV and the EIPV flags in the \(\text{IA32_MCG_STATUS}\) register may indicate the information for the most severe error recorded on the processor. System software is required to use the RIPV flag indication in the \(\text{IA32_MCG_STATUS}\) register to make a final decision of recoverability of the errors and find the restart-ability requirement after examining each \(\text{IA32_MCi_STATUS}\) register error information in the MC banks.

  In certain cases where system software observes more than one SRAR error logged for a single logical processor, it can no longer rely on affected threads as specified in Table 15-20 above. System software is recommended to reset the system if this condition is observed.

### 16.9.5 Machine-Check Error Codes Interpretation

Chapter 17, "Interpreting Machine Check Error Codes," provides information on interpreting the MCA error code, model-specific error code, and other information error code fields. For P6 family processors, information has been included on decoding external bus errors. For Pentium 4 and Intel Xeon processors; information is included on external bus, internal timer and cache hierarchy errors.
16.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE

The machine-check architecture and error logging can be used in three different ways:

- To detect machine errors during normal instruction execution, using the machine-check exception (#MC).
- To periodically check and log machine errors.
- To examine recoverable UCR errors, determine software recoverability and perform recovery actions via a machine-check exception handler or a corrected machine-check interrupt handler.

To use the machine-check exception, the operating system or executive software must provide a machine-check exception handler. This handler may need to be designed specifically for each family of processors.

A special program or utility is required to log machine errors.

Guidelines for writing a machine-check exception handler or a machine-error logging utility are given in the following sections.

16.10.1 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, Intel Atom, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When IA32_MCG_CAP[24] is clear, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32_MCG_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.
- The VAL (valid) flag in each IA32_MC.STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32_MC.STATUS register should be checked. See Section 16.9, "Interpreting the MCA Error Codes," for information that can be used to write an algorithm to interpret this field.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32_MC.STATUS register indicates whether the processor automatically corrected an error.
- The RIPV, PCC, and OVER flags in each IA32_MC.STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32_MCG_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture
MACHINE-CHECK ARCHITECTURE

does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

Example 16-2 gives typical steps carried out by a machine-check exception handler.

Example 16-2. Machine-Check Exception Handler Pseudocode

IF CPU supports MCE
    THEN
        IF CPU supports MCA
            THEN
                call errorlogging routine; (* returns restartability *)
            FI;
        ELSE (* Pentium(R) processor compatible *)
            READ P5_MC_ADDR
            READ P5_MC_TYPE;
            report RESTARTABILITY to console;
        FI;
    FI;
IF error is not restartable
    THEN
        report RESTARTABILITY to console;
        abort system;
    FI;
CLEAR MCIP flag in IA32_MCG_STATUS;

16.10.2 Pentium Processor Machine-Check Exception Handling

Machine-check exception handler on P6 family, Intel Atom and later processor families, should follow the guidelines described in Section 16.10.1 and Example 16-2 that check the processor’s support of MCA.

NOTE

On processors that support MCA (CPUID.1.EDX.MCA = 1) reading the P5_MC_TYPE and P5_MC_ADDR registers may produce invalid data.

When machine-check exceptions are enabled for the Pentium processor (MCE flag is set in control register CR4), the machine-check exception handler uses the RDMSR instruction to read the error type from the P5_MC_TYPE register and the machine check address from the P5_MC_ADDR register. The handler then normally reports these register values to the system console before aborting execution (see Example 16-2).

16.10.3 Logging Correctable Machine-Check Errors

The error handling routine for servicing the machine-check exceptions is responsible for logging uncorrected errors.

If a machine-check error is correctable, the processor does not generate a machine-check exception for it. To detect correctable machine-check errors, a utility program must be written that reads each of the machine-check error-reporting register banks and logs the results in an accounting file or data structure. This utility can be implemented in either of the following ways.

• A system daemon that polls the register banks on an infrequent basis, such as hourly or daily.
• A user-initiated application that polls the register banks and records the exceptions. Here, the actual polling service is provided by an operating-system driver or through the system call interface.
• An interrupt service routine servicing CMCI can read the MC banks and log the error. Please refer to Section 16.10.4.2 for guidelines on logging correctable machine checks.

Example 16-3 gives pseudocode for an error logging utility.
Example 16-3. Machine-Check Error Logging Pseudocode

Assume that execution is restartable;
IF the processor supports MCA
THEN
  FOR each bank of machine-check registers
    DO
      READ IA32_MCi_STATUS;
      IF VAL flag in IA32_MCi_STATUS = 1
        THEN
          IF ADDRv flag in IA32_MCi_STATUS = 1
            THEN READ IA32_MCi_ADDR;
          FI;
          IF MISCV flag in IA32_MCi_STATUS = 1
            THEN READ IA32_MCi_MISC;
          FI;
          IF MCIP flag in IA32_MCG_STATUS = 1
            (* Machine-check exception is in progress *)
            AND PCC flag in IA32_MCi_STATUS = 1
            OR RIPV flag in IA32_MCG_STATUS = 0
            (* execution is not restartable *)
            THEN
              RESTARTABILITY = FALSE;
              return RESTARTABILITY to calling procedure;
          FI;
        FI;
    OD;
  FI;
FI;

If the processor supports the machine-check architecture, the utility reads through the banks of error-reporting registers looking for valid register entries. It then saves the values of the IA32_MCi_STATUS, IA32_MCi_ADDR, IA32_MCi_MISC, and IA32_MCG_STATUS registers for each bank that is valid. The routine minimizes processing time by recording the raw data into a system data structure or file, reducing the overhead associated with polling. User utilities analyze the collected data in an off-line environment.

When the MCIP flag is set in the IA32_MCG_STATUS register, a machine-check exception is in progress and the machine-check exception handler has called the exception logging routine.

Once the logging process has been completed the exception-handling routine must determine whether execution can be restarted, which is usually possible when damage has not occurred (The PCC flag is clear, in the IA32_MCi_STATUS register) and when the processor can guarantee that execution is restartable (the RIPV flag is set in the IA32_MCG_STATUS register). If execution cannot be restarted, the system is not recoverable and the exception-handling routine should signal the console appropriately before returning the error status to the Operating System kernel for subsequent shutdown.

The machine-check architecture allows buffering of exceptions from a given error-reporting bank although the Pentium 4, Intel Xeon, Intel Atom, and P6 family processors do not implement this feature. The error logging routine should provide compatibility with future processors by reading each hardware error-reporting bank’s IA32_MCi_STATUS register and then writing 0s to clear the OVER and VAL flags in this register. The error logging utility should re-read the IA32_MCi_STATUS register for the bank ensuring that the valid bit is clear. The processor will write the next error into the register bank and set the VAL flags.

Additional information that should be stored by the exception-logging routine includes the processor’s time-stamp counter value, which provides a mechanism to indicate the frequency of exceptions. A multiprocessing operating system stores the identity of the processor node incurring the exception using a unique identifier, such as the processor’s APIC ID (see Section 11.8, “Handling Interrupts”).

The basic algorithm given in Example 16-3 can be modified to provide more robust recovery techniques. For example, software has the flexibility to attempt recovery using information unavailable to the hardware. Specifically, the machine-check exception handler can, after logging carefully analyze the error-reporting registers when the error-logging routine reports an error that does not allow execution to be restarted. These recovery techniques...
can use external bus related model-specific information provided with the error report to localize the source of the error within the system and determine the appropriate recovery strategy.

16.10.4 Machine-Check Software Handler Guidelines for Error Recovery

16.10.4.1 Machine-Check Exception Handler for Error Recovery

When writing a machine-check exception (MCE) handler to support software recovery from Uncorrected Recoverable (UCR) errors, consider the following:

- When IA32_MCG_CAP [24] is zero, there are no recoverable errors supported and all machine-check are fatal exceptions. The logging of status and error information is therefore a baseline implementation requirement.

- When IA32_MCG_CAP [24] is 1, certain uncorrected errors called uncorrected recoverable (UCR) errors may be software recoverable. The handler can analyze the reported error information, and in some cases attempt to recover from the uncorrected error and continue execution.

- For processors on which CPUID reports DisplayFamily_DisplayModel as 06H_0EH and onward, an MCA signal is broadcast to all logical processors in the system; see the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. Due to the potentially shared machine check MSR resources among the logical processors on the same package/core, the MCE handler may be required to synchronize with the other processors that received a machine check error and serialize access to the machine check registers when analyzing, logging, and clearing the information in the machine check registers.

  - On processors that indicate ability for local machine-check exception (MCG_LMCE_P), hardware can choose to report the error to only a single logical processor if system software has enabled LMCE by setting IA32_MCG_EXT_CTL[LMCE_EN] = 1 as outlined in Section 16.3.1.5.

- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and should not be checked.

- The MCE handler is primarily responsible for processing uncorrected errors. The UC flag in each IA32_MCI_Status register indicates whether the reported error was corrected (UC=0) or uncorrected (UC=1). The MCE handler can optionally log and clear the corrected errors in the MC banks if it can implement software algorithm to avoid the undesired race conditions with the CMCI or CMC polling handler.

- For uncorrectable errors, the EIPV flag in the IA32_MCG_STATUS register indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.

- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. When a machine check exception is generated, it is expected that the MCIP flag in the IA32_MCG_STATUS register is set to 1. If it is not set, this machine check was generated by either an INT 18 instruction or some piece of hardware signaling an interrupt with vector 18.

When IA32_MCG_CAP [24] is 1, the following rules can apply when writing a machine check exception (MCE) handler to support software recovery:

- The PCC flag in each IA32_MCI_STATUS register indicates whether recovery from the error is possible for uncorrected errors (UC=1). If the PCC flag is set for enabled uncorrected errors (UC=1 and EN=1), recovery is not possible. When recovery is not possible, the MCE handler typically records the error information and signals the operating system to reset the system.

- The RIPV flag in the IA32_MCG_STATUS register indicates whether restarting the program execution from the instruction pointer saved on the stack for the machine check exception is possible. When the RIPV is set, program execution can be restarted reliably when recovery is possible. If the RIPV flag is not set, program execution cannot be restarted reliably. In this case the recovery algorithm may involve terminating the current program execution and resuming an alternate thread of execution upon return from the machine check handler when recovery is possible. When recovery is not possible, the MCE handler signals the operating system to reset the system.
When the EN flag is zero but the VAL and UC flags are one in the IA32_MCI_STATUS register, the reported uncorrected error in this bank is not enabled. As uncorrected errors with the EN flag = 0 are not the source of machine check exceptions, the MCE handler should log and clear non-enabled errors when the S bit is set and should continue searching for enabled errors from the other IA32_MCI_STATUS registers. Note that when IA32_MCG_CAP [24] is 0, any uncorrected error condition (VAL =1 and UC=1) including the one with the EN flag cleared are fatal and the handler must signal the operating system to reset the system. For the errors that do not generate machine check exceptions, the EN flag has no meaning.

When the VAL flag is one, the UC flag is one, the EN flag is one and the PCC flag is zero in the IA32_MCI_STATUS register, the error in this bank is an uncorrected recoverable (UCR) error. The MCE handler needs to examine the S flag and the AR flag to find the type of the UCR error for software recovery and determine if software error recovery is possible.

When both the S and the AR flags are clear in the IA32_MCI_STATUS register for the UCR error (VAL=1, UC=1, EN=x and PCC=0), the error in this bank is an uncorrected no-action required error (UCNA). UCNA errors are uncorrected but do not require any OS recovery action to continue execution. These errors indicate that some data in the system is corrupt, but that data has not been consumed and may not be consumed. If that data is consumed a non-UCNA machine check exception will be generated. UCNA errors are signaled in the same way as corrected machine check errors and the CMCI and CMC polling handler is primarily responsible for handling UCNA errors. Like corrected errors, the MCA handler can optionally log and clear UCNA errors as long as it can avoid the undesired race condition with the CMCI or CMC polling handler. As UCNA errors are not the source of machine check exceptions, the MCA handler should continue searching for uncorrected or software recoverable errors in all other MC banks.

When the S flag in the IA32_MCI_STATUS register is set for the UCR error ((VAL=1, UC=1, EN=1 and PCC=0), the error in this bank is software recoverable and it was signaled through a machine-check exception. The AR flag in the IA32_MCI_STATUS register further clarifies the type of the software recoverable errors.

When the AR flag in the IA32_MCI_STATUS register is clear for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action optional (SRAO) error. The MCE handler and the operating system can analyze the IA32_MCI_STATUS [15:0] to implement MCA error code specific optional recovery action, but this recovery action is optional. System software can resume the program execution from the instruction pointer saved on the stack for the machine check exception when the RIPV flag in the IA32_MCG_STATUS register is set.

Even if the OVER flag in the IA32_MCI_STATUS register is set for the SRAO error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=0), the MCE handler can take recovery action for the SRAO error logged in the IA32_MCI_STATUS register. Since the recovery action for SRAO errors is optional, restarting the program execution from the instruction pointer saved on the stack for the machine check exception is still possible for the overflowed SRAO error if the RIPV flag in the IA32_MCG_STATUS is set.

When the AR flag in the IA32_MCI_STATUS register is set for the software recoverable error (VAL=1, UC=1, EN=1, PCC=0 and S=1), the error in this bank is a software recoverable action required (SRAR) error. The MCE handler and the operating system must take recovery action in order to continue execution after the machine-check exception. The MCA handler and the operating system need to analyze the IA32_MCI_STATUS [15:0] to determine the MCA error code specific recovery action. If no recovery action can be performed, the operating system must reset the system.

When the OVER flag in the IA32_MCI_STATUS register is set for the SRAR error (VAL=1, UC=1, EN=1, PCC=0, S=1 and AR=1), the MCE handler cannot take recovery action as the information of the SRAR error in the IA32_MCI_STATUS register was potentially lost due to the overflow condition. Since the recovery action for SRAR errors must be taken, the MCE handler must signal the operating system to reset the system.

When the MCE handler cannot find any uncorrected (VAL=1, UC=1 and EN=1) or any software recoverable errors (VAL=1, UC=1, EN=1, PCC=0 and S=1) in any of the IA32_MCI banks of the processors, this is an unexpected condition for the MCE handler and the handler should signal the operating system to reset the system.

Before returning from the machine-check exception handler, software must clear the MCIP flag in the IA32_MC-G_STATUS register. The MCIP flag is used to detect recursion. The machine-check architecture does not support recursion. When the processor receives a machine check when MCIP is set, it automatically enters the shutdown state.

Example 16-4 gives pseudocode for an MC exception handler that supports recovery of UCR.
Example 16-4. Machine-Check Error Handler Pseudocode Supporting UCR

MACHINE CHECK HANDLER: (* Called from INT 18 handler *)
NOERROR = TRUE;
ProcessorCount = 0;
IF CPU supports MCA 
  THEN
    RESTARTABILITY = TRUE;
    IF (Processor Family = 6 AND DisplayModel ≥ 0EH) OR (Processor Family > 6)
    THEN
        IF ( MCG_LMCE = 1)
            MCA_BROADCAST = FALSE;
        ELSE
            MCA_BROADCAST = TRUE;
        FI;
        Acquire SpinLock;
        ProcessorCount++;  (* Allowing one logical processor at a time to examine machine check registers *)
        CALL MCA ERROR PROCESSING; (* returns RESTARTABILITY and NOERROR *)
    ELSE
        MCA_BROADCAST = FALSE;
        (* Implement a rendezvous mechanism with the other processors if necessary *)
        CALL MCA ERROR PROCESSING;
    FI;
  ELSE (* Pentium(R) processor compatible *)
    READ P5_MC_ADDR
    READ P5_MC_TYPE;
    RESTARTABILITY = FALSE;
  FI;
IF NOERROR = TRUE 
  THEN
    IF NOT (MCG_RIPV = 1 AND MCG_EIPV = 0)
    THEN
        RESTARTABILITY = FALSE;
    FI
FI;
IF RESTARTABILITY = FALSE 
  THEN
    Report RESTARTABILITY to console;
    Reset system;
FI;
IF MCA_BROADCAST = TRUE 
  THEN
    IF ProcessorCount = MAX_PROCESSORS
        AND NOERROR = TRUE
    THEN
        Report RESTARTABILITY to console;
        Reset system;
    FI;
    Release SpinLock;
    Wait till ProcessorCount = MAX_PROCESSORS on system;
    (* implement a timeout and abort function if necessary *)
FI;
CLEAR IA32_MCG_STATUS;
RESUME Execution;
(* End of MACHINE CHECK HANDLER*)

MCA ERROR PROCESSING: (* MCA Error Processing Routine called from MCA Handler *)
IF MCIP flag in IA32_MCG_STATUS = 0 
    THEN (* MCIP=0 upon MCA is unexpected *)
        RESTARTABILITY = FALSE;
FI;
FOR each bank of machine-check registers
DO
    CLEAR_MC_BANK = FALSE;
    READ IA32_MCi_STATUS;
    IF VAL Flag in IA32_MCi_STATUS = 1
        THEN
            IF UC Flag in IA32_MCi_STATUS = 1
                THEN
                    IF Bit 24 in IA32_MCG_CAP = 0
                        THEN (* the processor does not support software error recovery *)
                            RESTARTABILITY = FALSE;
                            NOERROR = FALSE;
                            GOTO LOG MCA REGISTER;
                        FI;
                    (* the processor supports software error recovery *)
                    IF EN Flag in IA32_MCi_STATUS = 0 AND OVER Flag in IA32_MCi_STATUS=0
                        THEN (* It is a spurious MCA Log. Log and clear the register *)
                            CLEAR_MC_BANK = TRUE;
                            GOTO LOG MCA REGISTER;
                    FI;
                    IF PCC = 1 and EN = 1 in IA32_MCi_STATUS
                        THEN (* processor context might have been corrupted *)
                            RESTARTABILITY = FALSE;
                        ELSE (* It is a uncorrected recoverable (UCR) error *)
                            IF S Flag in IA32_MCi_STATUS = 0
                                THEN
                                    IF AR Flag in IA32_MCi_STATUS = 0
                                        THEN (* It is a uncorrected no action required (UCNA) error *)
                                            GOTO CONTINUE; (* let CMCI and CMC polling handler to process *)
                                        ELSE (* S=0, AR=1 is illegal *)
                                            FI;
                                    ELSE (* S in IA32_MCi_STATUS = 1 *)
                                        IF AR Flag in IA32_MCi_STATUS = 1
                                            THEN (* It is a software recoverable and action required (SRAR) error *)
                                                IF OVER Flag in IA32_MCi_STATUS = 1
                                                    THEN
                                                        RESTARTABILITY = FALSE;
                                                        NOERROR = FALSE;
                                                        GOTO LOG MCA REGISTER;
                                                ELSE (* It is a software recoverable and action optional (SRAO) error *)
                                                    IF OVER Flag in IA32_MCi_STATUS = 0 AND
                                                        MCACOD in IA32_MCi_STATUS is recognized
                                                        THEN
                                                            Implement MCACOD specific recovery action;
                                                            CLEAR_MC_BANK = TRUE;
                                                        ELSE
                                                            RESTARTABILITY = FALSE;
                                                        FI;
                                                ELSE (* It is a software recoverable and action optional (SRAO) error *)
                                                    IF OVER Flag in IA32_MCi_STATUS = 0 AND
                                                        MCACOD in IA32_MCi_STATUS is recognized
                                                        THEN
                                                            Implement MCACOD specific recovery action;
                                                        ELSE
                                                            CLEAR_MC_BANK = TRUE;
                                                        FI; AR
                                        FI;
                                FI;
                        ELSE (* It is a uncorrected recoverable (UCR) error *)
                            IF S Flag in IA32_MCi_STATUS = 0
                                THEN
                                    IF AR Flag in IA32_MCi_STATUS = 0
                                        THEN (* It is a uncorrected no action required (UCNA) error *)
                                            GOTO CONTINUE; (* let CMCI and CMC polling handler to process *)
                                        ELSE (* S=0, AR=1 is illegal *)
                                            FI;
                                    ELSE (* S in IA32_MCi_STATUS = 1 *)
                                        IF AR Flag in IA32_MCi_STATUS = 1
                                            THEN (* It is a software recoverable and action required (SRAR) error *)
                                                IF OVER Flag in IA32_MCi_STATUS = 1
                                                    THEN
                                                        RESTARTABILITY = FALSE;
                                                        NOERROR = FALSE;
                                                        GOTO LOG MCA REGISTER;
                                                ELSE (* It is a software recoverable and action optional (SRAO) error *)
                                                    IF OVER Flag in IA32_MCi_STATUS = 0 AND
                                                        MCACOD in IA32_MCi_STATUS is recognized
                                                        THEN
                                                            Implement MCACOD specific recovery action;
                                                        ELSE
                                                            CLEAR_MC_BANK = TRUE;
                                                        FI; AR
                                        FI;
                                FI;
                        FI;
                    FI;
GOTO LOG MCA REGISTER;
ELSE (* It is a corrected error; continue to the next IA32_MCi_STATUS *)
GOTO CONTINUE;
FI; UC
FI; VAL
LOG MCA REGISTER:
SAVE IA32_MCI_STATUS;
IF MISCV in IA32_MCI_STATUS
THEN
SAVE IA32_MCI_MISC;
FI;
IF ADDRV in IA32_MCI_STATUS
THEN
SAVE IA32_MCI_ADDR;
FI;
IF CLEAR_MC_BANK = TRUE
THEN
SET all 0 to IA32_MCI_STATUS;
IF MISCV in IA32_MCI_STATUS
THEN
SET all 0 to IA32_MCI_MISC;
FI;
IF ADDRV in IA32_MCI_STATUS
THEN
SET all 0 to IA32_MCI_ADDR;
FI;
FI;
CONTINUE:
OD;
(*END FOR*)
RETURN;
(* End of MCA ERROR PROCESSING*)

16.10.4.2 Corrected Machine-Check Handler for Error Recovery

When writing a corrected machine check handler, which is invoked as a result of CMCI or called from an OS CMC Polling dispatcher, consider the following:

- The VAL (valid) flag in each IA32_MCI_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank does not contain valid error information and does not need to be checked.

- The CMCI or CMC polling handler is responsible for logging and clearing corrected errors. The UC flag in each IA32_MCI_Status register indicates whether the reported error was corrected (UC=0) or not (UC=1).

- When IA32_MCG_CAP [24] is one, the CMC handler is also responsible for logging and clearing uncorrected no-action required (UCNA) errors. When the UC flag is one but the PCC, S, and AR flags are zero in the IA32_MCI_STATUS register, the reported error in this bank is an uncorrected no-action required (UCNA) error. In cases when SRAO error are signaled as UCNA error via CMCI, software can perform recovery for those errors identified in Table 16-16.

- In addition to corrected errors and UCNA errors, the CMC handler optionally logs uncorrected (UC=1 and PCC=1), software recoverable machine check errors (UC=1, PCC=0 and S=1), but should avoid clearing those errors from the MC banks. Clearing these errors may result in accidentally removing these errors before these errors are actually handled and processed by the MCE handler for attempted software error recovery.

Example 16-5 gives pseudocode for a CMCI handler with UCR support.
Example 16-5. Corrected Error Handler Pseudocode with UCR Support

Corrected Error HANDLER: (* Called from CMI handler or OS CMC Polling Dispatcher*)

IF CPU supports MCA THEN

 FOR each bank of machine-check registers
 DO
 READ IA32_MCI_STATUS;
 IF VAL flag in IA32_MCI_STATUS = 1 THEN
  IF UC Flag in IA32_MCI_STATUS = 0 (* It is a corrected error *)
   THEN
       GOTO LOG CMC ERROR;
   ELSE
       IF Bit 24 in IA32_MCG_CAP = 0
         THEN
             GOTO CONTINUE;
       FI;
   IF S Flag in IA32_MCI_STATUS = 0 AND AR Flag in IA32_MCI_STATUS = 0
     THEN (* It is a uncorrected no action required error *)
         GOTO LOG CMC ERROR
     FI;
   IF EN Flag in IA32_MCI_STATUS = 0
     THEN (* It is a spurious MCA error *)
         GOTO LOG CMC ERROR
     FI;
 FI;
 GOTO CONTINUE;
FI;
LOG CMC ERROR:
SAVE IA32_MCI_STATUS;
IF MISCV Flag in IA32_MCI_STATUS THEN
 SAVE IA32_MCI_MISC;
 SET all 0 to IA32_MCI_MISC;
FI;
IF ADDRV Flag in IA32_MCI_STATUS THEN
 SAVE IA32_MCI_ADDR;
 SET all 0 to IA32_MCI_ADDR
FI;
SET all 0 to IA32_MCI_STATUS;
CONTINUE:
OD; (*END FOR*)
FI;
CHAPTER 17
INTERPRETING MACHINE CHECK ERROR CODES

Encoding of the model-specific and other information fields is different across processor families. The differences are documented in the following sections.

17.1 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H, MACHINE ERROR CODES FOR MACHINE CHECK

This section provides information for interpreting additional model-specific fields for external bus errors relating to processor family 06H. The references to processor family 06H refers to only IA-32 processors with CPUID signatures listed in Table 17-1.

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Processor Families/Processor Number Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_06H</td>
<td>Intel® Core™ Duo processor, Intel® Core™ Solo processor</td>
</tr>
<tr>
<td>06_0DH</td>
<td>Intel Pentium M processor</td>
</tr>
<tr>
<td>06_09H</td>
<td>Intel Pentium M processor</td>
</tr>
<tr>
<td>06_7H, 06_08H, 06_0AH, 06_0BH</td>
<td>Intel Pentium III Xeon Processor, Intel Pentium III Processor</td>
</tr>
<tr>
<td>06_03H, 06_05H</td>
<td>Intel Pentium II Xeon Processor, Intel Pentium II Processor</td>
</tr>
<tr>
<td>06_01H</td>
<td>Intel Pentium Pro Processor</td>
</tr>
</tbody>
</table>

These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of 0000 1PPT RRRR IIII in the MCA error code field. See Chapter 16 for information on the interpretation of compound error codes. Incremental decoding information is listed in Table 17-2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes1</td>
<td>15:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>18:16</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
### Table 17-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes for Machine Check

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27:25</td>
<td>Bus Queue Error Type</td>
<td>000: BQ_ERR_HARD_TYPE error.</td>
<td>001: BQ_ERR_DOUBLE_TYPE error. 010: BQ_ERR_AERR2_TYPE error. 100: BQ_ERR_SINGLE_TYPE error. 101: BQ_ERR_AERR1_TYPE error.</td>
</tr>
<tr>
<td>28</td>
<td>FRC Error</td>
<td>1 if FRC error active.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>BERR</td>
<td>1 if BERR is driven.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Internal BINIT</td>
<td>1 if BINIT driven for this processor.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Other Information</td>
<td>34:32</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>35</td>
<td>External BINIT</td>
<td>1 if BINIT is received from external bus.</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>Response Parity Error</td>
<td>This bit is asserted in IA32_MCI_STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>Bus BINIT</td>
<td>This bit is asserted in IA32_MCI_STATUS if this component has received a hard error response on a split transaction one access that has needed to be split across the 64-bit external bus interface into two accesses).</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Timeout BINIT</td>
<td>This bit is asserted in IA32_MCI_STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 15-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock (the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.</td>
<td></td>
</tr>
<tr>
<td>41:39</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Hard Error</td>
<td>This bit is asserted in IA32_MCI_STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>IERR</td>
<td>This bit is asserted in IA32_MCI_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.</td>
<td></td>
</tr>
</tbody>
</table>
Table 17-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes for Machine Check

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>AERR</td>
<td>This bit is asserted in IA32_MCi_STATUS if this component has initiated 2 failing bus transactions which have failed due to Address Parity Errors AERR asserted. While this bit is asserted, it cannot be overwritten.</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>UECC</td>
<td>The Uncorrectable ECC error bit is asserted in IA32_MCi_STATUS for uncorrected ECC errors. While this bit is asserted, the ECC syndrome field will not be overwritten.</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>CECC</td>
<td>The correctable ECC error bit is asserted in IA32_MCi_STATUS for corrected ECC errors.</td>
<td></td>
</tr>
<tr>
<td>54:47</td>
<td>ECC Syndrome</td>
<td>The ECC syndrome field in IA32_MCi_STATUS contains the 8-bit ECC syndrome only if the error was a correctable/uncorrectable ECC error and there wasn't a previous valid ECC error syndrome logged in IA32_MCi_STATUS. A previous valid ECC error in IA32_MCi_STATUS is indicated by IA32_MCi_STATUS.bit45 uncorrectable error occurred) being asserted. After processing an ECC error, machine check handling software should clear IA32_MCi_STATUS.bit45 so that future ECC error syndromes can be logged.</td>
<td></td>
</tr>
<tr>
<td>56:55</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Status Register Validity Indicators

<table>
<thead>
<tr>
<th>Status Register Validity Indicators¹</th>
<th>63:57</th>
</tr>
</thead>
</table>

NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
2. For processors with a CPUID signature of 06_0EH, a ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit.
3. For processors with a CPUID signature of 6_06_60H and later, the PIC timer will count crystal clock cycles.

17.2 INCREMENTAL DECODING INFORMATION: INTEL® CORE™ 2 PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-4 provides information for interpreting additional model-specific fields for external bus errors relating to processors based on Intel® Core™ microarchitecture, which implements the P4 bus specification. Table 17-3 lists the CPUID signatures for Intel 64 processors that are covered by Table 17-4. These errors are reported in the IA32_MCi_STATUS MSRs. They are reported architecturally as compound errors with a general form of 0000 1PPT RRRR IILL in the MCA error code field. See Chapter 16 for information on the interpretation of compound error codes.

Table 17-3. CPUID DisplayFamily_DisplayModel Signatures for Processors Based on Intel® Core™ Microarchitecture

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Processor Families/Processor Number Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_1DH</td>
<td>Intel® Xeon® Processor 7400 series</td>
</tr>
<tr>
<td>06_17H</td>
<td>Intel® Xeon® Processor 5200, 5400 series, Intel® Core™ 2 Quad processor Q9650</td>
</tr>
<tr>
<td>06_0FH</td>
<td>Intel® Xeon® Processor 3000, 3200, 5100, 5300, 7300 series, Intel® Core™ 2 Quad, Intel® Core™ 2 Extreme, Intel® Core™ 2 Duo processors, Intel Pentium dual-core processors</td>
</tr>
</tbody>
</table>
# Table 17-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel® Core™ Microarchitecture

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>18:16</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>27:25 Bus Queue Error Type</td>
<td></td>
<td>'001: Address Parity Error. '010: Response Hard Error. '011: Response Parity Error.</td>
<td></td>
</tr>
<tr>
<td>28 MCE Driven</td>
<td></td>
<td></td>
<td>1 if MCE is driven.</td>
</tr>
<tr>
<td>29 MCE Observed</td>
<td></td>
<td></td>
<td>1 if MCE is observed.</td>
</tr>
<tr>
<td>30 Internal BINIT</td>
<td></td>
<td></td>
<td>1 if BINIT driven for this processor.</td>
</tr>
<tr>
<td>31 BINIT Observed</td>
<td></td>
<td></td>
<td>1 if BINIT is observed for this processor.</td>
</tr>
<tr>
<td>Other Information</td>
<td>33:32</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>34 PIC and FSB Data Parity</td>
<td></td>
<td>Data Parity detected on either PIC or FSB access.</td>
<td></td>
</tr>
<tr>
<td>35 Reserved</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>36 Response Parity Error</td>
<td></td>
<td>This bit is asserted in IA32_MCR/STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.</td>
<td></td>
</tr>
</tbody>
</table>
17.2.1 Model-Specific Machine Check Error Codes for Intel® Xeon® Processor 7400 Series

The Intel® Xeon® processor 7400 series has machine check register banks that generally follow the description of Chapter 16 and Section 17.2. Additional error codes specific to the Intel Xeon processor 7400 series are described in this section.

MC4\_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors for the Intel Xeon processor 7400 series. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>FSB Address Parity</td>
<td>Address parity error detected: 1: Address parity error detected. 0: No address parity error.</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Timeout BINIT</td>
<td>This bit is asserted in IA32_MC_STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 23-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.</td>
<td></td>
</tr>
<tr>
<td>41:39</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Hard Error</td>
<td>This bit is asserted in IA32_MC_STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>IERR</td>
<td>This bit is asserted in IA32_MC_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>54:47</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>56:55</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Status Register Validity Indicators(^1)</td>
<td>63:57</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.
17.2.1.1 Processor Machine Check Status Register, Incremental MCA Error Code Definition

The Intel Xeon processor 7400 series uses compound MCA Error Codes for logging its Bus internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines incremental Machine Check error types (IA32_MC6_STATUS[15:0]) beyond those defined in Chapter 16. Table 17-5 lists these incremental MCA error code types that apply to IA32_MC6_STATUS. Error code details are specified in MC6_STATUS [31:16] (see Section 17.2.2), the “Model Specific Error Code” field. The information in the “Other_Info” field (MC4_STATUS[56:32]) is common to the three processor error types. It contains a correctable event count and specifies the MC6_MISC register format.

<table>
<thead>
<tr>
<th>Type</th>
<th>Error Code</th>
<th>Binary Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Internal Error</td>
<td>0000 0100 0000 0000</td>
<td>Internal Error Type Code.</td>
</tr>
<tr>
<td>B</td>
<td>Bus and Interconnect Error</td>
<td>0000 100x 0000 1111</td>
<td>Not used but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 101x 0000 1111</td>
<td>Not used but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 110x 0000 1111</td>
<td>Not used but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 1110 0000 1111</td>
<td>Bus and Interconnection Error Type Code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 1111 0000 1111</td>
<td>Not used but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
</tbody>
</table>

The **Bold faced** binary encodings are the only encodings used by the processor for MC4_STATUS[15:0].

17.2.2 Intel® Xeon® Processor 7400 Model Specific Error Code Field

17.2.2.1 Processor Model Specific Error Code Field, Type B: Bus and Interconnect Error Codes

The Model Specific Error Code field in MC6_STATUS (bits 31:16) is defined in Table 17-6.

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Sub-Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>FSB Request Parity</td>
<td>Parity error detected during FSB request phase.</td>
</tr>
<tr>
<td>19:17</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>FSB Hard Fail Response</td>
<td>“Hard Failure” response received for a local transaction.</td>
</tr>
<tr>
<td>21</td>
<td>FSB Response Parity</td>
<td>Parity error on FSB response field detected.</td>
</tr>
<tr>
<td>22</td>
<td>FSB Data Parity</td>
<td>FSB data parity error on inbound data detected.</td>
</tr>
<tr>
<td>31:23</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
17.2.2.2 Processor Model Specific Error Code Field, Type C: Cache Bus Controller Error Codes

Table 17-7. Type C: Cache Bus Controller Error Codes

<table>
<thead>
<tr>
<th>MC4_STATUS[31:16] (MSCE) Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_0000_0000_0001 0001H</td>
<td>Inclusion Error from Core 0.</td>
</tr>
<tr>
<td>0000_0000_0000_0010 0002H</td>
<td>Inclusion Error from Core 1.</td>
</tr>
<tr>
<td>0000_0000_0000_0011 0003H</td>
<td>Write Exclusive Error from Core 0.</td>
</tr>
<tr>
<td>0000_0000_0000_0100 0004H</td>
<td>Write Exclusive Error from Core 1.</td>
</tr>
<tr>
<td>0000_0000_0000_0101 0005H</td>
<td>Inclusion Error from FSB.</td>
</tr>
<tr>
<td>0000_0000_0000_0110 0006H</td>
<td>SNP Stall Error from FSB.</td>
</tr>
<tr>
<td>0000_0000_0000_0111 0007H</td>
<td>Write Stall Error from FSB.</td>
</tr>
<tr>
<td>0000_0000_0000_1000 0008H</td>
<td>FSB Arb Timeout Error.</td>
</tr>
<tr>
<td>0000_0000_0000_1010 000AH</td>
<td>Inclusion Error from Core 2.</td>
</tr>
<tr>
<td>0000_0000_0000_1011 000BH</td>
<td>Write Exclusive Error from Core 2.</td>
</tr>
<tr>
<td>0000_0010_0000_0000 0200H</td>
<td>Internal Timeout Error.</td>
</tr>
<tr>
<td>0000_0011_0000_0000 0300H</td>
<td>Internal Timeout Error.</td>
</tr>
<tr>
<td>0000_0100_0000_0000 0400H</td>
<td>Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow.</td>
</tr>
<tr>
<td>0000_0101_0000_0000 0500H</td>
<td>Quiet cycle Timeout Error (correctable).</td>
</tr>
<tr>
<td>1100_0000_0000_0010 C002H</td>
<td>Correctable ECC event on outgoing Core 0 data.</td>
</tr>
<tr>
<td>1100_0000_0000_0100 C004H</td>
<td>Correctable ECC event on outgoing Core 1 data.</td>
</tr>
<tr>
<td>1100_0000_0000_1000 C008H</td>
<td>Correctable ECC event on outgoing Core 2 data.</td>
</tr>
<tr>
<td>1110_0000_0000_0010 E002H</td>
<td>Uncorrectable ECC error on outgoing Core 0 data.</td>
</tr>
<tr>
<td>1110_0000_0000_0100 E004H</td>
<td>Uncorrectable ECC error on outgoing Core 1 data.</td>
</tr>
<tr>
<td>1110_0000_0000_1000 E008H</td>
<td>Uncorrectable ECC error on outgoing Core 2 data.</td>
</tr>
<tr>
<td>— All other encodings —</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

17.3 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR 3400, 3500, 5500 SERIES, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-8 through Table 17-12 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor 3400, 3500, 5500 series with CPUID DisplayFamily_DisplaySignature 06_1AH, which supports Intel® QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC0 and IA32_MC1, incremental error codes for internal machine check are reported in the register bank IA32_MC7, and incremental error codes for the memory controller unit are reported in the register bank IA32_MC8.
17.3.1 Intel® QPI Machine Check Errors

Table 17-8. Intel® QPI Machine Check Error Codes for IA32_MC0_STATUS and IA32_MC1_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes(^1)</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRIILL</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>16</td>
<td>Header Parity</td>
<td>If 1, QPI Header had bad parity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Data Parity</td>
<td>If 1, QPI Data packet had bad parity.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Retries Exceeded</td>
<td>If 1, the number of QPI retries was exceeded.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Received Poison</td>
<td>If 1, received a data packet that was marked as poisoned by the sender.</td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Unsupported Message</td>
<td>If 1, QPI received a message encoding it does not support.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Unsupported Credit</td>
<td>If 1, QPI credit type is not supported.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Receive Flit Overrun</td>
<td>If 1, sender sent too many QPI flits to the receiver.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Received Failed Response</td>
<td>If 1, indicates that sender sent a failed response to receiver.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Receiver Clock Jitter</td>
<td>If 1, clock jitter detected in the internal QPI clocking.</td>
<td></td>
</tr>
<tr>
<td>56:27</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Register</td>
<td>63:57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validity Indicators(^1)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Table 17-9. Intel® QPI Machine Check Error Codes for IA32_MC0_MISC and IA32_MC1_MISC

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Specific Errors(^1)</td>
<td>7:0</td>
<td>QPI Opcode</td>
<td>Message class and opcode from the packet with the error.</td>
</tr>
<tr>
<td>13:8</td>
<td>RTID</td>
<td>QPI Request Transaction ID.</td>
<td></td>
</tr>
<tr>
<td>15:14</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>18:16</td>
<td>RHNID</td>
<td>QPI Requestor/Home Node ID.</td>
<td></td>
</tr>
<tr>
<td>23:19</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>24</td>
<td>IIB</td>
<td>QPI Interleave/Head Indication Bit.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. Which of these fields are valid depends on the error type.
17.3.2 Internal Machine Check Errors

Table 17-10. Machine Check Error Codes for IA32_MC7_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Reserved</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>23:16</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>31:24</td>
<td>Reserved, except for the following</td>
<td>00H: No error. 03H: Reset firmware did not complete. 08H: Received an invalid CMPD. 0AH: Invalid Power Management Request. 0DH: Invalid S-state transition. 11H: VID controller does not match POC controller selected. 1AH: MSID from POC does not match CPU MSID.</td>
</tr>
<tr>
<td>Status Register</td>
<td>56:32</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.3.3 Memory Controller Errors

Table 17-11. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory error format: 1MMMCCCC</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>16</td>
<td>Read ECC Error</td>
<td>If 1, ECC occurred on a read.</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>RAS ECC Error</td>
<td>If 1, ECC occurred on a scrub.</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Write Parity Error</td>
<td>If 1, bad parity on a write.</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>Redundancy Loss</td>
<td>If 1, error in half of redundant memory.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>Memory Range Error</td>
<td>If 1, memory access out of range.</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>RTID Out of Range</td>
<td>If 1, Internal ID invalid.</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Address Parity Error</td>
<td>If 1, bad address parity.</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>Byte Enable Parity Error</td>
<td>If 1, bad enable parity.</td>
</tr>
<tr>
<td>Other Information</td>
<td>37:25</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>52:38</td>
<td>CORE_ERR_CNT</td>
<td>Corrected error count.</td>
</tr>
<tr>
<td></td>
<td>56:53</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Status Register</td>
<td>63:57</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
17.4 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-13 through Table 17-15 provide information for interpreting additional model-specific fields for memory controller errors relating to the Intel® Xeon® processor E5 Family with CPUID DisplayFamily_DisplaySignature 06_2DH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC6 and IA32_MC7, incremental error codes for internal machine check error from PCU controller are reported in the register bank IA32_MC4, and incremental error codes for the memory controller unit are reported in the register banks IA32_MC8—IA32_MC11.

### 17.4.1 Internal Machine Check Errors

#### Table 17-13. Machine Check Error Codes for IA32_MC4_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>0000b: No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0001b: Non.IMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010b: I_Parity_Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011b: Bad_OpCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100b: I_Stack_Underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0101b: I_Stack_Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0110b: D_Stack_Underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0111b: D_Stack_Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000b: Non-DMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1001b: D_Parity_Error</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>19:16</td>
<td>Reserved, except for the following</td>
<td>0000b: No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0001b: Non.IMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010b: I_Parity_Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011b: Bad_OpCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100b: I_Stack_Underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0101b: I_Stack_Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0110b: D_Stack_Underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0111b: D_Stack_Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000b: Non-DMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1001b: D_Parity_Error</td>
</tr>
</tbody>
</table>
17.4.2 Intel® QPI Machine Check Errors

Table 17-14. Intel® QPI MC Error Codes for IA32_MC6_STATUS and IA32_MC7_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes1</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRRILL</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>56:16</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Status Register Validity Indicators1</td>
<td>63:57</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.4.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC8_STATUS–IA32_MC11_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMMCCCC; see Chapter 16, “Machine-Check Architecture.” MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in the IA32_MCi_STATUS and IA32_MCi_MISC, where i = 8, 11.
Table 17-15. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 8, 11)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes(^1)</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRRILL</td>
</tr>
<tr>
<td>Other Info</td>
<td>36:32</td>
<td>Reserved</td>
<td>When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first device error when corrected error is detected during normal read.</td>
</tr>
<tr>
<td>Valid_1stErrDev</td>
<td>63:57</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Table 17-16. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 8, 11)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Addr Info(^1)</td>
<td>8:0</td>
<td></td>
<td>See Chapter 16, “Machine-Check Architecture.”</td>
</tr>
</tbody>
</table>
| Model Specific Errors       | 13:9    | ErrMask_1stErrDev | • When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second device error when corrected error is detected during normal read.
• Otherwise, contains parity error if MCi_Status indicates HA_WB_Data or HA_W_BE parity error. |
| ErrMask_2ndErrDev           | 29:14   |              | When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error bit mask. |
| FailRank_1stErrDev          | 50:46   |              | When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error failing rank. |
| FailRank_2ndErrDev          | 55:51   |              | When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error failing rank. |
| Valid_1stErrDev             | 58:56   |              | Reserved                                                                    |
| Valid_2ndErrDev             | 61:59   |              | Reserved                                                                    |
| Valid_1stErrDev             | 62      |              | When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data from the first correctable error in a memory device. |
| Valid_2ndErrDev             | 63      |              | When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error information indicated by bit 62. |

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
17.5  INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V2 AND INTEL® XEON® PROCESSOR E7 V2 FAMILIES, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v2 family and the Intel® Xeon® processor E7 v2 family are based on the Ivy Bridge-EP microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_3EH. Incremental error codes for internal machine check error from the PCU controller is reported in the register bank IA32_MC4; Table 17-17 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register bank IA32_MC5. Information listed in Table 17-14 for QPI MC error codes apply to IA32_MC5_STATUS. Incremental error codes for the memory controller unit are reported in the register banks IA32_MC9–IA32_MC16. Table 17-18 lists model-specific error codes that apply to IA32_MCI_STATUS, where i = 9-16.

17.5.1  Internal Machine Check Errors

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes1</td>
<td>15:0</td>
<td>MCACOD</td>
<td>0000b: No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0001b: Non_JMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010b: I_Parity_Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011b: Bad_OpCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100b: I_Stack_Underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0101b: I_Stack_Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0110b: D_Stack_Underflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0111b: D_Stack_Overflow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000b: Non-DMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1001b: D_Parity_Error</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>19:16</td>
<td>Reserved, except for the following</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0000b: No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010b: Non_JMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011b: Bad_OpCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000b: Non-DMem_Sel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:24</td>
<td>Reserved, except for the following</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00H: No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>06H: MC_ICM_FORCE_SR_S3_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>06H: MC_CPD_UNCPD_ST_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0FH: MC_PKGS_SAFE_WP_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>43H: MC_PECI_MAILBOX_QUIESCE_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>44H: MC_CRITICAL_VR_FAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>45H: MC_ICC_MAX-NOTSUPPORTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5CH: MC_MORE_THAN_ONE_LT_AGENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>60H: MC_INVALID_PKGS_REQ_PCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>61H: MC_INVALID_PKGS_REQ_QPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>62H: MC_INVALID_PKGS_RES_QPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>63H: MC_INVALID_PKGS_RES_PCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64H: MC_INVALID_PKG_STATE_CONFIG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>70H: MC_WATCHDOG_TIMEOUT_PKGC_SECONDARY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>71H: MC_WATCHDOG_TIMEOUT_PKGC_MAIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>72H: MC_WATCHDOG_TIMEOUT_PKGS_MAIN</td>
</tr>
</tbody>
</table>
17.5.2  Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC16_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, "Machine-Check Architecture."

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC, where i = 9—16.

IA32_MCi_STATUS (i=9—12) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCi_STATUS (i=13—16).

### Table 17-17.  Machine Check Error Codes for IA32_MC4_STATUS  (Contd.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 000F 0000 1MMM CCCC</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>31:16</td>
<td>Reserved, except for the following</td>
<td>001H: Address parity error. 002H: HA Wrt buffer data parity error. 004H: HA Wrt byte enable parity error. 008H: Corrected patrol scrub error. 010H: Uncorrected patrol scrub error. 020H: Corrected spare error. 040H: Uncorrected spare error. 080H: Corrected memory read error. (Only applicable with IMC's &quot;Additional Error logging&quot; Mode-1 enabled.) 100H - IMC, WDB, parity errors</td>
</tr>
<tr>
<td>Status Register Validity Indicators 1</td>
<td>63:57</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>56:32</td>
<td>Reserved</td>
<td>Reserved</td>
<td>See Chapter 16, &quot;Machine-Check Architecture.&quot;</td>
</tr>
</tbody>
</table>
17.5.3 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors to the IA32_MC8_{STATUS,ADDR,MISC} registers.

### INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V3 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor E5 v3 family is based on the Haswell-E microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_3FH. Incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-20 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5, IA32_MC20, and IA32_MC21. Table 17-21 contains information for QPI MC error codes. Incremental error codes for the memory controller unit are reported in the register banks IA32_MC9–IA32_MC16. Table 17-22 lists model-specific error codes that apply to IA32_MCI_STATUS, where i = 9–16.
## 17.6.1 Internal Machine Check Errors

### Table 17-20. Machine Check Error Codes for IA32_MC4_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>0402H: PCU internal errors. 0403H: PCU internal errors. 0406H: Intel TXT errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0407H: Other UBOX internal errors. An IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.</td>
</tr>
<tr>
<td>MCACOD</td>
<td>15:0</td>
<td>Internal Errors</td>
<td>0402H: PCU internal errors. 0403H: PCU internal errors. 0406H: Intel TXT errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0407H: Other UBOX internal errors. An IERR caused by a core 3-strike the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>19:16</td>
<td>Reserved, except for the following</td>
<td>0000b: No error. 00xxb: PCU internal error.</td>
</tr>
<tr>
<td>Status Register</td>
<td>56:32</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Validity Indicators</td>
<td>63:57</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
2. The internal error codes may be model-specific.
17.6.2 Intel® QPI Machine Check Errors

MC error codes associated with the Intel QPI agents are reported in the IA32_MC5_STATUS, IA32_MC20_STATUS, and IA32_MC21_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1PPTRRRIILL; see Chapter 16, "Machine-Check Architecture."

Table 17-21 lists model-specific fields to interpret error codes applicable to IA32_MC5_STATUS, IA32_MC20_STATUS, and IA32_MC21_STATUS.

### Table 17-21. Intel® QPI MC Error Codes for IA32_MCi_STATUS (i = 5, 20, 21)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes(^1)</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRIILL</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>31:16</td>
<td>MSCOD</td>
<td>02H: Intel QPI physical layer detected drift buffer alarm.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>03H: Intel QPI physical layer detected latency buffer rollover.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10H: Intel QPI link layer detected control error from R3QPI.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11H: Rx entered LLR abort state on CRC error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12H: Unsupported or undefined packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13H: Intel QPI link layer control error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15H: RBT used un-initialized value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20H: Intel QPI physical layer detected a QPI in-band reset but aborted initialization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21H: Link failover data self-healing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22H: Phy detected in-band reset (no width change).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23H: Link failover clock failover.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30H: Rx detected CRC error; successful LLR after Phy re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31H: Rx detected CRC error; successful LLR without Phy re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
</tbody>
</table>

|            | 37:32 | Reserved    | Reserved                          |
|            | 52:38 | Corrected Error Cnt |                              |
| Status Register Validity Indicators\(^1\) | 56:53 | Reserved | Reserved |

\(^1\) These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

17.6.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS–IA32_MC16_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMMCCC; see Chapter 16, "Machine-Check Architecture."

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCI_STATUS and IA32_MCI_MISC, where i = 9–16.

IA32_MCI_STATUS (i=9–12) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCI_STATUS (i=13–16).
### Table 17-22. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MCA Error Codes</strong></td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 0000 0000 1MMM CCCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36:32 Other Info</td>
<td>When MSR_ERROR_CONTROL[1] is set, logs an encoded value from the first error device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>56:38</td>
<td>See Chapter 16, “Machine-Check Architecture.”</td>
</tr>
<tr>
<td><strong>Status Register</strong></td>
<td>63:57</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Validity Indicators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

### Table 17-23. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9—16)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MCA Addr Info</strong></td>
<td>8:0</td>
<td></td>
<td>See Chapter 16, “Machine-Check Architecture.”</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>13:9</td>
<td></td>
<td>If the error logged is an MCWrDataPar error or an MCWrBEPar error, this field is the WDB ID that has the parity error; OR if the second error logged is a correctable read error, MC logs the second error device in this field.</td>
</tr>
<tr>
<td></td>
<td>29:14</td>
<td>ErrMask_1stErrDev</td>
<td>When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error bit mask.</td>
</tr>
<tr>
<td></td>
<td>45:30</td>
<td>ErrMask_2ndErrDev</td>
<td>When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error bit mask.</td>
</tr>
<tr>
<td></td>
<td>50:46</td>
<td>FailRank_1stErrDev</td>
<td>When MSR_ERROR_CONTROL[1] is set, allows the iMC to log first-device error failing rank.</td>
</tr>
<tr>
<td></td>
<td>55:51</td>
<td>FailRank_2ndErrDev</td>
<td>When MSR_ERROR_CONTROL[1] is set, allows the iMC to log second-device error failing rank.</td>
</tr>
<tr>
<td></td>
<td>61:56</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>Valid_1stErrDev</td>
<td>When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data from a correctable error from a memory read associated with first error device.</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>Valid_2ndErrDev</td>
<td>When MSR_ERROR_CONTROL[1] is set, indicates the iMC has logged valid data due to a second correctable error in a memory device. Use this information only after there is valid first error information indicated by bit 62.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
17.6.4 Home Agent Machine Check Errors

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors in the IA32_MC8_{STATUS,ADDR,MISC} registers.

17.7 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR D FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

The Intel® Xeon® processor D family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_56H. Incremental error codes for internal machine check error from the PCU controller are reported in the register bank IA32_MC4. Table 17-24 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS. Incremental error codes for the memory controller unit are reported in the register banks IA32_MC9−IA32_MC10. Table 17-18 lists model-specific error codes that apply to IA32_M-Ci_STATUS, where i = 9—10.

17.7.1 Internal Machine Check Errors

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>0402H: PCU internal errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0403H: Internal errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0406H: Intel TXT errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0407H: Other UBOX internal errors.</td>
</tr>
<tr>
<td>MCACOD²</td>
<td>15:0</td>
<td>Internal Errors</td>
<td>On an IERR caused by a core 3-strike, the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>19:16</td>
<td>Reserved, except for the following</td>
<td>0000b: No error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00x1b: PCU internal error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>001xb: PCU internal error.</td>
</tr>
<tr>
<td></td>
<td>23:20</td>
<td>Reserved, except for the following</td>
<td>x1xxb: UBOX error.</td>
</tr>
<tr>
<td></td>
<td>31:24</td>
<td>Reserved, except for the following</td>
<td>00H: No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>09H: MC_MESSAGE_CHANNEL_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13H: MC_DMI_TRAINING_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15H: MC_DMI_CPU_RESET_ACK_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1EH: MC_VR_ICC_MAX_LT_FUSED_ICC_MAX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25H: MC_SVID_COMMAND_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26H: MCA_PKGC_DIRECT_WAKE_RING_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>29H: MC_VOUT_MACRO_LT_FUSED_SVID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2BH: MC_PKGC_WATCHDOG_HANG_CBZ_DOWN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2CH: MC_PKGC_WATCHDOG_HANG_CBZ_UP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>44H: MC_CRITICAL_VR FAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>46H: MC_VID_RAMP_DOWN_FAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>49H: MC_SVID_WRITE_REG_VOUT_MAX_FAILED</td>
</tr>
</tbody>
</table>
**17.7.2 Integrated Memory Controller Machine Check Errors**

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC10_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type $1\text{MMMCCCC}$; see Chapter 16, “Machine-Check Architecture.”

MSR_ERROR_CONTROL.[bit 1] can enable additional information logging of the IMC. The additional error information logged by the IMC is stored in IA32_MCi_STATUS and IA32_MCi_MISC, where $i = 9–10$.

### Table 17-24. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 0000 0000 1\text{MMMCCCC}</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>31:16</td>
<td>Reserved, except for the following</td>
<td>0001H: DDR3 address parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0002H: Uncorrected HA write data error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0004H: Uncorrected HA data byte enable error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008H: Corrected patrol scrub error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010H: Uncorrected patrol scrub error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100H: IMC, write data buffer parity errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0200H: DDR4 command address parity error.</td>
</tr>
<tr>
<td>Status Register</td>
<td>56:32</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Validity Indicators</td>
<td>63:57</td>
<td></td>
<td>56:32</td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
2. The internal error codes may be model-specific.

### Table 17-25. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9–10)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 0000 0000 1\text{MMMCCCC}</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>31:16</td>
<td>Reserved, except for the following</td>
<td>0001H: DDR3 address parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0002H: Uncorrected HA write data error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0004H: Uncorrected HA data byte enable error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008H: Corrected patrol scrub error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010H: Uncorrected patrol scrub error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100H: IMC, write data buffer parity errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0200H: DDR4 command address parity error.</td>
</tr>
<tr>
<td>Status Register</td>
<td>56:38</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Validity Indicators</td>
<td>63:57</td>
<td></td>
<td>56:38</td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
2. The internal error codes may be model-specific.
17.8 **INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V4 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK**

The Intel® Xeon® processor E5 v4 family is based on the Broadwell microarchitecture and can be identified with CPUID DisplayFamily_DisplaySignature 06_4FH. Incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-20 in Section 17.6.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC5, IA32_MC20, and IA32_MC21. Information listed in Table 17-21 of Section 17.6.1 covers QPI MC error codes.

### 17.8.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC9_STATUS—IA32_MC16_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, “Machine-Check Architecture.”

Table 17-26 lists model-specific error codes that apply to IA32_MCI_STATUS, where i = 9–16.

IA32_MCI_STATUS (i=9–12) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCI_STATUS (i=13–16).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 0000 0000 1MMM CCCC</td>
</tr>
</tbody>
</table>
| Model Specific Errors | 31:16  | Reserved, except for the following | 0001H: DDR3 address parity error.  
|                    |         |                         | 0002H: Uncorrected HA write data error.  
|                    |         |                         | 0004H: Uncorrected HA data byte enable error.  
|                    |         |                         | 0008H: Corrected patrol scrub error.  
|                    |         |                         | 0010H: Uncorrected patrol scrub error.  
|                    |         |                         | 0020H: Corrected spare error.  
|                    |         |                         | 0040H: Uncorrected spare error.  
|                    |         |                         | 0100H: iMC, write data buffer parity errors.  
|                    |         |                         | 0200H: DDR4 command address parity error.  |
| 36:32              | Other Info | Reserved               |                                                      |
| 37                 | Reserved   | Reserved               |                                                      |
| 56:38              | See Chapter 16, “Machine-Check Architecture.” |                                                      |

### 17.8.2 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the IA32_MC7_MISC and IA32_MC8_MISC MSRs. Table 17-27 lists model-specific error codes that apply to IA32_MCI_MISC, where i = 7, 8.

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors in the IA32_MC8_{STATUS,ADDR,MISC} registers.

### Table 17-27. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8)

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:0</td>
<td>LSB</td>
<td>See Figure 16-8.</td>
</tr>
<tr>
<td>8:6</td>
<td>Address Mode</td>
<td>See Table 16-3.</td>
</tr>
<tr>
<td>40:9</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>41</td>
<td>Failover</td>
<td>Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.</td>
</tr>
<tr>
<td>42</td>
<td>Mirrorcorr</td>
<td>Error was corrected by mirroring and primary channel scrubbed successfully.</td>
</tr>
<tr>
<td>63:43</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### 17.9 INCREMENTAL DECODING INFORMATION: INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the Intel® Xeon® Scalable Processor Family with CPUID DisplayFamily_DisplaySignature 06_55H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-28 in Section 17.9.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

### 17.9.1 Internal Machine Check Errors

#### Table 17-28. Machine Check Error Codes for IA32_MC4_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>0402H: PCU internal errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0403H: PCU internal errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0406H: Intel TXT errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0407H: Other UBOX internal errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>On an IERR caused by a core 3-strike, the IA32_MC3_STATUS (MLC) is copied to the IA32_MC4_STATUS. After a 3-strike, the core MCA banks will be unavailable.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>19:16</td>
<td>Reserved, except for the following</td>
<td>0000b: No error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00xxb: PCU internal error.</td>
</tr>
</tbody>
</table>
### Table 17-28. Machine Check Error Codes for IA32_MC4_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>23:20</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>52:32</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>54:53</td>
<td>CORR_ERR_STATUS</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
INTERPRETING MACHINE CHECK ERROR CODES

17.9.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the IA32_MC5_STATUS, IA32_MC12_STATUS, and IA32_MC19_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL; see Chapter 16, "Machine-Check Architecture."

Table 17-29 lists model-specific fields to interpret error codes applicable to IA32_MCi_STATUS, i = 5, 12, 19.

Table 17-29. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 12, 19)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes1</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRRIILL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The two supported compound error codes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0x0C0F: Unsupported/Undefined Packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0x0E0F: For all other corrected and uncorrected errors.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>21:16</td>
<td>MSCOD</td>
<td>The encoding of Uncorrectable (UC) errors are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00H: UC Phy Initialization Failure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01H: UC Phy detected drift buffer alarm.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>02H: UC Phy detected latency buffer rollover.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10H: UC link layer Rx detected CRC error; unsuccessful LLR entered abort state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11H: UC LL Rx unsupported or undefined packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12H: UC LL or Phy control error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13H: UC LL Rx parameter exchange exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1FH: UC LL detected control error from the link-mesh interface.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The encoding of correctable (COR) errors are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20H: COR Phy initialization abort.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21H: COR Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22H: COR Phy lane failure, recovery in x8 width.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23H: COR Phy L0c error corrected without Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24H: COR Phy L0c error triggering Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30H: COR LL Rx detected CRC error; successful LLR without Phy re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31H: COR LL Rx detected CRC error; successful LLR with Phy re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
</tbody>
</table>

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.
2. The internal error codes may be model-specific.
17.9.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC13_STATUS–IA32_MC18_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, “Machine-Check Architecture.”

IA32_MCI_STATUS (i=13,14,17) logs errors from the first memory controller. The second memory controller logs errors into IA32_MCI_STATUS (i=15,16,18).

### Table 17-29. Interconnect MC Error Codes for IA32_MCI_STATUS (i = 5, 12, 19)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSCOD_SPARE</td>
<td>31:22</td>
<td></td>
<td>The definition below applies to MSCOD 12h (UC LL or Phy Control Errors)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 22]: Phy Control Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 23]: Unexpected Retry.Ack flit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 24]: Unexpected Retry.Req flit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 25]: RF parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 26]: Routeback Table error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 27]: Unexpected Tx Protocol flit (EOP, Header or Data).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 28]: Rx Header-or-Credit BGF credit overflow/underflow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 29]: Link Layer Reset still in progress when Phy enters L0 (Phy training should not be enabled until after LL reset is complete as indicated by KTILCL:LinkLayerReset going back to 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 30]: Link Layer reset initiated while protocol traffic not idle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 31]: Link Layer Tx Parity Error.</td>
</tr>
<tr>
<td>Status Register</td>
<td>37:32</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Validity Indicators¹</td>
<td>52:38</td>
<td>Corrected Error Cnt</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>56:53</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>63:57</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
17.9.4 M2M Machine Check Errors

MC error codes associated with M2M are reported in the IA32_MC7_STATUS and IA32_MC8_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, “Machine-Check Architecture.”
17.9.5 Home Agent Machine Check Errors

MC error codes associated with mirrored memory corrections are reported in the IA32_MC7_MISC and IA32_MC8_MISC MSRs. Table 17-32 lists model-specific error codes that apply to IA32_MCi_MISC, where i = 7, 8.

Memory errors from the first memory controller may be logged in the IA32_MC7_{STATUS,ADDR,MISC} registers, while the second memory controller logs errors in the IA32_MC8_{STATUS,ADDR,MISC} registers.

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:0</td>
<td>LSB</td>
<td>See Figure 16-8.</td>
</tr>
<tr>
<td>8:6</td>
<td>Address Mode</td>
<td>See Table 16-3.</td>
</tr>
<tr>
<td>40:9</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>61:41</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>62</td>
<td>Mirrorcorr</td>
<td>Error was corrected by mirroring and primary channel scrubbed successfully.</td>
</tr>
<tr>
<td>63</td>
<td>Failover</td>
<td>Error occurred at a pair of mirrored memory channels. Error was corrected by mirroring with channel failover.</td>
</tr>
</tbody>
</table>
17.10 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_5FH, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel Atom® processors based on Goldmont Microarchitecture with CPUID DisplayFamily_DisplaySignature 06_5FH (Denverton), incremental error codes for the memory controller unit are reported in the register banks IA32_MC6 and IA32_MC7. Table 17-33 in Section 17.10.1 lists model-specific fields to interpret error codes applicable to IA32_MCI_STATUS, where i = 6, 7.

17.10.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the IA32_MC6_STATUS and IA32_MC7_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, "Machine-Check Architecture."

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td></td>
</tr>
</tbody>
</table>
| Model Specific Errors | 31:16   | Reserved, except for the following | 01H: Cmd/Addr parity.  
|                       |         |              | 02H: Corrected Demand/Patrol Scrub error.  
|                       |         |              | 04H: Uncorrected patrol scrub error.  
|                       |         |              | 08H: Uncorrected demand read error.  
|                       |         |              | 10H: WDB read ECC.  |
| Status Register       | 36:32   | Other Info   |                                      |
| Validity Indicators   | 37      | Reserved     | Reserved                             |
|                       | 56:38   | See Chapter 16, "Machine-Check Architecture."

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, "Machine-Check Architecture," for more information.

17.11 INCREMENTAL DECODING INFORMATION: 3RD GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the 3rd generation Intel® Xeon® Scalable Processor Family with CPUID DisplayFamily_DisplaySignatures of 06_6AH and 06_6CH, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-34 in Section 17.11.1 lists model-specific fields to interpret error codes applicable to IA32_MCI_STATUS.
### 17.11.1 Internal Machine Check Errors

#### Table 17-34. Machine Check Error Codes for IA32_MC4_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Check Error Codes</td>
<td>15:0</td>
<td>MCCOD</td>
<td>The value of this field will be 0402H for the PCU and 0406H for internal firmware errors. This applies for any logged error.</td>
</tr>
<tr>
<td>MCCOD</td>
<td>15:0</td>
<td>Internal Errors</td>
<td></td>
</tr>
</tbody>
</table>
| Model Specific Errors       | 19:16   | Reserved, except for the following | Model specific error code bits 19:16. This logs the type of HW UC (PCU/VCU) error that has occurred. There are 7 errors defined.  
01H: Instruction address out of valid space.  
02H: Double bit RAM error on Instruction Fetch.  
04H: Stack Underflow.  
05H: Stack Overflow.  
06H: Data address out of valid space.  
07H: Double bit RAM error on Data Fetch. |
| 23:20                       | Reserved, except for the following | Model specific error code bits 23:20. This logs the type of HW FSM error that has occurred. There are 3 errors defined.  
04H: Clock/power IP response timeout.  
05H: SMBus controller raised SMI.  
09H: PM controller received invalid transaction. |
| 31:24                       | Reserved, except for the following | Model specific error code bits 31:24.                                                                                                          |
|                             | 0DH     | MCA_LLCC_BIST_ACTIVE_TIMEOUT |  
06H: MCA_DMI_TRAINING_TIMEOUT  
0FH: MCA_DMI_STRAP_SET.ARRIVAL_TIMEOUT  
10H: MCA_DMI_CPU_RESET_ACK_TIMEOUT  
11H: MCA_MORE_THAN_ONE_LT_AGENT  
14H: MCA_INCOMPATIBLE_PCH_TYPE  
1EH: MCA_BIOS_RST_CPL_INVALID_SEQ  
1FH: MCA_BIOS_INVALID_PKG_STATE_CONFIG  
2DH: MCA_PCU_PMAX_CALIB_ERROR  
2EH: MCA_TSC100_SYNC_TIMEOUT  
3AH: MCA_GPSB_TIMEOUT  
3BH: MCA_PMSB_TIMEOUT  
3EH: MCA_IOSFSB_PIMREQ_CMP_TIMEOUT  
40H: MCA_SVID_VCCIN VR_ICC_MAX_FAILURE  
42H: MCA_SVID_VCCIN VR_VOUT_FAILURE  
43H: MCA_SVID_CPU VR_CAPABILITY_ERROR  
44H: MCA_SVID_CRITICAL VR_FAILED  
45H: MCA_SVID_SA_ITD_ERROR  
46H: MCA_SVID_READ_REG_FAILED  
47H: MCA_SVID_WRITE_REG_FAILED  
4AH: MCA_SVID_PKGREQUEST_FAILED |
### Table 17-34. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4BH</td>
<td>MCA_SVID_IMON_REQUEST_FAILED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4CH</td>
<td>MCA_SVID_ALERT_REQUEST_FAILED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4DH</td>
<td>MCA_SVID_MCP_VR_RAMP_ERROR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>56H</td>
<td>MCA_FIVR_PD_HARDERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>58H</td>
<td>MCA_WATCHDOG_TIMEOUT_PKGC_SECONDARY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59H</td>
<td>MCA_WATCHDOG_TIMEOUT_PKGC_MAIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5AH</td>
<td>MCA_WATCHDOG_TIMEOUT_PKGS_MAIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5BH</td>
<td>MCA_WATCHDOG_TIMEOUT_MSG_CH_FSM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5CH</td>
<td>MCA_WATCHDOG_TIMEOUT_BULK_CR_FSM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5DH</td>
<td>MCA_WATCHDOG_TIMEOUT_IOSFSB_FSM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>60H</td>
<td>MCA_PKGS_SAFE_WP_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>61H</td>
<td>MCA_PKGS_CPD_UNCPD_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>62H</td>
<td>MCA_PKGS_INVALID_REQ_PCH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>63H</td>
<td>MCA_PKGS_INVALID_REQ_INTERNAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64H</td>
<td>MCA_PKGS_INVALID_RSP_INTERNAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>65H-7AH</td>
<td>MCA_PKGS_RESET_PREP_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7BH</td>
<td>MCA_PKGS_SMBUS_VPP_PAUSE_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7CH</td>
<td>MCA_PKGS_SMBUS_MCP_PAUSE_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7DH</td>
<td>MCA_PKGS_SMBUS_SPD_PAUSE_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>80H</td>
<td>MCA_PKGC_DISP_BUSY_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>81H</td>
<td>MCA_PKGC_INVALID_RSP_PCH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>83H</td>
<td>MCA_PKGC_WATCHDOG_HANG_CBZ_DOWN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>84H</td>
<td>MCA_PKGC_WATCHDOG_HANG_CBZ_UP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>87H</td>
<td>MCA_PKGC_WATCHDOG_HANG_C2_BLKMASTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>88H</td>
<td>MCA_PKGC_WATCHDOG_HANG_C2_PSLIMIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>89H</td>
<td>MCA_PKGC_WATCHDOG_HANG_SETDISP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8BH</td>
<td>MCA_PKGC_ALLOW_L1_ERROR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90H</td>
<td>MCA_RECOVERABLE_DIE_THERMAL_TOO_HOT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A0H</td>
<td>MCA_ADR_SIGNAL_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A1H</td>
<td>MCA_BCLK_FREQ_OC_ABOVE_THRESHOLD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B0H</td>
<td>MCA_DISPATCHER_RUN_BUSY_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37:32</td>
<td>ENH_MCA_AVAIL0</td>
<td>Available when Enhanced MCA is in use.</td>
</tr>
<tr>
<td></td>
<td>52:38</td>
<td>CORR_ERR_COUNT</td>
<td>Correctable error count.</td>
</tr>
<tr>
<td></td>
<td>54:53</td>
<td>CORRERRORSTATUSIND</td>
<td>These bits are used to indicate when the number of corrected errors has exceeded the safe threshold to the point where an uncorrected error has become more likely to happen. Table 3 shows the encoding of these bits.</td>
</tr>
<tr>
<td></td>
<td>56:55</td>
<td>ENH_MCA_AVAIL1</td>
<td>Available when Enhanced MCA is in use.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
17.11.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the IA32_MC5_STATUS, IA32_MC7_STATUS, and IA32_MC8_STATUS MSRs. The supported error codes follow the architectural MCACOD definition type 1PPTRRRRIILL; see Chapter 16, "Machine-Check Architecture."

NOTE
The interconnect machine check errors in this section apply only to the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6AH. These do not apply to the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6CH.

Table 17-35 lists model-specific fields to interpret error codes applicable to IA32_MCI_STATUS, where i= 5, 7, 8.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRRIILL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The two supported compound error codes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0x0C0F: Unsupported/Undefined Packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0x0E0F: For all other corrected and uncorrected errors.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>21:16</td>
<td>MSCOD</td>
<td>The encoding of Uncorrectable (UC) errors are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00H: Phy Initialization Failure (NumInit).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01H: Phy Detected Drift Buffer Alarm.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>02H: Phy Detected Latency Buffer Rollover.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10H: LL Rx detected CRC error: unsuccessful LLR (entered Abort state).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11H: LL Rx Unsupported/Undefined packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12H: LL or Phy Control Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13H: LL Rx Parameter Exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1FH: LL Detected Control Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The encoding of correctable (COR) errors are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20H: Phy Initialization Abort.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21H: Phy Inband Reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22H: Phy Lane failure, recovery in x8 width.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23H: Phy LOc error corrected without Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24H: Phy LOc error triggering Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30H: LL Rx detected CRC error: successful LLR without Phy Re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31H: LL Rx detected CRC error: successful LLR with Phy Re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32H: Tx received LLR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
</tbody>
</table>
17.11.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers for the 3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture are defined in Table 17-37.

The MSRs reporting MC error codes differ depending on the CPUID DisplayFamily_DisplaySignature of the processor. See Table 17-36 for details.

Table 17-36. MSRs Reporting MC Error Codes by CPUID DisplayFamily_DisplaySignature

<table>
<thead>
<tr>
<th>Processor</th>
<th>CPUID DisplayFamily_DisplaySignature</th>
<th>MSRs Reporting MC Error Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture</td>
<td>06_6AH</td>
<td>IA32_MC13_STATUS–IA32_MC14_STATUS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_MC17_STATUS–IA32_MC18_STATUS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_MC21_STATUS–IA32_MC22_STATUS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_MC25_STATUS–IA32_MC26_STATUS</td>
</tr>
<tr>
<td>3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture</td>
<td>06_6CH</td>
<td>IA32_MC13_STATUS–IA32_MC14_STATUS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_MC17_STATUS–IA32_MC18_STATUS</td>
</tr>
</tbody>
</table>
The supported error codes follow the architectural MCACOD definition type \texttt{1MMMCCCC}; see Chapter 16, “Machine-Check Architecture.”

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes(^1)</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 0000 0000 1MMM CCCC</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>27:16</td>
<td>Error Codes</td>
<td>0000H: Uncorrectable spare error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0001H: End to End address parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0002H: Write data parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0003H: End to End uncorrectable/correctable write data ECC error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0004H: Write byte enable parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0007H: Transaction ID parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008H: Correctable patrol scrub error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010H: Uncorrectable patrol scrub error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0020H: Correctable spare error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0080H: Transient or correctable error for demand or underfill reads or read 2LM metadata error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00A0H: Uncorrectable error for demand or underfill reads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100H: WDB read parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0108H: DDR/DDRT link failure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0111H: PCLS address CSR parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0112H: PCLS illegal ADDDC configuration error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0200H: DDR4 command / address parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0400H: RPQ scheduler address parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0800H: 2LM unrecognized request type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0801H: 2LM read response to an invalid scoreboard entry.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0802H: 2LM unexpected read response.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0803H: 2LM DDR4 completion to an invalid scoreboard entry.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0804H: 2LM DDRT completion to an invalid scoreboard entry.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0805H: 2LM completion FIFO overflow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0806H: DDRT link parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0807H: DDRT RID uncorrectable error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0809H: DDRT RID FIFO overflow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>080AH: DDRT error on FNV write credits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>080BH: DDRT error on FNV read credits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>080CH: DDRT scheduler error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>080DH: DDRT FNV error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>080EH: DDRT FNV thermal error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>080FH: DDRT unexpected data packet during CMI idle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0810H: DDRT RPQ request parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0811H: DDRT WPQ request parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0812H: 2LM NmFillWr CAM multiple hit error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0813H: CMI credit oversubscription error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0814H: CMI total credit count error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0815H: CMI reserved credit pool error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0816H: DDRT link ECC error.</td>
</tr>
</tbody>
</table>
### Table 17-37. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—14, 17—18, 21—22, 25—26) (Contd.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Mirror secondary error</td>
<td>Mirror secondary error</td>
<td></td>
</tr>
<tr>
<td>31:29</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>37:32</td>
<td>Other Info</td>
<td>Other Info.</td>
<td></td>
</tr>
<tr>
<td>56:38</td>
<td>See Chapter 16, “Machine-Check Architecture.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Register Validity Indicators</td>
<td>63:57</td>
<td></td>
<td>See Chapter 16, “Machine-Check Architecture.”</td>
</tr>
</tbody>
</table>

### NOTES:

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

**Table 17-38. Additional Information Reported in IA32_MCI_MISC (i= 13—14, 17—18, 21—22, 25—26)**

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:0</td>
<td>LSB</td>
<td>See Figure 16-8.</td>
</tr>
<tr>
<td>8:6</td>
<td>Address Mode</td>
<td>See Table 16-3.</td>
</tr>
<tr>
<td>18:9</td>
<td>Column</td>
<td>Component of sub-DIMM address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits 18-17: Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 16: Column 9.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 15: Column 8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 14: Column 7.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 13: Column 6.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 12: Column 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 11: Column 4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 10: Column 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 9: Reserved.</td>
</tr>
<tr>
<td>39:19</td>
<td>Row</td>
<td>Component of sub-DIMM address.</td>
</tr>
<tr>
<td>45:40</td>
<td>Bank</td>
<td>Component of sub-DIMM address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 45: Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 44: Bank group 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 43: Bank address 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 42: Bank address 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 41: Bank group 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 40: Bank group 0.</td>
</tr>
<tr>
<td>51:46</td>
<td>Failed Device</td>
<td>Failing device for correctable error (not valid for uncorrectable or transient errors).</td>
</tr>
<tr>
<td>55:52</td>
<td>CBit</td>
<td>CBit</td>
</tr>
<tr>
<td>58:56</td>
<td>Chip Select</td>
<td>Chip Select</td>
</tr>
<tr>
<td>62:59</td>
<td>ECC Mode</td>
<td>00000b: SDDC 2LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001b: SDDC 1LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010b: SDDC + 1 2LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011b: SDDC + 1 1LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100b: ADDDC 2LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101b: ADDDC 1LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110b: ADDDC + 1 2LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111b: ADDDC + 1 1LM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000b: Read from DDRT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001b: x8 SDDC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010b: x8 SDDC + 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011b: Not a valid ECC mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other values: Reserved.</td>
</tr>
<tr>
<td>63</td>
<td>Transient</td>
<td>0b:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1b: Error was transient.</td>
</tr>
</tbody>
</table>
### 17.11.4 M2M Machine Check Errors

MC error codes associated with M2M for the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6AH are reported in the IA32_MC12_STATUS, IA32_MC16_STATUS, IA32_MC20_STATUS, and IA32_MC24_STATUS MSRs.

MC error codes associated with M2M for the 3rd generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_6CH are reported in the IA32_MC12_STATUS and IA32_MC16_STATUS MSRs.

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, “Machine-Check Architecture.”

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes¹</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Compound error format: 0000 0000 1MMM CCCC</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>23:16</td>
<td>MSCOD</td>
<td>Logged an MC error.</td>
</tr>
<tr>
<td></td>
<td>25:24</td>
<td>MsccddDRTypE</td>
<td>Logged a DDR/DDRT specific error.</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>MsccdfailoverWhileResetPrep</td>
<td>Logged a failover specific error while preparing to reset.</td>
</tr>
<tr>
<td></td>
<td>31:27</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>37:32</td>
<td>Other Info</td>
<td>Other information.</td>
</tr>
<tr>
<td></td>
<td>56:38</td>
<td></td>
<td>See Chapter 16, “Machine-Check Architecture.”</td>
</tr>
</tbody>
</table>

**NOTES:**
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

MC error codes associated with mirrored memory corrections are reported in the IA32_MC12_MISC, IA32_MC16_MISC, IA32_MC20_MISC, and IA32_MC24_MISC MSRs. The model-specific error codes listed in Table 17-32 also apply to IA32_MCI_MISC, where i = 12, 16, 20, 24.

### 17.12 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_86H, MACHINE ERROR CODES FOR MACHINE CHECK

In Intel Atom® processors based on Tremont microarchitecture with CPUID DisplayFamily_DisplaySignature 06_86H, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-34 in Section 17.11.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

#### 17.12.1 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC13_STATUS–IA32_MC15_STATUS. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, “Machine-Check Architecture.”

The IA32_MCI_STATUS MSR (where i = 13, 14, 15) contains information related to a machine check error if its VAL(valid) flag is set. Bit definitions are the same as those found in Table 17-37 "Intel IMC MC Error Codes for IA32_MCI_STATUS (i= 13—14, 17—18, 21—22, 25—26)."

The IA32_MCI_MISC MSR (where i = 13, 14, 15) contains information related memory corrections. Bit definitions are the same as those found in Table 17-38 "Additional Information Reported in IA32_MCI_MISC (i= 13—14, 17—18, 21—22, 25—26)."
17.12.2 M2M Machine Check Errors

MC error codes associated with M2M are reported in the IA32_MC12_STATUS MSR. The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, “Machine-Check Architecture.” Bit definitions are the same as those found in Table 17-39 "M2M MC Error Codes for IA32_MCi_STATUS (i = 12, 16, 20, 24)."

17.13 INCREMENTAL DECODING INFORMATION: 4TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK

In the 4th generation Intel® Xeon® Scalable Processor Family with CPUID DisplayFamily_DisplaySignature of 06_8FH, incremental error codes for internal machine check errors from the PCU controller are reported in the register bank IA32_MC4. Table 17-40 in Section 17.13.1 lists model-specific fields to interpret error codes applicable to IA32_MC4_STATUS.

17.13.1 Internal Machine Check Errors

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCACOD</td>
<td>15:0</td>
<td>Internal Errors</td>
<td>The value of this field will be 0402H for the PCU and 0406H for internal firmware errors. This applies for any logged error.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>19:16</td>
<td>Reserved, except for the following</td>
<td>Model specific error code bits 19:16. If MACOD = 40CH, MSCOD encoding should be interpreted as: 01H: MCE when CR4.MCE is clear. 02H: MCE when MCIP bit is set. 03H: MCE under WPS. 04H: Unrecoverable error during security flow execution. 05H: Software triple fault shutdown. 06H: VMX-exit-consistency-check failures. 07H: RSM-consistency-check failures. 08H: Invalid conditions on protected mode SMM entry. 09H: Unrecoverable error during security flow execution. For all other MACOD values, MSCOD logs the type of hardware UC (PCU/VCU) error that has occurred. There are seven errors defined: 01H: Instruction address out of valid space. 02H: Double bit RAM error on Instruction Fetch. 03H: Invalid OpCode seen. 04H: Stack Underflow. 05H: Stack Overflow. 06H: Data address out of valid space. 07H: Double bit RAM error on Data Fetch.</td>
</tr>
</tbody>
</table>
### Table 17-40. Machine Check Error Codes for IA32_MC4_STATUS (Contd.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>23:20</td>
<td>Reserved except for the following</td>
<td>Model specific error code bits 23:20. This logs the type of HW FSM error that has occurred. There are 3 errors defined: 04H: Clock/power IP response timeout. 05H: SMBus controller raised SMI. 09H: PM controller received invalid transaction.</td>
<td></td>
</tr>
</tbody>
</table>
17.13.2 Interconnect Machine Check Errors

MC error codes associated with the link interconnect agents are reported in the IA32_MC5_STATUS MSR. The supported error codes follow the architectural MCACOD definition type 1PPTRRRIII; see Chapter 16, "Machine-Check Architecture."

Table 17-41 lists model-specific fields to interpret error codes applicable to IA32_MC5_STATUS.
## Table 17-41. Interconnect MC Error Codes for IA32_MC5_STATUS

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Bus error format: 1PPTRRRIILL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The two supported compound error codes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0x0C0F: Unsupported/Undefined Packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 0x0E0F: For all other corrected and uncorrected errors.</td>
</tr>
<tr>
<td>Model Specific Errors</td>
<td>21:16</td>
<td>MSCOD</td>
<td>The encoding of Uncorrectable (UC) errors are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00H: UC Phy Initialization Failure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01H: UC Phy Detected Drift Buffer Alarm.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>02H: UC Phy Detected Latency Buffer Rollover.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10H: UC LL Rx detected CRC error: unsuccessful LLR (entered Abort state).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11H: UC LL Rx Unsupported/Undefined packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12H: UC LL or Phy Control Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13H: UC LL Rx Parameter Exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15H: UC LL Rx SGX MAC Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1FH: UC LL Detected Control Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The encoding of correctable (COR) errors are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20H: COR Phy Initialization Abort.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21H: COR Phy Inband Reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22H: COR Phy Lane failure, recovery in x8 width.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23H: COR Phy L0c error corrected without Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24H: COR Phy L0c error triggering Phy reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30H: COR LL Rx detected CRC error: successful LLR without Phy Re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31H: COR LL Rx detected CRC error: successful LLR with Phy Re-init.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td></td>
<td>31:22</td>
<td>MSCOD_SPARE</td>
<td>The definition below applies to MSCOD T2H (UC LL or Phy Control Errors).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 22]: Phy Control Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 23]: Unexpected Retry.Ack flit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 24]: Unexpected Retry.Req flit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 25]: RF parity error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 26]: Routeback Table error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 27]: Unexpected Tx Protocol flit (EOP, Header, or Data).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 28]: Rx Header-or-Credit BGF credit overflow/underflow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 29]: Link Layer Reset still in progress when Phy enters L0 (Phy training</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>should not be enabled until after LL reset is complete as indicated by KTLCL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 30]: Link Layer reset initiated while protocol traffic not idle.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[Bit 31]: Link Layer Tx Parity Error.</td>
</tr>
<tr>
<td></td>
<td>37:32</td>
<td>OTHER_INFO</td>
<td>Other Info.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
17.13.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers for the 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture are reported in the IA32_MC13_STATUS—IA32_MC20_STATUS MSRs.

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, "Machine-Check Architecture."

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes</td>
<td>15:0</td>
<td>MCACOD</td>
<td>Memory Controller error format: 0000 0000 1MMM CCCC</td>
</tr>
</tbody>
</table>
| Model Specific Errors       | 31:16   | Reserved, except for the following | 0001H: Address parity error.  
0002H: Data parity error.  
0003H: Data ECC error.  
0004H: Data byte enable parity error.  
0007H: Transaction ID parity error.  
0008H: Corrected patrol scrub error.  
0010H: Uncorrected patrol scrub error.  
0020H: Corrected spare error.  
0040H: Uncorrected spare error.  
0080H: Corrected read error.  
00A0H: Uncorrected read error.  
00A1H: Uncorrected metadata.  
0100H: WDB read parity error.  
0108H: DDR link failure.  
0200H: DDR5 command / address parity error.  
0400H: RPQ0 parity (primary) error.  
0800H: DDR-T bad request.  
0801H: DDR Data response to an invalid entry.  
0802H: DDR data response to an entry not expecting data.  
0803H: DDR5 completion to an invalid entry.  
0804H: DDR-T completion to an invalid entry.  
0805H: DDR data/completion FIFO overflow.  
0806H: DDR-T ERID correctable parity error.  
0807H: DDR-T ERID uncorrectable error.  
0808H: DDR-T interrupt received while outstanding interrupt was not ACKed.  
0809H: ERID FIFO overflow.  
080AH: DDR-T error on FNV write credits.  
080BH: DDR-T error on FNV read credits.  
080CH: DDR-T scheduler error.  
080DH: DDR-T FNV error event.  
080EH: DDR-T FNV thermal event.  
080FH: CMI packet while idle.  
0810H: DDR_T_RPQ_REQ_PARITY_ERR.  
0811H: DDR_T_WPQ_REQ_PARITY_ERR.  
0812H: 2LM_NMFILLWR_CAM_ERR. |
### Table 17-42. Intel IMC MC Error Codes for IA32_MCI_STATUS (i = 13—20)

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0813H:</td>
<td>CMI_CREDIT_OVERSUB_ERR.</td>
</tr>
<tr>
<td></td>
<td>0814H:</td>
<td>CMI_CREDIT_TOTAL_ERR.</td>
</tr>
<tr>
<td></td>
<td>0815H:</td>
<td>CMI_CREDIT_RSVD_POOL_ERR.</td>
</tr>
<tr>
<td></td>
<td>0816H:</td>
<td>DDR_T_RD_ERROR.</td>
</tr>
<tr>
<td></td>
<td>0817H:</td>
<td>WDB_FIFO_ERR.</td>
</tr>
<tr>
<td></td>
<td>0818H:</td>
<td>CMI_REQ_FIFO_OVERFLOW.</td>
</tr>
<tr>
<td></td>
<td>0819H:</td>
<td>CMI_REQ_FIFO_UNDERFLOW.</td>
</tr>
<tr>
<td></td>
<td>081AH:</td>
<td>DDR_T_WR_CMPL_FIFO_OVERFLOW.</td>
</tr>
<tr>
<td></td>
<td>081BH:</td>
<td>DDR_T_WR_CMPL_FIFO_UNDERFLOW.</td>
</tr>
<tr>
<td></td>
<td>081CH:</td>
<td>CMI_MISC_MC_CREDIT_CRITICAL_ERRORS.</td>
</tr>
<tr>
<td></td>
<td>081DH:</td>
<td>CMI_MISC_MC_ARB_ERRORS.</td>
</tr>
<tr>
<td></td>
<td>081EH:</td>
<td>DDR_T_WR_CMPL_FIFO_OVERFLOW.</td>
</tr>
<tr>
<td></td>
<td>081FH:</td>
<td>DDR_T_WR_CMPL_FIFO_UNDERFLOW.</td>
</tr>
<tr>
<td></td>
<td>0820H:</td>
<td>CMI_RD_CPL_FIFO_OVERFLOW.</td>
</tr>
<tr>
<td></td>
<td>0821H:</td>
<td>CMI_RD_CPL_FIFO_UNDERFLOW.</td>
</tr>
<tr>
<td></td>
<td>0822H:</td>
<td>TME_KEY_PAR_ERR.</td>
</tr>
<tr>
<td></td>
<td>0823H:</td>
<td>TME_CMI_MISC_ERR.</td>
</tr>
<tr>
<td></td>
<td>0824H:</td>
<td>TME_CMI_OVFL_ERR.</td>
</tr>
<tr>
<td></td>
<td>0825H:</td>
<td>TME_CMI_UFLL_ERR.</td>
</tr>
<tr>
<td></td>
<td>0826H:</td>
<td>TME_TEM_SECURE_ERR.</td>
</tr>
<tr>
<td></td>
<td>0827H:</td>
<td>TME_UFILL_PAR_ERR.</td>
</tr>
<tr>
<td></td>
<td>0829H:</td>
<td>INTERNAL_ERR.</td>
</tr>
<tr>
<td></td>
<td>082AH:</td>
<td>TME_INTEGRITY_ERR.</td>
</tr>
<tr>
<td></td>
<td>082BH:</td>
<td>TME_TDX_ERR</td>
</tr>
<tr>
<td></td>
<td>082CH:</td>
<td>TME_UFILL_TEM_SECURE_ERR.</td>
</tr>
<tr>
<td></td>
<td>082DH:</td>
<td>TME_KEY_POISON_ERR.</td>
</tr>
<tr>
<td></td>
<td>082EH:</td>
<td>TME_SECURITY_ENGINE_ERR.</td>
</tr>
<tr>
<td></td>
<td>1008H:</td>
<td>CORR_PATSCRUB_MIRR2ND_ERR.</td>
</tr>
<tr>
<td></td>
<td>1010H:</td>
<td>UC_PATSCRUB_MIRR2ND_ERR.</td>
</tr>
<tr>
<td></td>
<td>1020H:</td>
<td>COR_SPARE_MIRR2ND_ERR.</td>
</tr>
<tr>
<td></td>
<td>1040H:</td>
<td>UC_SPARE_MIRR2ND_ERR.</td>
</tr>
<tr>
<td></td>
<td>1080H:</td>
<td>HA_RD_MIRR2ND_ERR.</td>
</tr>
<tr>
<td></td>
<td>10A0H:</td>
<td>HA_UNCORR_RD_MIRR2ND_ERR.</td>
</tr>
</tbody>
</table>

### Status Register

| Validity Indicators<sup>1</sup> | 63:57 | Other Info |

### Notes:

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Additional information is reported in the IA32_MC13_MISC–IA32_MC20_MISC MSRs. Table 17-43 lists the information reported in IA32_MCI_MISC, where i = 13–20.
### Table 17-43. Additional Information Reported in IA32_MCi_MISC (i= 13—20)

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:0</td>
<td>LSB</td>
<td>See Figure 16-8.</td>
</tr>
<tr>
<td>8:6</td>
<td>Address Mode</td>
<td>See Table 16-3.</td>
</tr>
<tr>
<td>18:9</td>
<td>Column</td>
<td>Column address for the last retry. To get the real column address from this field, shift the value left by 2.</td>
</tr>
<tr>
<td>36:19</td>
<td>Row</td>
<td>Component of sub-DIMM address.</td>
</tr>
<tr>
<td>42:37</td>
<td>Bank ID</td>
<td>Component of sub-DIMM address. &lt;br&gt; Bit 42: Reserved. &lt;br&gt; Bit 41: Bank group 2. &lt;br&gt; Bit 40: Bank address 1. &lt;br&gt; Bit 39: Bank address 0. &lt;br&gt; Bit 38: Bank group 1. &lt;br&gt; Bit 37: Bank group 0.</td>
</tr>
<tr>
<td>48:43</td>
<td>Failed Device</td>
<td>Failing device for correctable error (not valid for uncorrectable or transient errors).</td>
</tr>
<tr>
<td>50:49</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>55:51</td>
<td>Failed Device Number</td>
<td>In HBM mode, holds the failed device number for upper 32 bytes.</td>
</tr>
<tr>
<td>55:52</td>
<td>CBit</td>
<td>In DDR mode, bits 54-52: sub_rank[2:0]; bit 55: reserved.</td>
</tr>
<tr>
<td>58:56</td>
<td>Chip Select</td>
<td>Chip Select</td>
</tr>
<tr>
<td>63</td>
<td>Transient</td>
<td>Indicates if the error was a transient error. A transient error is only indicated for demand reads, underfill reads, and patrol. If there was a WDBParity Error, this field indicates the WDB ID bit 6.</td>
</tr>
</tbody>
</table>

### 17.13.4 M2M Machine Check Errors

MC error codes associated with M2M for the 4th generation Intel Xeon Scalable Processor Family with a CPUID DisplayFamily_DisplaySignature of 06_8FH are reported in the IA32_MC12_STATUS MSR.

The supported error codes follow the architectural MCACOD definition type 1MMMCCCC; see Chapter 16, "Machine-Check Architecture."
17.13.5 High Bandwidth Memory Machine Check Errors

MC error codes associated with high bandwidth memory for the 4th generation Intel Xeon Scalable Processor Family are reported in the IA32_MC29_STATUS–IA32_MC31_STATUS MSRs.

17.14 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 0FH, MACHINE ERROR CODES FOR MACHINE CHECK

Table 17-45 provides information for interpreting additional family 0FH model-specific fields for external bus errors. These errors are reported in the IA32_MCI_STATUS MSRs. They are reported architecturally as compound errors with a general form of 0000 1PPT RRRR IILL in the MCA error code field. See Chapter 16 for information on the interpretation of compound error codes.
Table 17-45. Incremental Decoding Information: Processor Family 0FH, Machine Error Codes for Machine Check

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA Error Codes †</td>
<td>15:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model-Specific Error Codes</td>
<td>16</td>
<td>FSB Address Parity</td>
<td>Address parity error detected: 1: Address parity error detected. 0: No address parity error.</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Response Hard Fail</td>
<td>Hardware failure detected on response.</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Response Parity</td>
<td>Parity error detected on response.</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>PIC and FSB Data Parity</td>
<td>Data Parity detected on either PIC or FSB access.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Processor Signature =</td>
<td>Processor Signature = 00000F04H: Indicates error due to an invalid PIC request access was made to PIC space with WB memory): 1: Invalid PIC request error. 0: No Invalid PIC request error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00000F04H: Invalid PIC Request</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other processors:</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>Pad State Machine</td>
<td>The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>Pad Strobe Glitch</td>
<td>Data strobe glitch.</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Pad Address Glitch</td>
<td>Address strobe glitch.</td>
</tr>
<tr>
<td>Other Information</td>
<td>56:24</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Status Register Validity Indicators †</td>
<td>63:57</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.

Table 17-10 provides information on interpreting additional family 0FH model specific fields for cache hierarchy errors. These errors are reported in one of the IA32_MCI_STATUS MSRs. These errors are reported, architecturally, as compound errors with a general form of **0000 0001 RRRR TTLL** in the MCA error code field. See Chapter 16 for how to interpret the compound error code.

17.14.1 Model-Specific Machine Check Error Codes for the Intel® Xeon® Processor MP 7100 Series

The Intel Xeon processor MP 7100 series has five register banks which contain information related to Machine Check Errors. MCI_STATUS[63:0] refers to all five register banks. MC0_STATUS[63:0] through MC3_STATUS[63:0] is the same as previous generations of Intel Xeon processors within Family 0FH. MC4_STATUS[63:0] is the main error logging for the processor’s L3 and front side bus errors. It supports the L3 Errors, Bus and Interconnect Errors Compound Error Codes in the MCA Error Code Field.
17.14.1.1 Processor Machine Check Status Register MCA Error Code Definition

The Intel Xeon processor MP 7100 series uses compound MCA Error Codes for logging its CBC internal machine check errors, L3 Errors, and Bus/Interconnect Errors. It defines additional Machine Check error types (IA32_MC4_STATUS[15:0]) beyond those defined in Chapter 16. Table 17-47 lists these model-specific MCA error codes. Error code details are specified in MC4_STATUS [31:16]; see Section 17.14.3, the "Model Specific Error Code" field. The information in the "Other_Info" field (MC4_STATUS[56:32]) is common to the three processor error types and contains a correctable event count and specifies the MC4_MISC register format.

Table 17-46. MCI_STATUS Register Bit Definition

<table>
<thead>
<tr>
<th>Bit Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA_Error_Code</td>
<td>15:0</td>
<td>This field specifies the machine check architecture defined error code for the machine check error condition detected. The machine check architecture defined error codes are guaranteed to be the same for all Intel Architecture processors that implement the machine check architecture. See tables below.</td>
</tr>
<tr>
<td>Model_Specific_Error_Code</td>
<td>31:16</td>
<td>This field specifies the model specific error code that uniquely identifies the machine check error condition detected. The model specific error codes may differ among Intel Architecture processors for the same Machine Check Error condition. See tables below.</td>
</tr>
<tr>
<td>Other_Info</td>
<td>56:32</td>
<td>The functions of the bits in this field are implementation specific and are not part of the machine check architecture. Software that is intended to be portable among Intel Architecture processors should not rely on the values in this field.</td>
</tr>
<tr>
<td>PCC</td>
<td>57</td>
<td>The Processor Context Corrupt flag indicates that the state of the processor might have been corrupted by the error condition detected and that reliable restarting of the processor may not be possible. When clear, this flag indicates that the error did not affect the processor’s state. This bit will always be set for MC errors, which are not corrected.</td>
</tr>
<tr>
<td>ADDRV</td>
<td>58</td>
<td>The MC_ADDR register valid flag indicates that the MC_ADDR register contains the address where the error occurred. When clear, this flag indicates that the MC_ADDR register does not contain the address where the error occurred. The MC_ADDR register should not be read if the ADDRV bit is clear.</td>
</tr>
<tr>
<td>MISCV</td>
<td>59</td>
<td>The MC_MISC register valid flag indicates that the MC_MISC register contains additional information regarding the error. When clear, this flag indicates that the MC_MISC register does not contain additional information regarding the error. MC_MISC should not be read if the MISCV bit is not set.</td>
</tr>
<tr>
<td>EN</td>
<td>60</td>
<td>The error enabled flag indicates that reporting of the machine check exception for this error was enabled by the associated flag bit of the MC_CTL register. Note that correctable errors do not have associated enable bits in the MC_CTL register so the EN bit should be clear when a correctable error is logged.</td>
</tr>
<tr>
<td>UC</td>
<td>61</td>
<td>The error uncorrected flag indicates that the processor did not correct the error condition. When clear, this flag indicates that the processor was able to correct the event condition.</td>
</tr>
<tr>
<td>OVER</td>
<td>62</td>
<td>The machine check overflow flag indicates that a machine check error occurred while the results of a previous error were still in the register bank (i.e., the VAL bit was already set in the MC_STATUS register). The processor sets the OVER flag and software is responsible for clearing it. Enabled errors are written over disabled errors, and uncorrected errors are written over corrected events. Uncorrected errors are not written over previous valid uncorrected errors.</td>
</tr>
<tr>
<td>VAL</td>
<td>63</td>
<td>The MC_STATUS register valid flag indicates that the information within the MC_STATUS register is valid. When this flag is set, the processor follows the rules given for the OVER flag in the MC_STATUS register when overwriting previously valid entries. The processor sets the VAL flag and software is responsible for clearing it.</td>
</tr>
</tbody>
</table>
### Table 17-47. Incremental MCA Error Code for Intel® Xeon® Processor MP 7100

<table>
<thead>
<tr>
<th>Type</th>
<th>Error Code</th>
<th>Binary Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Internal Error</td>
<td>0000 0100 0000 0000</td>
<td>Internal Error Type Code.</td>
</tr>
<tr>
<td>A</td>
<td>L3 Tag Error</td>
<td>0000 0001 0000 1011</td>
<td>L3 Tag Error Type Code.</td>
</tr>
<tr>
<td>B</td>
<td>Bus and Interconnect Error</td>
<td>0000 100x 0000 1111</td>
<td>Not used, but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 101x 0000 1111</td>
<td>Not used, but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 110x 0000 1111</td>
<td>Not used, but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 1110 0000 1111</td>
<td>Bus and Interconnection Error Type Code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 1111 0000 1111</td>
<td>Not used, but this encoding is reserved for compatibility with other MCA implementations.</td>
</tr>
</tbody>
</table>

The **bold faced** binary encodings are the only encodings used by the processor for MC4_STATUS[15:0].

#### 17.14.2 Other_Info Field (All MCA Error Types)

The MC4_STATUS[56:32] field is common to the processor's three MCA error types (A, B, and C).

### Table 17-48. Other Information Field Bit Definition

<table>
<thead>
<tr>
<th>Bit Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>39:32</td>
<td>8-bit Correctable Event Count</td>
<td>This field holds a count of the number of correctable events since cold reset. This is a saturating counter; the counter begins at 1 (with the first error) and saturates at a count of 255.</td>
</tr>
<tr>
<td>41:40</td>
<td>MC4_MISC Format Type</td>
<td>The value in this field specifies the format of information in the MC4_MISC register. Currently, only two values are defined. Valid only when MISCV is asserted.</td>
</tr>
<tr>
<td>43:42</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>51:44</td>
<td>ECC Syndrome</td>
<td>ECC syndrome value for a correctable ECC event when the &quot;Valid ECC syndrome&quot; bit is asserted.</td>
</tr>
<tr>
<td>52</td>
<td>Valid ECC Syndrome</td>
<td>Set when a correctable ECC event supplies the ECC syndrome.</td>
</tr>
<tr>
<td>54:53</td>
<td>Threshold-Based Error Status</td>
<td>00: No tracking. No hardware status tracking is provided for the structure reporting this event. 01: Green. Status tracking is provided for the structure posting the event; the current status is green (below threshold). 10: Yellow. Status tracking is provided for the structure posting the event; the current status is yellow (above threshold). 11: Reserved for future use. Valid only if the Valid bit (bit 63) is set. Undefined if the UC bit (bit 61) is set.</td>
</tr>
<tr>
<td>56:55</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
17.14.3 Processor Model Specific Error Code Field

17.14.3.1 MCA Error Type A: L3 Error

Note: The Model Specific Error Code field in MC4_STATUS (bits 31:16).

<table>
<thead>
<tr>
<th>Bit Num</th>
<th>Sub-Field Name</th>
<th>Description</th>
<th>Legal Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18:16</td>
<td>L3 Error Code</td>
<td>Describes the L3 error encountered</td>
<td>000: No error. 001: More than one way reporting a correctable event. 010: More than one way reporting an uncorrectable error. 011: More than one way reporting a tag hit. 100: No error. 101: One way reporting a correctable event. 110: One way reporting an uncorrectable error. 111: One or more ways reporting a correctable event while one or more ways are reporting an uncorrectable error.</td>
</tr>
<tr>
<td>20:19</td>
<td>---</td>
<td>Reserved</td>
<td>00</td>
</tr>
<tr>
<td>31:21</td>
<td>---</td>
<td>Fixed pattern</td>
<td>0010_0000_000</td>
</tr>
</tbody>
</table>

17.14.3.2 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error

Note: The Model Specific Error Code field in MC4_STATUS (bits 31:16).

<table>
<thead>
<tr>
<th>Bit Num</th>
<th>Sub-Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>FSB Request Parity</td>
<td>Parity error detected during FSB request phase.</td>
</tr>
<tr>
<td>17</td>
<td>Core0 Addr Parity</td>
<td>Parity error detected on Core 0 request’s address field.</td>
</tr>
<tr>
<td>18</td>
<td>Core1 Addr Parity</td>
<td>Parity error detected on Core 1 request’s address field.</td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>FSB Response Parity</td>
<td>Parity error on FSB response field detected.</td>
</tr>
<tr>
<td>21</td>
<td>FSB Data Parity</td>
<td>FSB data parity error on inbound data detected.</td>
</tr>
<tr>
<td>22</td>
<td>Core0 Data Parity</td>
<td>Data parity error on data received from Core 0 detected.</td>
</tr>
<tr>
<td>23</td>
<td>Core1 Data Parity</td>
<td>Data parity error on data received from Core 1 detected.</td>
</tr>
<tr>
<td>24</td>
<td>IDS Parity</td>
<td>Detected an Enhanced Defer parity error (phase A or phase B).</td>
</tr>
<tr>
<td>25</td>
<td>FSB Inbound Data ECC</td>
<td>Data ECC event to error on inbound data (correctable or uncorrectable).</td>
</tr>
<tr>
<td>26</td>
<td>FSB Data Glitch</td>
<td>Pad logic detected a data strobe ‘glitch’ (or sequencing error).</td>
</tr>
<tr>
<td>27</td>
<td>FSB Address Glitch</td>
<td>Pad logic detected a request strobe ‘glitch’ (or sequencing error).</td>
</tr>
<tr>
<td>31:28</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Exactly one of the bits defined in the preceding table will be set for a Bus and Interconnect Error. The Data ECC can be correctable or uncorrectable; the MC4_STATUS.UC bit distinguishes between correctable and uncorrectable cases with the Other_Info field possibly providing the ECC Syndrome for correctable errors. All other errors for this processor MCA Error Type are uncorrectable.

17.14.3.3 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error

Table 17-51. Type C: Cache Bus Controller Error Codes

<table>
<thead>
<tr>
<th>MC4_STATUS[31:16] (MSCE) Value</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_0000_0000_0001 0001H</td>
<td>Inclusion Error from Core 0.</td>
</tr>
<tr>
<td>0000_0000_0000_0010 0002H</td>
<td>Inclusion Error from Core 1.</td>
</tr>
<tr>
<td>0000_0000_0000_0011 0003H</td>
<td>Write Exclusive Error from Core 0.</td>
</tr>
<tr>
<td>0000_0000_0000_0100 0004H</td>
<td>Write Exclusive Error from Core 1.</td>
</tr>
<tr>
<td>0000_0000_0000_0101 0005H</td>
<td>Inclusion Error from FSB.</td>
</tr>
<tr>
<td>0000_0000_0000_0110 0006H</td>
<td>SNP Stall Error from FSB.</td>
</tr>
<tr>
<td>0000_0000_0000_0111 0007H</td>
<td>Write Stall Error from FSB.</td>
</tr>
<tr>
<td>0000_0000_0000_1000 0008H</td>
<td>FSB Arb Timeout Error.</td>
</tr>
<tr>
<td>0000_0000_0000_1001 0009H</td>
<td>CBC ODD Queue Underflow/overflow.</td>
</tr>
<tr>
<td>0000_0001_0000_0000 0100H</td>
<td>Enhanced Intel SpeedStep Technology TM1-TM2 Error.</td>
</tr>
<tr>
<td>0000_0010_0000_0000 0200H</td>
<td>Internal Timeout Error.</td>
</tr>
<tr>
<td>0000_0011_0000_0000 0300H</td>
<td>Internal Timeout Error.</td>
</tr>
<tr>
<td>0000_0011_0000_0000 0400H</td>
<td>Intel® Cache Safe Technology Queue Full Error or Disabled-ways-in-a-set overflow.</td>
</tr>
<tr>
<td>1100_0000_0000_0001 C001H</td>
<td>Correctable ECC event on outgoing FSB data.</td>
</tr>
<tr>
<td>1100_0000_0000_0010 C002H</td>
<td>Correctable ECC event on outgoing Core 0 data.</td>
</tr>
<tr>
<td>1100_0000_0000_0100 C004H</td>
<td>Correctable ECC event on outgoing Core 1 data.</td>
</tr>
<tr>
<td>1110_0000_0000_0001 E001H</td>
<td>Uncorrectable ECC error on outgoing FSB data.</td>
</tr>
<tr>
<td>1110_0000_0000_0010 E002H</td>
<td>Uncorrectable ECC error on outgoing Core 0 data.</td>
</tr>
<tr>
<td>1110_0000_0000_0100 E004H</td>
<td>Uncorrectable ECC error on outgoing Core 1 data.</td>
</tr>
<tr>
<td>— All other encodings —</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

All errors, except for the correctable ECC types, in this table are uncorrectable. The correctable ECC events may supply the ECC syndrome in the Other_Info field of the MC4_STATUS MSR.
### Table 17-52. Decoding Family 0FH Machine Check Codes for Cache Hierarchy Errors

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit No.</th>
<th>Bit Function</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCA error codes</td>
<td>15:0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Specific Error Codes</td>
<td>17:16</td>
<td>Tag Error Code</td>
<td>Contains the tag error code for this machine check error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00: No error detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01: Parity error on tag miss with a clean line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10: Parity error/multiple tag match on tag hit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11: Parity error/multiple tag match on tag miss.</td>
</tr>
<tr>
<td></td>
<td>19:18</td>
<td>Data Error Code</td>
<td>Contains the data error code for this machine check error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00: No error detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01: Single bit error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10: Double bit error on a clean line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11: Double bit error on a modified line.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>L3 Error</td>
<td>This bit is set if the machine check error originated in the L3 (it can be ignored for invalid PIC request errors):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: L3 error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: L2 error.</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>Invalid PIC Request</td>
<td>Indicates error due to invalid PIC request access was made to PIC space with WB memory:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: Invalid PIC request error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: No invalid PIC request error.</td>
</tr>
<tr>
<td></td>
<td>31:22</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Other Information</td>
<td>39:32</td>
<td>8-bit Error Count</td>
<td>Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 255.</td>
</tr>
<tr>
<td></td>
<td>56:40</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>Status Register Validity Indicators</td>
<td>63:57</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. These fields are architecturally defined. Refer to Chapter 16, “Machine-Check Architecture,” for more information.
NOTE
This chapter makes numerous references to last-branch recording (LBR) facilities. Unless noted otherwise, all such references in this chapter are to an earlier non-architectural form of the feature. Chapter 19 defines an architectural form of last-branch recording that is supported on newer processors.

Intel 64 and IA-32 architectures provide debug facilities for use in debugging code and monitoring performance. These facilities are valuable for debugging application software, system software, and multitasking operating systems. Debug support is accessed using debug registers (DR0 through DR7) and model-specific registers (MSRs):

• Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.

• MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.

• Time stamp counter is described in Section 18.17, “Time-Stamp Counter.”

• Features that allow monitoring of shared platform resources such as the L3 cache are described in Section 18.18, “Intel® Resource Director Technology (Intel® RDT) Monitoring Features.”

• Features that enable control over shared platform resources are described in Section 18.19, “Intel® Resource Director Technology (Intel® RDT) Allocation Features.”

• Features that enable control over shared platform resources for non-CPU agents are described in Section 18.20, “Intel® Resource Director Technology (Intel® RDT) for Non-CPU Agents.”

18.1 OVERVIEW OF DEBUG SUPPORT FACILITIES
The following processor facilities support debugging and performance monitoring:

• **Debug exception (#DB)** — Transfers program control to a debug procedure or task when a debug event occurs.

• **Breakpoint exception (#BP)** — See breakpoint instruction (INT3) below.

• **Breakpoint-address registers (DR0 through DR3)** — Specifies the addresses of up to 4 breakpoints.

• **Debug status register (DR6)** — Reports the conditions that were in effect when a debug or breakpoint exception was generated.

• **Debug control register (DR7)** — Specifies the forms of memory or I/O access that cause breakpoints to be generated.

• **T (trap) flag, TSS** — Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.

• **RF (resume) flag, EFLAGS register** — Suppresses multiple exceptions to the same instruction.

• **TF (trap) flag, EFLAGS register** — Generates a debug exception (#DB) after every execution of an instruction.

• **Breakpoint instruction (INT3)** — Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set instruction breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.

• **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consists of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

• Task switch to a specific task.
• Execution of the breakpoint instruction.
• Execution of any instruction.
• Execution of an instruction at a specified address.
• Read or write to a specified memory address/range.
• Write to a specified memory address/range.
• Input from a specified I/O address/range.
• Output to a specified I/O address/range.
• Attempt to change the contents of a debug register.

### 18.2 DEBUG REGISTERS

Eight debug registers (see Figure 18-1 for 32-bit operation and Figure 18-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.
Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 though 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 18.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

Figure 18-1. Debug Registers
18.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 18-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

18.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

18.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 18-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3** (breakpoint condition detected) flags (bits 0 through 3) — Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LENn, and R/Wn flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.

- **BLD** (bus-lock detected) flag (bit 11) — Indicates (when clear) that the debug exception was triggered by the assertion of a bus lock when CPL > 0 and OS bus-lock detection was enabled (see Section 18.3.1.6). Other debug exceptions do not modify this bit. To avoid confusion in identifying debug exceptions, software debug-exception handlers should set bit 11 to 1 before returning. (Software that never enables OS bus-lock detection need not do this as DR6[11] = 1 following reset.) This bit is always 1 if the processor does not support OS bus-lock detection.

- **BD** (debug register access detected) flag (bit 13) — Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 18.2.4, “Debug Control Register (DR7),” for further explanation of the purpose of this flag.

- **BS** (single step) flag (bit 14) — Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.

- **BT** (task switch) flag (bit 15) — Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 8.2.1, “Task-State Segment (TSS),” for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.

- **RTM** (restricted transactional memory) flag (bit 16) — Indicates (when clear) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 18.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.

Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

18.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 18-1). The flags and fields in this register control the following things:
• **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6)** — Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated Ln flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.

• **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7)** — Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated Gn flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.

• **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9)** — This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.

• **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 18.3.3). This advanced debugging is enabled only if IA32_DEBUGCTL.RTM is also set.

• **GD (general detect enable) flag (bit 13)** — Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.

• **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29)** — Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the R/Wn fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

00 — Break on instruction execution only.
01 — Break on data writes only.
10 — Break on I/O reads or writes.
11 — Break on data reads or writes but not instruction fetches.

When the DE flag is clear, the processor interprets the R/Wn bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

00 — Break on instruction execution only.
01 — Break on data writes only.
10 — Undefined.
11 — Break on data reads or writes but not instruction fetches.

• **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31)** — Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

00 — 1-byte length.
01 — 2-byte length.
10 — Undefined (or 8 byte length, see note below).
11 — 4-byte length.

If the corresponding RWn field in register DR7 is 00 (instruction execution), then the LENn field should also be 00. The effect of using other lengths is undefined. See Section 18.2.5, “Breakpoint Field Recognition,” below.
NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), breakpoint conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LENn field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). The Encoding 10B is supported in processors based on Intel Atom® microarchitecture, with CPUID signature of family 6, DisplayModel value 1CH. The encoding 10B is undefined for other processors.

18.2.5 Breakpoint Field Recognition

Breakpoint address registers (debug registers DR0 through DR3) and the LENn fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The LENn fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register (DRn). Two-byte ranges must be aligned on word boundaries; 4-byte ranges must be aligned on doubleword boundaries, 8-byte ranges must be aligned on quadword boundaries. I/O addresses are zero-extended (from 16 to 32 bits, for comparison with the breakpoint address in the selected debug register). These requirements are enforced by the processor; it uses LENn field bits to mask the lower address bits in the debug registers. Unaligned data or I/O breakpoint addresses do not yield valid results.

A data breakpoint for reading or writing data is triggered if any of the bytes participating in an access is within the range defined by a breakpoint address register and its LENn field. Table 18-1 provides an example setup of debug registers and data accesses that would subsequently trap or not trap on the breakpoints.

A data breakpoint for an unaligned operand can be constructed using two breakpoints, where each breakpoint is byte-aligned and the two breakpoints together cover the operand. The breakpoints generate exceptions only for the operand, not for neighboring bytes.

Instruction breakpoints addresses must have a length specification of 1 byte (the LENn field is set to 00). Instruction breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

Table 18-1. Breakpoint Examples

<table>
<thead>
<tr>
<th>Debug Register</th>
<th>R/Wn</th>
<th>Breakpoint Address</th>
<th>LENn</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR0</td>
<td>R/W0 = 11</td>
<td>A0001H</td>
<td>LEN0 = 00</td>
</tr>
<tr>
<td></td>
<td>(Read/Write)</td>
<td></td>
<td>(1 byte)</td>
</tr>
<tr>
<td>DR1</td>
<td>R/W1 = 01</td>
<td>A0002H</td>
<td>LEN1 = 00</td>
</tr>
<tr>
<td></td>
<td>(Write)</td>
<td></td>
<td>(1 byte)</td>
</tr>
<tr>
<td>DR2</td>
<td>R/W2 = 11</td>
<td>B0002H</td>
<td>LEN2 = 01</td>
</tr>
<tr>
<td></td>
<td>(Read/Write)</td>
<td></td>
<td>(2 bytes)</td>
</tr>
<tr>
<td>DR3</td>
<td>R/W3 = 01</td>
<td>C0000H</td>
<td>LEN3 = 11</td>
</tr>
<tr>
<td></td>
<td>(Write)</td>
<td></td>
<td>(4 bytes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address</th>
<th>Access Length (In Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Read or write</td>
<td>A0001H</td>
<td>1</td>
</tr>
<tr>
<td>- Read or write</td>
<td>A0001H</td>
<td>2</td>
</tr>
<tr>
<td>- Write</td>
<td>A0002H</td>
<td>1</td>
</tr>
<tr>
<td>- Write</td>
<td>A0002H</td>
<td>2</td>
</tr>
<tr>
<td>- Read or write</td>
<td>B0001H</td>
<td>4</td>
</tr>
<tr>
<td>- Read or write</td>
<td>B0002H</td>
<td>1</td>
</tr>
<tr>
<td>- Read or write</td>
<td>B0002H</td>
<td>2</td>
</tr>
<tr>
<td>- Write</td>
<td>C0000H</td>
<td>4</td>
</tr>
<tr>
<td>- Write</td>
<td>C0001H</td>
<td>2</td>
</tr>
<tr>
<td>- Write</td>
<td>C0003H</td>
<td>1</td>
</tr>
</tbody>
</table>
### 18.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 18-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Breakpoint conditions for 8-byte memory read/writes are supported in all modes.

### 18.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

#### 18.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 18-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 18.3.1.3, “General-Detect Exception Condition”) result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

The INT1 instruction generates a debug exception as a trap. Hardware vendors may use the INT1 instruction for hardware debug. For that reason, Intel recommends software vendors instead use the INT3 instruction for software breakpoints.

See also: Chapter 6, "Interrupt 1—Debug Exception (#DB),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

---

**Table 18-1. Breakpoint Examples (Contd.)**

<table>
<thead>
<tr>
<th>Debug Register</th>
<th>R/Wn</th>
<th>Breakpoint Address</th>
<th>LENn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data operations that do not trap</td>
<td></td>
<td>A0000H</td>
<td>1</td>
</tr>
<tr>
<td>- Read or write</td>
<td></td>
<td>A0002H</td>
<td>1</td>
</tr>
<tr>
<td>- Read</td>
<td></td>
<td>A0003H</td>
<td>4</td>
</tr>
<tr>
<td>- Read or write</td>
<td></td>
<td>B0000H</td>
<td>2</td>
</tr>
<tr>
<td>- Read</td>
<td></td>
<td>C0000H</td>
<td>2</td>
</tr>
<tr>
<td>- Read or write</td>
<td></td>
<td>C0004H</td>
<td>4</td>
</tr>
</tbody>
</table>

---

Vol. 3B 18-7
Figure 18-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture
18.3.1.1 Instruction-Breakpoint Exception Condition

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if an instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint will be suppressed as if EFLAGS.RF were 1 (see the next paragraph and Section 6.8.3, "Masking Exceptions and Interrupts When Switching Stacks," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, "System Flags and Fields in the EFLAGS Register," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for instruction breakpoints, CS limit violations, and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  - Debug exceptions generated in response to instruction breakpoints
  - Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)
18.3.1.1  Debug Exception Conditions

- Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)
- Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

18.3.1.2  Data Memory and I/O Breakpoint Exception Conditions

Data memory and I/O breakpoints are reported when the processor attempts to access a memory or I/O address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect data or I/O accesses (R/W flag is set to 1, 2, or 3). The processor generates the exception after it executes the instruction that made the access, so these breakpoint condition causes a trap-class exception to be generated.

Because data breakpoints are traps, an instruction that writes memory overwrites the original data before the debug exception generated by a data breakpoint is generated. If a debugger needs to save the contents of a write breakpoint location, it should save the original contents before setting the breakpoint. The handler can report the saved value after the breakpoint is triggered. The address in the debug registers can be used to locate the new value stored by the instruction that triggered the breakpoint.

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

Intel486 and later processors ignore the GE and LE flags in DR7. In Intel386 processors, exact data breakpoint matching does not occur unless it is enabled by setting the LE and/or the GE flags.

For repeated INS and OUTS instructions that generate an I/O-breakpoint debug exception, the processor generates the exception after the completion of the first iteration. Repeated INS and OUTS instructions generate a data-breakpoint debug exception after the iteration in which the memory address breakpoint location is accessed.

If an execution of the MOV or POP instruction loads the SS register and encounters a data breakpoint, the resulting debug exception is delivered after completion of the next instruction (the one after the MOV or POP).

Any pending data or I/O breakpoints are lost upon delivery of an exception. For example, if a machine-check exception (#MC) occurs following an instruction that encounters a data breakpoint (but before the resulting debug exception is delivered), the data breakpoint is lost. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, the data breakpoint is lost if the next instruction causes a fault.

Delivery of events due to INT n, INT3, or INTO does not cause a loss of data breakpoints. If a MOV or POP instruction that loads the SS register encounters a data breakpoint, and the next instruction is software interrupt (INT n, INT3, or INTO), a debug exception (#DB) resulting from a data breakpoint will be delivered after the transition to the software-interrupt handler. The #DB handler should account for the fact that the #DB may have been delivered after a invocation of a software-interrupt handler, and in particular that the CPL may have changed between recognition of the data breakpoint and delivery of the #DB.

18.3.1.3  General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required.
debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

18.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT n, INT3, and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT n or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT n instruction that calls the interrupt handler.

If an occurrence of the MOV or POP instruction loads the SS register executes with EFLAGS.TF = 1, no single-step debug exception occurs following the MOV or POP instruction.

18.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task’s TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

18.3.1.6 OS Bus-Lock Detection

OS bus-lock detection is a feature that causes the processor to generate a debug exception (called a bus-lock detection debug exception) if it detects that a bus lock has been asserted (see Section 9.1.2). Such an exception is a trap-class exception, because it is generated after execution of an instruction that asserts a bus lock. The exception thus does not prevent assertion of the bus lock. Delivery of a bus-lock detection debug exception clears DR6.BLD.

Software can enable OS bus-lock detection by setting IA32_DEBUGCTL.BLD[bit 2]. Bus-lock detection debug exceptions occur only if CPL > 0.

18.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT3 instruction. See Chapter 6, "Interrupt 3—Breakpoint Exception (#BP).” Debuggers use breakpoint exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers,
because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.

### 18.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify transactional regions (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a transactional abort. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable advanced debugging of RTM transactional regions by setting DR7.RTM[bit 11] and IA32_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does not resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered. (A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

### 18.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel Atom® processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt, and exception recording:

- Section 18.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel Atom® Processors).”
- Section 18.6, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture.”
- Section 18.9, “Last Branch, Interrupt, and Exception Recording for Processors based on Nehalem Microarchitecture.”
- Section 18.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Sandy Bridge Microarchitecture.”
- Section 18.11, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture.”
- Section 18.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture.”
- Section 18.14, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors).”
- Section 18.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).”
- Section 18.16, “Last Branch, Interrupt, and Exception Recording (P6 Family Processors).”
The following subsections of Section 18.4 describe common features of profiling branches. These features are generally enabled using the IA32_DEBUGCTL MSR (older processor may have implemented a subset or model-specific features, see definitions of MSR_DEBUGCTLA, MSR_DEBUGCTLB, MSR_DEBUGCTL).

18.4.1 IA32_DEBUGCTL MSR

The IA32_DEBUGCTL MSR provides bit field controls to enable debug trace interrupts, debug trace stores, trace messages enable, single stepping on branches, last branch record recording, and to control freezing of LBR stack or performance counters on a PMI request. IA32_DEBUGCTL MSR is located at register address 01D9H.

See Figure 18-3 for the MSR layout and the bullets below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the Section 18.5.1, “LBR Stack” (Intel® Core™2 Duo and Intel Atom® processor family) and Section 18.9.1, “LBR Stack” (processors based on Nehalem microarchitecture).

- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.

- **BLD (bus-lock detection) flag (bit 2)** — If this bit is set, OS bus-lock detection is enabled when CPL > 0. See Section 18.3.1.6.

- **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, “Branch Trace Messages,” for more information about the TR flag.

- **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, “BTS and DS Save Area.”

- **BTINT (branch trace interrupt) flag (bit 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.

![Figure 18-3. IA32_DEBUGCTL MSR for Processors Based on Intel® Core™ Microarchitecture](image-url)
• **BTS_OFF_OS (branch trace off in privileged code) flag (bit 9)** — When set, BTS or BTM is skipped if CPL is 0. See Section 18.13.2.

• **BTS_OFF_USR (branch trace off in user code) flag (bit 10)** — When set, BTS or BTM is skipped if CPL is greater than 0. See Section 18.13.2.

• **FREEZE_LBRS_ON_PMI flag (bit 11)** — When set, the LBR stack is frozen on a hardware PMI request (e.g., when a counter overflows and is configured to trigger PMI). See Section 18.4.7 for details.

• **FREEZE_PERFMON_ON_PMI flag (bit 12)** — When set, the performance counters (IA32_PMCx and IA32_FIXED_CTRx) are frozen on a PMI request. See Section 18.4.7 for details.

• **FREEZE_WHILE_SMM (bit 14)** — If this bit is set, upon the delivery of an SMI, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler. If Intel Thread Director support was enabled before transferring control to the SMI handler, then the processor will also reset the Intel Thread Director history (see Section 15.6.11 for more details about Intel Thread Director enable, reset, and history reset operations).

Subsequently, the enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its service. If Intel Thread Director support is enabled when RSM is executed, then the processor resets the Intel Thread Director history. Note that system software must check if the processor supports the IA32_DEBUGCTL.FREEZE_WHILE_SMM control bit. IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 20.8 for details of detecting the presence of IA32_PERF_CAPABILITIES MSR.

• **RTM (bit 15)** — If this bit is set, advanced debugging of RTM transactional regions is enabled if DR7.RTM is also set. See Section 18.3.3.

### 18.4.2 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag (bit 0) in the IA32_DEBUGCTL MSR is set, the processor automatically begins recording branch records for taken branches, interrupts, and exceptions (except for debug exceptions) in the LBR stack MSRs.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler. This action does not clear previously stored LBR stack MSRs.

A debugger can use the linear addresses in the LBR stack to re-set breakpoints in the breakpoint address registers (DR0 through DR3). This allows a backward trace from the manifestation of a particular bug toward its source.

On some processors, if the LBR flag is cleared and TR flag in the IA32_DEBUGCTL MSR remains set, the processor will continue to update LBR stack MSRs. This is because those processors use the entries in the LBR stack in the process of generating BTM/BTS records. A #DB does not automatically clear the TR flag.

### 18.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.² This mechanism allows a debugger to single-step on control transfers caused by branches. This “branch single stepping” helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

---

² Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 8.2.1, “Task-State Segment (TSS).”
18.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 18.4.2, “Monitoring Branches, Exceptions, and Interrupts.”

For the P6 processor family, Pentium M processor family, and processors based on Intel Core microarchitecture, TR and LBR bits can not be set at the same time due to hardware limitation. The content of LBR stack is undefined when TR is set.

For processors with Intel NetBurst microarchitecture, Intel Atom processors, and Intel Core and related Intel Xeon processors both starting with the Nehalem microarchitecture, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32_DEBUGCTL MSR (or the equivalent MSR_DEBUGCTLA, MSR_DEBUGCTLB).

The following exception applies:
- BTM may not be observable on Intel Atom processor families that do not provide an externally visible system bus (i.e., processors based on the Silvermont microarchitecture or later).

18.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

18.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 18.4.9.2, “Setting Up the DS Save Area,” for additional details.

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

18.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if CPUID.01H:ECX[bit 4] = 1.

The CPL-qualified branch trace mechanism is described in Section 18.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, BTS_OFF_USR (bit 10) and BTS_OFF_OS (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

18.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or
greater (i.e., CPUID.0AH:EAX[7:0] > 1). These capabilities provides the following interface in IA32_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

• **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  
  — Legacy Freeze_LBR_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32_DEBUGCTL. Software must then re-enable IA32_DEBUGCTL.LBR to resume recording branches. When using this feature, software should be careful about writes to IA32_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  
  — Streamlined Freeze_LBR_on_PMI is supported for ArchPerfMonVerID >= 4. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the processor behaves as follows:
    • sets IA32_PERF_GLOBAL_STATUS.LBR_Frz =1 to disable recording, but does not change the LBR bit (bit 0) in IA32_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.

• **Freezing PMCs on PMI (bit 12)** — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  
  — Legacy Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_Perfmon_On_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32_PERF_GLOBAL_CTRL MSR (see Figure 20-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 20.6.2.1, “Fixed-function Performance Counters”). Software must re-enable counts by writing 1s to the corresponding enable bits in IA32_PERF_GLOBAL_CTRL before leaving a PMI service routine to continue counter operation.
  
  — Streamlined Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID >= 4. The processor behaves as follows:
    • sets IA32_PERF_GLOBAL_STATUS.CTR_Frz =1 to disable counting on a counter overflow condition, but does not change the IA32_PERF_GLOBAL_CTRL MSR.

Freezing LBRs and PMCs on PMIs (both legacy and streamlined operation) occur when one of the following applies:

• A performance counter had an overflow and was programmed to signal a PMI in case of an overflow.
  
  — For the general-purpose counters; enabling PMI is done by setting bit 20 of the IA32_PERFEVTSELx register.
  
  — For the fixed-function counters; enabling PMI is done by setting the 3rd bit in the corresponding 4-bit control field of the MSR_PERF_FIXED_CTR_CTRL register (see Figure 20-1) or IA32_FIXED_CTR_CTRL MSR (see Figure 20-2).

• The PEBS buffer is almost full and reaches the interrupt threshold.
• The BTS buffer is almost full and reaches the interrupt threshold.

Table 18-3 compares the interaction of the processor with the PMI handler using the legacy versus streamlined Freeze_Perfmon_On_PMI interface.
Table 18-3. Legacy and Streamlined Operation with Freeze_Perfmon_On_PMI = 1, Counter Overflowed

<table>
<thead>
<tr>
<th>Legacy Freeze_Perfmon_On_PMI</th>
<th>Streamlined Freeze_Perfmon_On_PMI</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor freezes the counters on overflow</td>
<td>Processor freezes the counters on overflow</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Processor clears IA32_PERF_GLOBAL_CTRL</td>
<td>Processor set IA32_PERF_GLOBAL_STATUS.CTR_FTZ</td>
<td></td>
</tr>
<tr>
<td>Handler reads IA32_PERF_GLOBAL_STATUS(0x38E) to examine which counter(s) overflowed</td>
<td><em>mask = RDMSR(0x38E)</em></td>
<td>Similar</td>
</tr>
<tr>
<td>Handler services the PMI</td>
<td>Handler services the PMI</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Handler writes 1s to IA32_PERF_GLOBAL_OVF_CTL(0x390)</td>
<td>Handler writes <em>mask</em> into IA32_PERF_GLOBAL_OVF_RESET(0x390)</td>
<td></td>
</tr>
<tr>
<td>Processor clears IA32_PERF_GLOBAL_STATUS</td>
<td>Processor clears IA32_PERF_GLOBAL_STATUS</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Handler re-enables IA32_PERF_GLOBAL_CTRL</td>
<td>None</td>
<td>Reduced software overhead</td>
</tr>
</tbody>
</table>

### 18.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 18-4 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily_DisplayModel encoding (see the CPUID instruction in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

Table 18-4. LBR Stack Size and TOS Pointer Range

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Size of LBR Stack</th>
<th>Component of an LBR Entry</th>
<th>Range of TOS Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_5CH, 06_5FH</td>
<td>32</td>
<td>FROM_IP, TO_IP Entry</td>
<td>0 to 31</td>
</tr>
<tr>
<td>06_4EH, 06_5EH, 06_8EH, 06_9EH, 06_55H, 06_66H, 06_7AH, 06_67H, 06_6AH, 06_6CH, 06_7DH, 06_7EH, 06_8CH, 06_8DH, 06_6AH, 06_A5H, 06_A6H, 06_A7H, 06_A8H, 06_86H, 06_8AH, 06_96H, 06_9CH</td>
<td>32</td>
<td>FROM_IP, TO_IP, LBR_INFO</td>
<td>0 to 31</td>
</tr>
<tr>
<td>06_3DH, 06_47H, 06_4FH, 06_56H, 06_3CH, 06_45H, 06_46H, 06_3FH, 06_2AH, 06_2DH, 06_3AH, 06_3EH, 06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH</td>
<td>16</td>
<td>FROM_IP, TO_IP Entry</td>
<td>0 to 15</td>
</tr>
<tr>
<td>06_17H, 06_1DH, 06_0FH</td>
<td>4</td>
<td>FROM_IP, TO_IP Entry</td>
<td>0 to 3</td>
</tr>
<tr>
<td>06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH, 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H</td>
<td>8</td>
<td>FROM_IP, TO_IP Entry</td>
<td>0 to 7</td>
</tr>
</tbody>
</table>

**NOTES:**
1. See Section 18.12.
The last branch recording mechanism tracks not only branch instructions (e.g., JMP, Jcc, LOOP, and CALL instructions), but also other operations that cause a change in the instruction pointer (e.g., external interrupts, traps, and faults). The branch recording mechanisms generally employ a set of MSRs, referred to as last branch record (LBR) stack. The size and exact locations of the LBR stack are generally model-specific (see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for model-specific MSR addresses).

- **Last Branch Record (LBR) Stack** — The LBR consists of N pairs of MSRs (N is listed in the LBR stack size column of Table 18-4) that store source and destination address of recent branches (see Figure 18-3):
  - MSR_LASTBRANCH_0_FROM_IP (address is model specific) through the next consecutive (N-1) MSR address store source addresses.
  - MSR_LASTBRANCH_0_TO_IP (address is model specific) through the next consecutive (N-1) MSR address store destination addresses.

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant M bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address is model specific) contains an M-bit pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. The valid range of the M-bit POS pointer is given in Table 18-4.

### 18.4.8.1 LBR Stack and Intel® 64 Processors

LBR MSRs are 64-bits. In 64-bit mode, last branch records store the full address. Outside of 64-bit mode, the upper 32-bits of branch addresses will be stored as 0.

<table>
<thead>
<tr>
<th>MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_(N-1)_FROM_IP</th>
<th>Source Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_(N-1)_TO_IP</th>
<th>Destination Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 18-4. 64-bit Address Layout of LBR MSR**

Software should query an architectural MSR IA32_PERF_CAPABILITIES[5:0] about the format of the address that is stored in the LBR stack. Four formats are defined by the following encoding:

- **000000B (32-bit record format)** — Stores 32-bit offset in current CS of respective source/destination,
- **000001B (64-bit LIP record format)** — Stores 64-bit linear address of respective source/destination,
- **000010B (64-bit EIP record format)** — Stores 64-bit offset (effective address) of respective source/destination.
- **000011B (64-bit EIP record format) and Flags** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction info is reported in the upper bit of ‘FROM’ registers in the LBR stack. See LBR stack details below for flag support and definition.
- **000100B (64-bit EIP record format), Flags, and TSX** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction and TSX info are reported in the upper bits of ‘FROM’ registers in the LBR stack.
- **000101B (64-bit EIP record format), Flags, TSX, and LBR_INFO** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction, TSX, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.
- **000110B (64-bit LIP record format), Flags, and Cycles** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction info is reported in the upper bits of
'FROM' registers in the LBR stack. Elapsed cycles since the last LBR update are reported in the upper 16 bits of the 'TO' registers in the LBR stack (see Section 18.6).

— **000111B (64-bit LIP record format), Flags, and LBR_INFO** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.

Processor’s support for the architectural MSR IA32_PERF_CAPABILITIES is provided by CPUID.01H:ECX[PERF_CAPAB_MSR] (bit 15).

### 18.4.8.2 LBR Stack and IA-32 Processors

The LBR MSRs in IA-32 processors introduced prior to Intel 64 architecture store the 32-bit “To Linear Address” and “From Linear Address” using the high and low half of each 64-bit MSR.

### 18.4.8.3 Last Exception Records and Intel 64 Architecture

Intel 64 and IA-32 processors also provide MSRs that store the branch record for the last branch taken prior to an exception or an interrupt. The location of the last exception record (LER) MSRs are model specific. The MSRs that store last exception records are 64-bits. If IA-32e mode is disabled, only the lower 32-bits of the address is recorded. If IA-32e mode is enabled, the processor writes 64-bit values into the MSR. In 64-bit mode, last exception records store 64-bit addresses; in compatibility mode, the upper 32-bits of last exception records are cleared.

### 18.4.9 BTS and DS Save Area

The **Debug store (DS)** feature flag (bit 21), returned by CPUID.1:EDX[21] indicates that the processor provides the debug store (DS) mechanism. The DS mechanism allows:

- BTMs to be stored in a memory-resident BTS buffer. See Section 18.4.5, “Branch Trace Store (BTS).”
- Processor event-based sampling (PEBS) also uses the DS save area provided by debug store mechanism. The capability of PEBS varies across different microarchitectures. See Section 20.6.2.4, “Processor Event Based Sampling (PEBS),” and the relevant PEBS sub-sections across the core PMU sections in Chapter 20, “Performance Monitoring.”

When CPUID.1:EDX[21] is set:

- The BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags in the IA32_MISC_ENABLE MSR indicate (when clear) the availability of the BTS and PEBS facilities, including the ability to set the BTS and BTINT bits in the appropriate DEBUGCTL MSR.
- The IA32_DS_AREA MSR exists and points to the DS save area.

The debug store (DS) save area is a software-designated area of memory that is used to collect the following two types of information:

- **Branch records** — When the BTS flag in the IA32_DEBUGCTL MSR is set, a branch record is stored in the BTS buffer in the DS save area whenever a taken branch, interrupt, or exception is detected.
- **PEBS records** — When a performance counter is configured for PEBS, a PEBS record is stored in the PEBS buffer in the DS save area after the counter overflow occurs. This record contains the architectural state of the processor (state of the 8 general purpose registers, EIP register, and EFLAGS register) at the next occurrence of the PEBS event that caused the counter to overflow. When the state information has been logged, the counter is automatically reset to a specified value, and event counting begins again. The content layout of a PEBS record varies across different implementations that support PEBS. See Section 20.6.2.4.2 for details of enumerating PEBS record format.
NOTES

Prior to processors based on the Goldmont microarchitecture, PEBS facility only supports a subset of implementation-specific precise events. See Section 20.5.3.1 for a PEBS enhancement that can generate records for both precise and non-precise events.

The DS save area and recording mechanism are disabled on INIT, processor Reset or transition to system-management mode (SMM) or IA-32e mode. It is similarly disabled on the generation of a machine-check exception on 45nm and 32nm Intel Atom processors and on processors with Netburst or Intel Core microarchitecture.

The BTS and PEBS facilities may not be available on all processors. The availability of these facilities is indicated by the BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags, respectively, in the IA32_MISC_ENABLE MSR (see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4).

The DS save area is divided into three parts: buffer management area, branch trace store (BTS) buffer, and PEBS buffer. The processor then uses the buffer management area to keep track of the branch and/or PEBS records in their respective buffers and to record the performance counter reset value. The linear address of the first byte of the DS buffer management area is specified with the IA32_DS_AREA MSR.

The fields in the buffer management area are as follows:

- **BTS buffer base** — Linear address of the first byte of the BTS buffer. This address should point to a natural doubleword boundary.
- **BTS index** — Linear address of the first byte of the next BTS record to be written to. Initially, this address should be the same as the address in the BTS buffer base field.
- **BTS absolute maximum** — Linear address of the next byte past the end of the BTS buffer. This address should be a multiple of the BTS record size (12 bytes) plus 1.
- **BTS interrupt threshold** — Linear address of the BTS record on which an interrupt is to be generated. This address must point to an offset from the BTS buffer base that is a multiple of the BTS record size. Also, it must be several records short of the BTS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the BTS absolute maximum record.
- **PEBS buffer base** — Linear address of the first byte of the PEBS buffer. This address should point to a natural doubleword boundary.
- **PEBS index** — Linear address of the first byte of the next PEBS record to be written to. Initially, this address should be the same as the address in the PEBS buffer base field.
- **PEBS absolute maximum** — Linear address of the next byte past the end of the PEBS buffer. This address should be a multiple of the PEBS record size (40 bytes) plus 1.
- **PEBS interrupt threshold** — Linear address of the PEBS record on which an interrupt is to be generated. This address must point to an offset from the PEBS buffer base that is a multiple of the PEBS record size. Also, it must be several records short of the PEBS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the PEBS absolute maximum record.
- **PEBS counter reset value** — A 64-bit value that the counter is to be set to when a PEBS record is written. Bits beyond the size of the counter are ignored. This value allows state information to be collected regularly every time the specified number of events occur.
Figure 18-6 shows the structure of a 12-byte branch record in the BTS buffer. The fields in each record are as follows:

- **Last branch from** — Linear address of the instruction from which the branch, interrupt, or exception was taken.
- **Last branch to** — Linear address of the branch target or the first instruction in the interrupt or exception service routine.
- **Branch predicted** — Bit 4 of field indicates whether the branch that was taken was predicted (set) or not predicted (clear).
Figure 18-7 shows the structure of the 40-byte PEBS records. Nominally the register values are those at the beginning of the instruction that caused the event. However, there are cases where the registers may be logged in a partially modified state. The linear IP field shows the value in the EIP register translated from an offset into the current code segment to a linear address.

18.4.9.1  64 Bit Format of the DS Save Area

When DTES64 = 1 (CPUID.1.ECX[2] = 1), the structure of the DS save area is shown in Figure 18-8.

When DTES64 = 0 (CPUID.1.ECX[2] = 0) and IA-32e mode is active, the structure of the DS save area is shown in Figure 18-8. If IA-32e mode is not active the structure of the DS save area is as shown in Figure 18-5.
The IA32_DS_AREA MSR holds the 64-bit linear address of the first byte of the DS buffer management area. The structure of a branch trace record is similar to that shown in Figure 18-6, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 18-9). The structure of a PEBS record is similar to that shown in Figure 18-7, but each field is 8 bytes in length and architectural states include register R8 through R15. This makes the size of a PEBS record in 64-bit mode 144 bytes (see Figure 18-10).

NOTES:
1. This example represents the format for a system that supports PEBS on only one counter.

The IA32_DS_AREA MSR holds the 64-bit linear address of the first byte of the DS buffer management area. The structure of a branch trace record is similar to that shown in Figure 18-6, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 18-9). The structure of a PEBS record is similar to that shown in Figure 18-7, but each field is 8 bytes in length and architectural states include register R8 through R15. This makes the size of a PEBS record in 64-bit mode 144 bytes (see Figure 18-10).
Fields in the buffer management area of a DS save area are described in Section 18.4.9.

The format of a branch trace record and a PEBS record are the same as the 64-bit record formats shown in Figures 18-9 and Figures 18-10, with the exception that the branch predicted bit is not supported by Intel Core microarchitecture or Intel Atom microarchitecture. The 64-bit record formats for BTS and PEBS apply to DS save area for all operating modes.

The procedures used to program IA32_DEBUGCTL MSR to set up a BTS buffer or a CPL-qualified BTS are described in Section 18.4.9.3 and Section 18.4.9.4.

Required elements for writing a DS interrupt service routine are largely the same on processors that support using DS Save area for BTS or PEBS records. However, on processors based on Intel NetBurst® microarchitecture, re-enabling counting requires writing to CCCRs. But a DS interrupt service routine on processors supporting architectural performance monitoring should:

- Re-enable the enable bits in IA32_PERF_GLOBAL_CTRL MSR if it is servicing an overflow PMI due to PEBS.
- Clear overflow indications by writing to IA32_PERF_GLOBAL_OVF_CTRL when a counting configuration is changed. This includes bit 62 (ClrOvfBuffer) and the overflow indication of counters used in either PEBS or general-purpose counting (specifically: bits 0 or 1; see Figures 20-3).

18.4.9.2 Setting Up the DS Save Area

To save branch records with the BTS buffer, the DS save area must first be set up in memory as described in the following procedure (See Section 20.6.2.4.1, “Setting up the PEBS Buffer,” for instructions for setting up a PEBS buffer, respectively, in the DS save area):

1. Create the DS buffer management information area in memory (see Section 18.4.9, “BTS and DS Save Area,” and Section 18.4.9.1, “64 Bit Format of the DS Save Area”). Also see the additional notes in this section.

2. Write the base linear address of the DS buffer management area into the IA32_DS_AREA MSR.

3. Set up the performance counter entry in the xAPIC LVT for fixed delivery and edge sensitive. See Section 11.5.1, “Local Vector Table.”

4. Establish an interrupt handler in the IDT for the vector associated with the performance counter entry in the xAPIC LVT.
5. Write an interrupt service routine to handle the interrupt. See Section 18.4.9.5, "Writing the DS Interrupt Service Routine."

The following restrictions should be applied to the DS save area:

- The recording of branch records in the BTS buffer (or PEBS records in the PEBS buffer) may not operate properly if accesses to the linear addresses in any of the three DS save area sections cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do "lazy" page-table entry propagation for these pages. Some newer processor generations support "lazy" EPT page-table entry propagation for PEBS; see Section 20.3.9.1 and Section 20.9.5 for more information. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.

- The DS save area can be larger than a page, but the pages must be mapped to contiguous linear addresses. The buffer may share a page, so it need not be aligned on a 4-KByte boundary. For performance reasons, the base of the buffer must be aligned on a doubleword boundary and should be aligned on a cache line boundary.

- It is recommended that the buffer size for the BTS buffer and the PEBS buffer be an integer multiple of the corresponding record sizes.

- The precise event records buffer should be large enough to hold the number of precise event records that can occur while waiting for the interrupt to be serviced.

- The DS save area should be in kernel space. It must not be on the same page as code, to avoid triggering self-modifying code actions.

- There are no memory type restrictions on the buffers, although it is recommended that the buffers be designated as WB memory type for performance considerations.

- Either the system must be prevented from entering A20M mode while DS save area is active, or bit 20 of all addresses within buffer bounds must be 0.

- Pages that contain buffers must be mapped to the same physical addresses for all processes, such that any change to control register CR3 will not change the DS addresses.

- The DS save area is expected to used only on systems with an enabled APIC. The LVT Performance Counter entry in the APCI must be initialized to use an interrupt gate instead of the trap gate.

### 18.4.9.3 Setting Up the BTS Buffer

Three flags in the MSR_DEBUGCTLA MSR (see Table 18-5), IA32_DEBUGCTL (see Figure 18-3), or MSR_DEBUGCTLB (see Figure 18-16) control the generation of branch records and storing of them in the BTS buffer; these are TR, BTS, and BTINT. The TR flag enables the generation of BTMs. The BTS flag determines whether the BTMs are sent out on the system bus (clear) or stored in the BTS buffer (set). BTMs cannot be simultaneously sent to the system bus and logged in the BTS buffer. The BTINT flag enables the generation of an interrupt when the BTS buffer is full. When this flag is clear, the BTS buffer is a circular buffer.

<table>
<thead>
<tr>
<th>TR</th>
<th>BTS</th>
<th>BTINT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Branch trace messages (BTMs) off</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Generate BTMs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Store BTMs in the BTS buffer, used here as a circular buffer</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Store BTMs in the BTS buffer, and generate an interrupt when the buffer is nearly full</td>
</tr>
</tbody>
</table>

The following procedure describes how to set up a DS Save area to collect branch records in the BTS buffer:

1. Place values in the BTS buffer base, BTS index, BTS absolute maximum, and BTS interrupt threshold fields of the DS buffer management area to set up the BTS buffer in memory.

2. Set the TR and BTS flags in the IA32_DEBUGCTL for Intel Core Solo and Intel Core Duo processors or later processors (or MSR_DEBUGCTLA MSR for processors based on Intel NetBurst Microarchitecture; or MSR_DEBUGCTLB for Pentium M processors).
3. Clear the BTINT flag in the corresponding IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR; or MSR_DEBUGCTLB) if a circular BTS buffer is desired.

**NOTES**

If the buffer size is set to less than the minimum allowable value (i.e., BTS absolute maximum < 1 + size of BTS record), the results of BTS are undefined.

In order to prevent generating an interrupt, when working with circular BTS buffer, SW need to set BTS interrupt threshold to a value greater than BTS absolute maximum (fields of the DS buffer management area). It's not enough to clear the BTINT flag itself only.

### 18.4.9.4 Setting Up CPL-Qualified BTS

If the processor supports CPL-qualified last branch recording mechanism, the generation of branch records and storing of them in the BTS buffer are determined by: TR, BTS, BTS_OFF_OS, BTS_OFF_USR, and BTINT. The encoding of these five bits are shown in Table 18-6.

<table>
<thead>
<tr>
<th>TR</th>
<th>BTS</th>
<th>BTS_OFF_OS</th>
<th>BTS_OFF_USR</th>
<th>BTINT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Branch trace messages (BTMs) off</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td></td>
<td>Generates BTMs but do not store BTMs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Store all BTMs in the BTS buffer, used here as a circular buffer</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Store BTMs with CPL &gt; 0 in the BTS buffer</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Store BTMs with CPL = 0 in the BTS buffer</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Generate BTMs but do not store BTMs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Store BTMs with CPL &gt; 0 in the BTS buffer; generate an interrupt when the buffer is nearly full</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full</td>
</tr>
</tbody>
</table>

### 18.4.9.5 Writing the DS Interrupt Service Routine

The BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector and interrupt service routine (called the debug store interrupt service routine or DS ISR). To handle BTS, non-precise event-based sampling, and PEBS interrupts: separate handler routines must be included in the DS ISR. Use the following guidelines when writing a DS ISR to handle BTS, non-precise event-based sampling, and/or PEBS interrupts.

- The DS interrupt service routine (ISR) must be part of a kernel driver and operate at a current privilege level of 0 to secure the buffer storage area.
- Because the BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector, the DS ISR must check for all the possible causes of interrupts from these facilities and pass control on to the appropriate handler.
- BTS and PEBS buffer overflow would be the sources of the interrupt if the buffer index matches/exceeds the interrupt threshold specified. Detection of non-precise event-based sampling as the source of the interrupt is accomplished by checking for counter overflow.
- There must be separate save areas, buffers, and state for each processor in an MP system.
- Upon entering the ISR, branch trace messages and PEBS should be disabled to prevent race conditions during access to the DS save area. This is done by clearing TR flag in the IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR) and by clearing the precise event enable flag in the MSR_PEBS_ENABLE MSR. These settings should be restored to their original values when exiting the ISR.
• The processor will not disable the DS save area when the buffer is full and the circular mode has not been selected. The current DS setting must be retained and restored by the ISR on exit.

• After reading the data in the appropriate buffer, up to but not including the current index into the buffer, the ISR must reset the buffer index to the beginning of the buffer. Otherwise, everything up to the index will look like new entries upon the next invocation of the ISR.

• The ISR must clear the mask bit in the performance counter LVT entry.

• The ISR must re-enable the counters to count via IA32_PERF_GLOBAL_CTRL/IA32_PERF_GLOBAL_OVF_CTRL if it is servicing an overflow PMI due to PEBS (or via CCCR’s ENABLE bit on processor based on Intel NetBurst microarchitecture).

• The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

18.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL ATOM® PROCESSORS)

The Intel Core 2 Duo processor family and Intel Xeon processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture provide last branch interrupt and exception recording. The facilities described in this section also apply to 45 nm and 32 nm Intel Atom processors. These capabilities are similar to those found in Pentium 4 processors, including support for the following facilities:

• **Debug Trace and Branch Recording Control** — The IA32_DEBUGCTL MSR provide bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 18.4.1 for a description of the flags. See Figure 18-3 for the MSR layout.

• **Last branch record (LBR) stack** — There are a collection of MSR pairs that store the source and destination addresses related to recently executed branches. See Section 18.5.1.

• **Monitoring and single-stepping of branches, exceptions, and interrupts**
  — See Section 18.4.2 and Section 18.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
  — 45 nm and 32 nm Intel Atom processors clear the TR flag when the FREEZE_LBRS_ON_PMI flag is set.

• **Branch trace messages** — See Section 18.4.4.

• **Last exception records** — See Section 18.13.3.

• **Branch trace store and CPL-qualified BTS** — See Section 18.4.5.

• **FREEZE_LBRS_ON_PMI flag (bit 11)** — see Section 18.4.7 for legacy Freeze_LBRS_On_PMI operation.

• **FREEZE_PERFMON_ON_PMI flag (bit 12)** — see Section 18.4.7 for legacy Freeze_Perfmon_On_PMI operation.

• **FREEZE_WHILE_SMM (bit 14)** — FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.4.1.

18.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Atom processor families, and Intel processors based on Intel NetBurst microarchitecture.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 processors families and Intel processors based on Intel NetBurst microarchitecture:

• **Last Branch Record (LBR) Stack**
  — MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_3_FROM_IP (address 43H) store source addresses
  — MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_3_TO_IP (address 63H) store destination addresses
• **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for 45 nm and 32 nm Intel Atom processors:

• **Last Branch Record (LBR) Stack**
  
  — MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_7_FROM_IP (address 47H) store source addresses
  
  — MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_7_TO_IP (address 67H) store destination addresses

• **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 18.4.8.1. The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

### 18.5.2 LBR Stack in Intel Atom® Processors based on the Silvermont Microarchitecture

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in Intel Atom processors based on the Silvermont and Airmont microarchitectures. Eight pairs of MSRs are supported in the LBR stack.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 18-11.

### 18.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR Processors based on Goldmont Microarchitecture

Processors based on the Goldmont microarchitecture extend the capabilities described in Section 18.5.2 with the following enhancements:

• Supports new LBR format encoding 00110b in IA32_PERF_CAPABILITIES[5:0].

• Size of LBR stack increased to 32. Each entry includes MSR_LASTBRANCH_x_FROM_IP (address 0x680..0x69f) and MSR_LASTBRANCH_x_TO_IP (address 0x6c0..0x6df).

• LBR call stack filtering supported. The layout of MSR_LBR_SELECT is described in Table 18-13.

• Elapsed cycle information is added to MSR_LASTBRANCH_x_TO_IP. Format is shown in Table 18-7.

• Misprediction info is reported in the upper bits of MSR_LASTBRANCH_x_FROM_IP. MISPRED bit format is shown in Table 18-8.

• Streamlined Freeze_LBRs_On_PMI operation; see Section 18.12.2.

• LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 18.12.3.

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Bit Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>47:0</td>
<td>R/W</td>
<td>This is the “branch to” address. See Section 18.4.8.1 for address format.</td>
</tr>
<tr>
<td>Cycle Count</td>
<td>63:48</td>
<td>R/W</td>
<td>Elapsed core clocks since last update to the LBR stack.</td>
</tr>
</tbody>
</table>
18.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont Plus microarchitecture. Processors based on the Goldmont Plus microarchitecture extend the capabilities described in Section 18.6 with the following changes:

- Enumeration of new LBR format: encoding 00111b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 18.4.8.1.
- Each LBR stack entry consists of three MSRs:
  - MSR_LastBranch_x_From_IP, the layout is simplified, see Table 18-9.
  - MSR_LastBranch_x_To_IP, the layout is the same as Table 18-9.
  - MSR_LBR_Info_x, stores branch prediction flag, TSX info, and elapsed cycle data. Layout is the same as Table 18-16.

18.8 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in the Intel® Xeon Phi™ processor 7200/5200/3200 series based on the Knights Landing microarchitecture. Eight pairs of MSRs are supported in the LBR stack, per thread:

- **Last Branch Record (LBR) Stack**
  - MSR_LastBranch_0_From_IP (address 680H) through MSR_LastBranch_7_From_IP (address 687H) store source addresses.
  - MSR_LastBranch_0_To_IP (address 6C0H) through MSR_LastBranch_7_To_IP (address 6C7H) store destination addresses.

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LastBranch_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 18-11.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 18.4.8.1. The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

18.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR Processors Based on Nehalem Microarchitecture

The processors based on Nehalem microarchitecture and Westmere microarchitecture support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and add additional capabilities:

- **Debug Trace and Branch Recording Control** — The IA32_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 18.4.1 for a description of the flags. See Figure 18-11 for the MSR layout.

- **Last branch record (LBR) stack** — There are 16 MSRs that store the source and destination addresses related to recently executed branches. See Section 18.9.1.

- **Monitoring and single-stepping of branches, exceptions, and interrupts** — See Section 18.4.2 and Section 18.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
• **Branch trace messages** — The IA32_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 18.4.4. However, not all BTM messages are observable using the Intel® QPI link.

• **Last exception records** — See Section 18.13.3.

• **Branch trace store and CPL-qualified BTS** — See Section 18.4.6 and Section 18.4.5.

• **FREEZE_LBRS_ON_PMI flag (bit 11)** — see Section 18.4.7 for legacy Freeze_LBRs_On_PMI operation.

• **FREEZE_PERFMON_ON_PMI flag (bit 12)** — see Section 18.4.7 for legacy Freeze_Perfmon_On_PMI operation.

• **UNCORE_PMI_EN (bit 13)** — When set, this logical processor is enabled to receive a counter overflow interrupt form the uncore.

• **FREEZE_WHILE_SMM (bit 14)** — FREEZE_WHILE_SMM is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.4.1.

Processors based on Nehalem microarchitecture provide additional capabilities:

• **Independent control of uncore PMI** — The IA32_DEBUGCTL MSR provides a bit field (see Figure 18-11) for software to enable each logical processor to receive an uncore counter overflow interrupt.

• **LBR filtering** — Processors based on Nehalem microarchitecture support filtering of LBR based on combination of CPL and branch type conditions. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT.

![Figure 18-11. IA32_DEBUGCTL MSR for Processors Based on Nehalem Microarchitecture](image)

### 18.9.1 LBR Stack

Processors based on Nehalem microarchitecture provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 18-8 and Table 18-9.

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Bit Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>47:0</td>
<td>R/W</td>
<td>This is the “branch from” address. See Section 18.4.8.1 for address format.</td>
</tr>
<tr>
<td>SIGN_EXt</td>
<td>62:48</td>
<td>R/W</td>
<td>Signed extension of bit 47 of this register.</td>
</tr>
<tr>
<td>MISPRED</td>
<td>63</td>
<td>R/W</td>
<td>When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.</td>
</tr>
</tbody>
</table>
Processors based on Nehalem microarchitecture have an LBR MSR Stack as shown in Table 18-10.

### Table 18-10. LBR Stack Size and TOS Pointer Range

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Size of LBR Stack</th>
<th>Range of TOS Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_1AH</td>
<td>16</td>
<td>0 to 15</td>
</tr>
</tbody>
</table>

### 18.9.2 Filtering of Last Branch Records

MSR_LBR_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e., all branches will be captured. MSR_LBR_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR_LBR_SELECT is shown in Table 18-11.

### Table 18-11. MSR_LBR_SELECT for Nehalem Microarchitecture

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Bit Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPL_EQ_0</td>
<td>0</td>
<td>R/W</td>
<td>When set, do not capture branches ending in ring 0</td>
</tr>
<tr>
<td>CPL_NEQ_0</td>
<td>1</td>
<td>R/W</td>
<td>When set, do not capture branches ending in ring &gt;0</td>
</tr>
<tr>
<td>JCC</td>
<td>2</td>
<td>R/W</td>
<td>When set, do not capture conditional branches</td>
</tr>
<tr>
<td>NEAR_REL_CALL</td>
<td>3</td>
<td>R/W</td>
<td>When set, do not capture near relative calls</td>
</tr>
<tr>
<td>NEAR_IND_CALL</td>
<td>4</td>
<td>R/W</td>
<td>When set, do not capture near indirect calls</td>
</tr>
<tr>
<td>NEAR_RET</td>
<td>5</td>
<td>R/W</td>
<td>When set, do not capture near returns</td>
</tr>
<tr>
<td>NEAR_IND_JMP</td>
<td>6</td>
<td>R/W</td>
<td>When set, do not capture near indirect jumps</td>
</tr>
<tr>
<td>NEAR_REL_JMP</td>
<td>7</td>
<td>R/W</td>
<td>When set, do not capture near relative jumps</td>
</tr>
<tr>
<td>FAR_BRANCH</td>
<td>8</td>
<td>R/W</td>
<td>When set, do not capture far branches</td>
</tr>
<tr>
<td>Reserved</td>
<td>63:9</td>
<td></td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

### 18.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SANDY BRIDGE MICROARCHITECTURE

Generally, all of the last branch record, interrupt, and exception recording facility described in Section 18.9, "Last Branch, Interrupt, and Exception Recording for Processors based on Nehalem Microarchitecture," apply to processors based on Sandy Bridge microarchitecture. For processors based on Ivy Bridge microarchitecture, the same holds true.

One difference of note is that MSR_LBR_SELECT is shared between two logical processors in the same core. In Sandy Bridge microarchitecture, each logical processor has its own MSR_LBR_SELECT. The filtering semantics for "Near_ind_jmp" and "Near_rel_jmp" has been enhanced, see Table 18-12.
18.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE

Generally, all of the last branch record, interrupt, and exception recording facility described in Section 18.10, “Last Branch, Interrupt, and Exception Recording for Processors based on Sandy Bridge Microarchitecture,” apply to next generation processors based on Haswell microarchitecture.

The LBR facility also supports an alternate capability to profile call stack profiles. Configuring the LBR facility to conduct call stack profiling is by writing 1 to the MSR_LBR_SELECT.EN_CALLSTACK[bit 9]; see Table 18-13. If MSR_LBR_SELECT.EN_CALLSTACK is clear, the LBR facility will capture branches normally as described in Section 18.10.

Table 18-13. MSR_LBR_SELECT for Haswell Microarchitecture

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Bit Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPL_EQ_0</td>
<td>0</td>
<td>R/W</td>
<td>When set, do not capture branches ending in ring 0</td>
</tr>
<tr>
<td>CPL_NEQ_0</td>
<td>1</td>
<td>R/W</td>
<td>When set, do not capture branches ending in ring &gt;0</td>
</tr>
<tr>
<td>JCC</td>
<td>2</td>
<td>R/W</td>
<td>When set, do not capture conditional branches</td>
</tr>
<tr>
<td>NEAR_REL_CALL</td>
<td>3</td>
<td>R/W</td>
<td>When set, do not capture near relative calls</td>
</tr>
<tr>
<td>NEAR_IND_CALL</td>
<td>4</td>
<td>R/W</td>
<td>When set, do not capture near indirect calls</td>
</tr>
<tr>
<td>NEAR_RET</td>
<td>5</td>
<td>R/W</td>
<td>When set, do not capture near returns</td>
</tr>
<tr>
<td>NEAR_IND_JMP</td>
<td>6</td>
<td>R/W</td>
<td>When set, do not capture near indirect jumps except near indirect calls and near returns</td>
</tr>
<tr>
<td>NEAR_REL_JMP</td>
<td>7</td>
<td>R/W</td>
<td>When set, do not capture near relative jumps except near relative calls.</td>
</tr>
<tr>
<td>FAR_BRANCH</td>
<td>8</td>
<td>R/W</td>
<td>When set, do not capture far branches</td>
</tr>
<tr>
<td>Reserved</td>
<td>63:9</td>
<td></td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

NOTES:
1. Must set valid combination of bits 0-8 in conjunction with bit 9 (as described below), otherwise the contents of the LBR MSRs are undefined.

The call stack profiling capability is an enhancement of the LBR facility. The LBR stack is a ring buffer typically used to profile control flow transitions resulting from branches. However, the finite depth of the LBR stack often become less effective when profiling certain high-level languages (e.g., C++), where a transition of the execution flow is accompanied by a large number of leaf function calls, each of which returns an individual parameter to form the list
of parameters for the main execution function call. A long list of such parameters returned by the leaf functions would serve to flush the data captured in the LBR stack, often losing the main execution context.

When the call stack feature is enabled, the LBR stack will capture unfiltered call data normally, but as return instructions are executed the last captured branch record is flushed from the on-chip registers in a last-in first-out (LIFO) manner. Thus, branch information relative to leaf functions will not be captured, while preserving the call stack information of the main line execution path.

The configuration of the call stack facility is summarized below:

- Set IA32_DEBUGCTL.LBR (bit 0) to enable the LBR stack to capture branch records. The source and target addresses of the call branches will be captured in the 16 pairs of From/To LBR MSRs that form the LBR stack.
- Program the Top of Stack (TOS) MSR that points to the last valid from/to pair. This register is incremented by 1, modulo 16, before recording the next pair of addresses.
- Program the branch filtering bits of MSR_LBR_SELECT (bits 0:8) as desired.
- Program the MSR_LBR_SELECT to enable LIFO filtering of return instructions with:
  - The following bits in MSR_LBR_SELECT must be set to ‘1’: JCC, NEAR_IND_JMP, NEAR_REL_JMP, FAR_BRANCH, EN_CALLSTACK;
  - The following bits in MSR_LBR_SELECT must be cleared: NEAR_REL_CALL, NEAR_IND_CALL, NEAR_RET;
  - At most one of CPL_EQ_0, CPL_NEQ_0 is set.

Note that when call stack profiling is enabled, “zero length calls” are excluded from writing into the LBRs. (A “zero length call” uses the attribute of the call instruction to push the immediate instruction pointer on to the stack and then pops off that address into a register. This is accomplished without any matching return on the call.)

### 18.11.1 LBR Stack Enhancement

Processors based on Haswell microarchitecture provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is enumerated by IA32_PERF_CAPABILITIES[5:0] = 04H, and is shown in Table 18-14 and Table 18-9.

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Bit Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>47:0</td>
<td>R/w</td>
<td>This is the “branch from” address. See Section 18.4.8.1 for address format.</td>
</tr>
<tr>
<td>SIGN_EXT</td>
<td>60:48</td>
<td>R/w</td>
<td>Signed extension of bit 47 of this register.</td>
</tr>
<tr>
<td>TSX_ABORT</td>
<td>61</td>
<td>R/w</td>
<td>When set, indicates a TSX Abort entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LBR_FROM: EIP at the time of the TSX Abort</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LBR_TO: EIP of the start of HLE region, or EIP of the RTM Abort Handler</td>
</tr>
<tr>
<td>IN_TSX</td>
<td>62</td>
<td>R/w</td>
<td>When set, indicates the entry occurred in a TSX region</td>
</tr>
<tr>
<td>MISPRED</td>
<td>63</td>
<td>R/w</td>
<td>When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.</td>
</tr>
</tbody>
</table>

### 18.12 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SKYLAKE MICROARCHITECTURE

Processors based on the Skylake microarchitecture provide a number of enhancement with storing last branch records:

- enumeration of new LBR format: encoding 00101b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 18.4.8.1.
- Each LBR stack entry consists of a triplets of MSRs:
--- MSR_LASTBRANCH_x_FROM_IP, the layout is simplified, see Table 18-9.
--- MSR_LASTBRANCH_x_TO_IP, the layout is the same as Table 18-9.
--- MSR_LBR_INFO_x, stores branch prediction flag, TSX info, and elapsed cycle data.
• Size of LBR stack increased to 32.

Processors based on the Skylake microarchitecture support the same LBR filtering capabilities as described in Table 18-13.

### Table 18-15. LBR Stack Size and TOS Pointer Range

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Size of LBR Stack</th>
<th>Range of TOS Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_4EH, 06_5EH</td>
<td>32</td>
<td>0 to 31</td>
</tr>
</tbody>
</table>

### 18.12.1 MSR_LBR_INFO_x MSR

The layout of each MSR_LBR_INFO_x MSR is shown in Table 18-16.

### Table 18-16. MSR_LBR_INFO_x

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Bit Offset</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle Count</td>
<td>15:0</td>
<td>R/W</td>
<td>Elapsed core clocks since last update to the LBR stack.</td>
</tr>
<tr>
<td>Reserved</td>
<td>60:16</td>
<td>R/W</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
| TSX_ABORT     | 61         | R/W    | When set, indicates a TSX Abort entry  
                        LBR_FROM: EIP at the time of the TSX Abort  
                        LBR_TO: EIP of the start of HLE region OR  
                        EIP of the RTM Abort Handler |
| IN_TSX        | 62         | R/W    | When set, indicates the entry occurred in a TSX region. |
| MISPRED       | 63         | R/W    | When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted. |

### 18.12.2 Streamlined Freeze_LBRs_On_PMI Operation

The FREEZE_LBRs_ON_PMI feature causes the LBRs to be frozen on a hardware request for a PMI. This prevents the LBRs from being overwritten by new branches, allowing the PMI handler to examine the control flow that preceded the PMI generation. Architectural performance monitoring version 4 and above supports a streamlined FREEZE_LBRs_ON_PMI operation for PMI service routine that replaces the legacy FREEZE_LBRs_ON_PMI operation (see Section 18.4.7).

While the legacy FREEZE_LBRs_ON_PMI clear the LBR bit in the IA32_DEBUGCTL MSR on a PMI request, the streamlined FREEZE_LBRs_ON_PMI will set the LBR_FRZ bit in IA32_PERF_GLOBAL_STATUS. Branches will not cause the LBRs to be updated when LBR_FRZ is set. Software can clear LBR_FRZ at the same time as it clears overflow bits by setting the LBR_FRZ bit as well as the needed overflow bit when writing to IA32_PERF_GLOBAL_STATUS_RESET MSR.

This streamlined behavior avoids race conditions between software and processor writes to IA32_DEBUGCTL that are possible with FREEZE_LBRs_ON_PMI clearing of the LBR enable.
18.12.3 LBR Behavior and Deep C-State

When MWAIT is used to request a C-state that is numerically higher than C1, then LBR state may be initialized to zero depending on optimized "waiting" state that is selected by the processor. The affected LBR states include the FROM, TO, INFO, LAST_BRANCH, LER, and LBR_TOS registers. The LBR enable bit and LBR_FROZEN bit are not affected. The LBR-time of the first LBR record inserted after an exit from such a C-state request will be zero.

18.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

Pentium 4 and Intel Xeon processors based on Intel NetBurst microarchitecture provide the following methods for recording taken branches, interrupts, and exceptions:

- Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consists of a branch-from and a branch-to instruction address.
- Send the branch records out on the system bus as branch trace messages (BTMs).
- Log BTMs in a memory-resident branch trace store (BTS) buffer.

To support these functions, the processor provides the following MSRs and related facilities:

- **MSR_DEBUGCTLA MSR** — Enables last branch, interrupt, and exception recording; single-stepping on taken branches; branch trace messages (BTMs); and branch trace store (BTS). This register is named DebugCtIMSR in the P6 family processors.
- **Debug store (DS) feature flag (CPUID.1:EDX.DS[bit 21])** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer.
- **CPL-qualified debug store (DS) feature flag (CPUID.1:ECX.DS-CPL[bit 4])** — Indicates that the processor provides a CPL-qualified debug store (DS) mechanism, which allows software to selectively skip sending and storing BTMs, according to specified current privilege level settings, into a memory-resident BTS buffer.
- **IA32_MISC_ENABLE MSR** — Indicates that the processor provides the BTS facilities.
- **Last branch record (LBR) stack** — The LBR stack is a circular stack that consists of four MSRs (MSR_LAST_BRANCH_0 through MSR_LASTBRANCH_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_15_FROM_IP and MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_15_TO_IP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- **Last branch record top-of-stack (TOS) pointer** — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H]. See also: Table 18-17, Figure 18-12, and Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”
- **Last exception record** — See Section 18.13.3, “Last Exception Records.”

18.13.1 MSR_DEBUGCTLA MSR

The MSR_DEBUGCTLA MSR enables and disables the various last branch recording mechanisms described in the previous section. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 18-12 shows the flags in the MSR_DEBUGCTLA MSR. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. Each branch, interrupt, or exception is recorded as a 64-bit branch record. The processor clears this flag whenever a debug exception is generated (for example,
when an instruction or data breakpoint or a single-step trap occurs). See Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, “Single-Stepping on Branches.”

- **TR (trace message enable) flag (bit 2)** — When set, branch trace messages are enabled. Thereafter, when the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, “Branch Trace Messages.”

- **BTS (branch trace store) flag (bit 3)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.5, “Branch Trace Store (BTS).”

- **BTINT (branch trace interrupt) flag (bits 4)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, “Branch Trace Store (BTS).”

- **BTS_OFF_OS (disable ring 0 branch trace store) flag (bit 5)** — When set, enables the BTS facilities to skip sending/logging CPL_0 BTMs to the memory-resident BTS buffer. See Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

- **BTS_OFF_USR (disable ring 0 branch trace store) flag (bit 6)** — When set, enables the BTS facilities to skip sending/logging non-CPL_0 BTMs to the memory-resident BTS buffer. See Section 18.13.2, “LBR Stack for Processors Based on Intel NetBurst® Microarchitecture.”

**NOTE**

The initial implementation of BTS_OFF_USR and BTS_OFF_OS in MSR_DEBUGCTLA is shown in Figure 18-12. The BTS_OFF_USR and BTS_OFF_OS fields may be implemented on other model-specific debug control register at different locations.

See Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for a detailed description of each of the last branch recording MSRs.

### 18.13.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture

The LBR stack is made up of LBR MSRs that are treated by the processor as a circular stack. The TOS pointer (MSR_LASTBRANCH_TOS MSR) points to the LBR MSR (or LBR MSR pair) that contains the most recent (last) branch record placed on the stack. Prior to placing a new branch record on the stack, the TOS is incremented by 1. When the TOS pointer reaches its maximum value, it wraps around to 0. See Table 18-17 and Figure 18-12.
The registers in the LBR MSR stack and the MSR_LASTBRANCH_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 18-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the "from" and "to" instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the "from" address is the address of the branch instruction and the "to" address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the "from" address the return instruction pointer (RIP) saved for the interrupt and the "to" address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the "from" address is the linear address of the instruction that caused the exception to be generated and the "to" address is the address of the first instruction in the exception handler routine.

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

### 18.13.3 Last Exception Records

The Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel Atom® processors provide two MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.
The MSR_LER_TO_LIP and MSR_LER_FROM_LIP MSRs contain a branch record for the last branch that the processor took prior to an exception or interrupt being generated.

### 18.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

Intel Core Solo and Intel Core Duo processors provide last branch interrupt and exception recording. This capability is almost identical to that found in Pentium 4 and Intel Xeon processors. There are differences in the stack and in some MSR names and locations.

Note the following:

- **IA32_DEBUGCTL MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. IA32_DEBUGCTL MSR is located at register address 01D9H.

  See Figure 18-14 for the layout and the entries below for a description of the flags:

  - **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" below.

  - **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, "Single-Stepping on Branches," for more information about the BTF flag.

  - **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, "Branch Trace Messages," for more information about the TR flag.

  - **BTS (branch trace store) flag (bit 7)** — When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, "BTS and DS Save Area."

  - **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

![Figure 18-14. IA32_DEBUGCTL MSR for Intel® Core™ Solo and Intel® Core™ Duo Processors](Figure 18-14. IA32_DEBUGCTL MSR for Intel® Core™ Solo and Intel® Core™ Duo Processors)

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 18.4.5, "Branch Trace Store (BTS)."

- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address (MSR addresses start at 40H). See Figure 18-15.
• **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Intel Core Solo and Intel Core Duo processors, this MSR is located at register address 01C9H.

For compatibility, the Intel Core Solo and Intel Core Duo processors provide two 32-bit MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate functions of the LastExceptionToIP and Last-ExceptionFromIP MSRs found in P6 family processors.

For details, see Section 18.12, “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture,” and Section 2.20, “MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

---

18.15 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS)

Like the Pentium 4 and Intel Xeon processor family, Pentium M processors provide last branch interrupt and exception recording. The capability operates almost identically to that found in Pentium 4 and Intel Xeon processors. There are differences in the shape of the stack and in some MSR names and locations. Note the following:

- **MSR_DEBUGCTLB MSR** — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. For Pentium M processors, this MSR is located at register address 01D9H. See Figure 18-16 and the entries below for a description of the flags.
  - **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the “Last Branch Record (LBR) Stack” bullet below.
  - **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag rather than a “single-step on instructions” flag. This mechanism allows single-stepping the processor on taken branches. See Section 18.4.3, “Single-Stepping on Branches,” for more information about the BTF flag.
  - **PBi (performance monitoring/breakpoint pins) flags (bits 5-2)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BPi# pin when a breakpoint match occurs. When a PBi flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6)** — When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 18.4.4, “Branch Trace Messages,” for more information about the TR flag.
  - **BTS (branch trace store) flag (bit 7)** — When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 18.4.9, “BTS and DS Save Area.”
  - **BTINT (branch trace interrupt) flag (bits 8)** — When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 18.4.5, “Branch Trace Store (BTS),” for a description of this mechanism.
• **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 18.4.5, “Branch Trace Store (BTS).”

• **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the ‘from’ address, bits 63-32 hold the ‘to’ address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 18-17.

• **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.

For more detail on these capabilities, see Section 18.13.3, “Last Exception Records,” and Section 2.21, “MSRs In the Pentium M Processor,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

### 18.16 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (P6 FAMILY PROCESSORS)

The P6 family processors provide five MSRs for recording the last branch, interrupt, or exception taken by the processor: DEBUGCTLMSR, LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP. These registers can be used to collect last branch records, to set breakpoints on branches, interrupts, and exceptions, and to single-step from one branch to the next.

See Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for a detailed description of each of the last branch recording MSRs.

#### 18.16.1 DEBUGCTLMSR Register

The version of the DEBUGCTLMSR register found in the P6 family processors enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode.
A protected-mode operating system procedure is required to provide user access to this register. Figure 18-18 shows the flags in the DEBUGCTLMSR register for the P6 family processors. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0)** — When set, the processor records the source and target addresses (in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs) for the last branch and the last exception or interrupt taken by the processor prior to a debug exception being generated. The processor clears this flag whenever a debug exception, such as an instruction or data breakpoint or single-step trap occurs.

- **BTF (single-step on branches) flag (bit 1)** — When set, the processor treats the TF flag in the EFLAGS register as a “single-step on branches” flag. See Section 18.4.3, “Single-Stepping on Branches.”

- **PBi (performance monitoring/breakpoint pins) flags (bits 2 through 5)** — When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP# pin when a breakpoint match occurs. When a PBi flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.

- **TR (trace message enable) flag (bit 6)** — When set, trace messages are enabled as described in Section 18.4.4, “Branch Trace Messages.” Setting this flag greatly reduces the performance of the processor. When trace messages are enabled, the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are undefined.

### 18.16.2 Last Branch and Last Exception MSRs

The LastBranchToIP and LastBranchFromIP MSRs are 32-bit registers for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. When a branch occurs, the processor loads the address of the branch instruction into the LastBranchFromIP MSR and loads the target address for the branch into the LastBranchToIP MSR.

When an interrupt or exception occurs (other than a debug exception), the address of the instruction that was interrupted by the exception or interrupt is loaded into the LastBranchFromIP MSR and the address of the exception or interrupt handler that is called is loaded into the LastBranchToIP MSR.

The LastExceptionToIP and LastExceptionFromIP MSRs (also 32-bit registers) record the instruction pointers for the last branch that the processor took prior to an exception or interrupt being generated. When an exception or interrupt occurs, the contents of the LastBranchToIP and LastBranchFromIP MSRs are copied into these registers before the to and from addresses of the exception or interrupt are recorded in the LastBranchToIP and LastBranchFromIP MSRs.

These registers can be read using the RDMSR instruction.

Note that the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into the current code segment, as opposed to linear addresses, which are saved in last branch records for the Pentium 4 and Intel Xeon processors.
18.16.3 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag in the DEBUGCTLMRSR register is set, the processor automatically begins recording branches that it takes, exceptions that are generated (except for debug exceptions), and interrupts that are serviced. Each time a branch, exception, or interrupt occurs, the processor records the to and from instruction pointers in the LastBranchToIP and LastBranchFromIP MSRs. In addition, for interrupts and exceptions, the processor copies the contents of the LastBranchToIP and LastBranchFromIP MSRs into the LastExceptionToIP and LastExceptionFromIP MSRs prior to recording the to and from addresses of the interrupt or exception.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler, but does not touch the last branch and last exception MSRs. The addresses for the last branch, interrupt, or exception taken are thus retained in the LastBranchToIP and LastBranchFromIP MSRs and the addresses of the last branch prior to an interrupt or exception are retained in the LastExceptionToIP, and LastExceptionFromIP MSRs.

The debugger can use the last branch, interrupt, and/or exception addresses in combination with code-segment selectors retrieved from the stack to reset breakpoints in the breakpoint-address registers (DR0 through DR3), allowing a backward trace from the manifestation of a particular bug toward its source. Because the instruction pointers recorded in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into a code segment, software must determine the segment base address of the code segment associated with the control transfer to calculate the linear address to be placed in the breakpoint-address registers. The segment base address can be determined by reading the segment selector for the code segment from the stack and using it to locate the segment descriptor for the segment in the GDT or LDT. The segment base address can then be read from the segment descriptor.

Before resuming program execution from a debug-exception handler, the handler must set the LBR flag again to re-enable last branch and last exception/interrupt recording.

18.17 TIME-STAMP COUNTER

The Intel 64 and IA-32 architectures (beginning with the Pentium processor) define a time-stamp counter mechanism that can be used to monitor and identify the relative time occurrence of processor events. The counter’s architecture includes the following components:

- **TSC flag** — A feature bit that indicates the availability of the time-stamp counter. The counter is available in an if the function CPUID.1:EDX.TSC[bit 4] = 1.

- **IA32_TIME_STAMP_COUNTER MSR** (called TSC MSR in P6 family and Pentium processors) — The MSR used as the counter.

- **RDTSC instruction** — An instruction used to read the time-stamp counter.

- **TSD flag** — A control register flag is used to enable or disable the time-stamp counter (enabled if CR4.TSD[bit 2] = 1).

The time-stamp counter (as implemented in the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter increments even when the processor is halted by the HLT instruction or the external STPCLK# pin. Note that the assertion of the external DPSLP# pin may cause the time-stamp counter to stop.

Processor families increment the time-stamp counter differently:

- For Pentium M processors (family [06H], models [09H, 0DH]); for Pentium 4 processors, Intel Xeon processors (family [0FH], models [00H, 01H, or 02H]); and for P6 family processors: the time-stamp counter increments with every internal processor clock cycle.

- For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the...
processor or may be set by the maximum resolved frequency at which the processor is booted. The maximum resolved frequency may differ from the processor base frequency, see Section 20.7.2 for more detail. On certain processors, the TSC frequency may not be the same as the frequency in the brand string.

The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. This is the architectural behavior moving forward.

**NOTE**

To determine average processor clock frequency, Intel recommends the use of performance monitoring logic to count processor core clocks over the period of time for which the average is required. See Section 20.6.4.5, “Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture,” and [https://perfmon-events.intel.com/](https://perfmon-events.intel.com/) for more information.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound. Intel guarantees that the time-stamp counter will not wraparound within 10 years after being reset. The period for counter wrap is longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag allows use of this instruction to be restricted to programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions read and write the time-stamp counter, treating the time-stamp counter as an ordinary MSR (address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter are read using RDMSR (just as with RDTSC). When WRMSR is used to write the time-stamp counter on processors before family [0FH], models [03H, 04H]: only the low-order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family [0FH], models [03H, 04H, 06H]; for family [06H], model [0EH, 0FH]; for family [06H], DisplayModel [17H, 1AH, 1CH, 1DH]: all 64 bits are writable.

### 18.17.1 Invariant TSC

The time stamp counter in newer processors may support an enhancement, referred to as invariant TSC. Processor’s support for invariant TSC is indicated by CPUID.80000007H:EDX[8].

The invariant TSC will run at a constant rate in all ACPI P-, C-, and T-states. This is the architectural behavior moving forward. On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource.

### 18.17.2 IA32_TSC_AUX Register and RDTSCP Support

Processors based on Nehalem microarchitecture provide an auxiliary TSC register, IA32_TSC_AUX that is designed to be used in conjunction with IA32_TSC. IA32_TSC_AUX provides a 32-bit field that is initialized by privileged software with a signature value (for example, a logical processor ID).

The primary usage of IA32_TSC_AUX in conjunction with IA32_TSC is to allow software to read the 64-bit time stamp in IA32_TSC and signature value in IA32_TSC_AUX with the instruction RDTSCP in an atomic operation. RDTSCP returns the 64-bit time stamp in EDX:EAX and the 32-bit TSC_AUX signature value in ECX. The atomicity of RDTSCP ensures that no context switch can occur between the reads of the TSC and TSC_AUX values.

Support for RDTSCP is indicated by CPUID.80000001H:EDX[27]. As with RDTSC instruction, non-ring 0 access is controlled by CR4.TSD (Time Stamp Disable flag).
User mode software can use RDTSCP to detect if CPU migration has occurred between successive reads of the TSC. It can also be used to adjust for per-CPU differences in TSC values in a NUMA system.

### 18.17.3 Time-Stamp Counter Adjustment

Software can modify the value of the time-stamp counter (TSC) of a logical processor by using the WRMSR instruction to write to the IA32\_TIME\_STAMP\_COUNTER MSR (address 10H). Because such a write applies only to that logical processor, software seeking to synchronize the TSC values of multiple logical processors must perform these writes on each logical processor. It may be difficult for software to do this in a way that ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32\_TSC\_ADJUST MSR (address 3BH). Like the IA32\_TIME\_STAMP\_COUNTER MSR, the IA32\_TSC\_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32\_TSC\_ADJUST MSR as follows:

- On RESET, the value of the IA32\_TSC\_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32\_TIME\_STAMP\_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32\_TSC\_ADJUST MSR.
- If an execution of WRMSR to the IA32\_TSC\_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32\_TSC\_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32\_TIME\_STAMP\_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32\_TSC\_ADJUST MSR on each logical processor.

Processor support for the IA32\_TSC\_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC\_ADJUST (bit 1).

### 18.17.4 Invariant Time-Keeping

The invariant TSC is based on the invariant timekeeping hardware (called Always Running Timer or ART), that runs at the core crystal clock frequency. The ratio defined by CPUID leaf 15H expresses the frequency relationship between the ART hardware and TSC.

If CPUID.15H:EBX[31:0] != 0 and CPUID.80000007H:EDX[InvariantTSC] = 1, the following linearity relationship holds between TSC and the ART hardware:

\[
\text{TSC\_Value} = \left( \frac{\text{ART\_Value} \times \text{CPUID.15H:EBX[31:0]}}{\text{CPUID.15H:EAX[31:0]}} \right) + K
\]

Where 'K' is an offset that can be adjusted by a privileged agent\(^3\).

When ART hardware is reset, both invariant TSC and K are also reset.

### 18.18 INTEL\textsuperscript{®} RESOURCE DIRECTOR TECHNOLOGY (INTEL\textsuperscript{®} RDT) MONITORING FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of monitoring capabilities including Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM). The Intel\textsuperscript{®} Xeon\textsuperscript{®} processor E5 v3 family introduced resource monitoring capability in each logical processor to measure specific platform shared resource metrics, for example, L3 cache occupancy. The programming interface for these monitoring features is described in this section. Two features within the monitoring feature set provided are described - Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

---

3. IA32\_TSC\_ADJUST MSR and the TSC-offset field in the VM execution controls of VMCS are some of the common interfaces that privileged software can use to manage the time stamp counter for keeping time
Cache Monitoring Technology (CMT) allows an Operating System, Hypervisor or similar system management agent to determine the usage of cache by applications running on the platform. The initial implementation is directed at L3 cache monitoring (currently the last level cache in most server platforms).

Memory Bandwidth Monitoring (MBM), introduced in the Intel® Xeon® processor E5 v4 family, builds on the CMT infrastructure to allow monitoring of bandwidth from one level of the cache hierarchy to the next - in this case focusing on the L3 cache, which is typically backed directly by system memory. As a result of this implementation, memory bandwidth can be monitored.

The monitoring mechanisms described provide the following key shared infrastructure features:

- A mechanism to enumerate the presence of the monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for the OS or Hypervisor to indicate a software-defined ID for each of the software threads (applications, virtual machines, etc.) that are scheduled to run on a logical processor. These identifiers are known as Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for the OS or Hypervisor to read back the collected metrics such as L3 occupancy or Memory Bandwidth for a given software ID at any point during runtime.

### 18.18.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring

The shared resource monitoring features described in this chapter provide a layer of abstraction between applications and logical processors through the use of Resource Monitoring IDs (RMIDs). Each logical processor in the system can be assigned an RMID independently, or multiple logical processors can be assigned to the same RMID value (e.g., to track an application with multiple threads). For each logical processor, only one RMID value is active at a time. This is enforced by the IA32_PQR_ASSOC MSR, which specifies the active RMID of a logical processor. Writing to this MSR by software changes the active RMID of the logical processor from an old value to a new value.

The underlying platform shared resource monitoring hardware tracks cache metrics such as cache utilization and misses as a result of memory accesses according to the RMIDs and reports monitored data via a counter register (IA32_QM_CTR). The specific event types supported vary by generation and can be enumerated via CPUID. To read back monitored data, software configures an event selection MSR (IA32_QM_EVTSEL) to specify which metric is to be reported and the specific RMID for which the data should be returned.

Processor support of the monitoring framework and sub-features such as CMT is reported via the CPUID instruction. The resource type available to the monitoring framework is enumerated via a new leaf function in CPUID. Reading and writing to the monitoring MSRs requires the RDMSR and WRMSR instructions.

The Cache Monitoring Technology feature set provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the CMT feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- CMT-specific event codes to read occupancy for a given level of the cache hierarchy.

The Memory Bandwidth Monitoring feature provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the MBM feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- MBM-specific event codes to read bandwidth out to the next level of the hierarchy and various sub-event codes to read more specific metrics as discussed later (e.g., total bandwidth vs. bandwidth only from local memory controllers on the same package).

### 18.18.2 Enabling Monitoring: Usage Flow

Figure 18-19 illustrates the key steps for OS/VMM to detect support of shared resource monitoring features such as CMT and enable resource monitoring for available resource types and monitoring events.
18.18.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring

Software can query processor support of shared resource monitoring features capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] reports 1, the processor provides the following programming interfaces for shared resource monitoring, including Cache Monitoring Technology:

- CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides information on available resource types (see Section 18.18.4), and monitoring capabilities for each resource type (see Section 18.18.5). Note CMT and MBM capabilities are enumerated as separate event vectors using shared enumeration infrastructure under a given resource type.
- IA32_PQR_ASSOC.RMID: The per-logical-processor MSR, IA32_PQR_ASSOC, that OS/VMM can use to assign an RMID to each logical processor, see Section 18.18.6.
- IA32_QM_EVTSEL: This MSR specifies an Event ID (EvtID) and an RMID which the platform uses to look up and provide monitoring data in the monitoring counter, IA32_QM_CTR, see Section 18.18.7.
- IA32_QM_CTR: This MSR reports monitored resource data when available along with bits to allow software to check for error conditions and verify data validity.

Software must follow the following sequence of enumeration to discover Cache Monitoring Technology capabilities:

1. Execute CPUID with EAX=0 to discover the “cpuid_maxLeaf” supported in the processor;
2. If cpuid_maxLeaf >= 7, then execute CPUID with EAX=7, ECX= 0 to verify CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] is set;
3. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] = 1, then execute CPUID with EAX=0FH, ECX= 0 to query available resource types that support monitoring;
4. If CPUID.(EAX=0FH, ECX=0):EDX.L3[bit 1] = 1, then execute CPUID with EAX=0FH, ECX= 1 to query the specific capabilities of L3 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.
5. If CPUID.(EAX=0FH, ECX=0):EDX reports additional resource types supporting monitoring, then execute CPUID with EAX=0FH, ECX set to a corresponding resource type ID (ResID) as enumerated by the bit position of CPUID.(EAX=0FH, ECX=0):EDX.

18.18.4 Monitoring Resource Type and Capability Enumeration

CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides one sub-leaf (sub-function 0) that reports shared enumeration infrastructure, and one or more sub-functions that report feature-specific enumeration data:

- Monitoring leaf sub-function 0 enumerates available resources that support monitoring, i.e., executing CPUID with EAX=0FH and ECX=0H. In the initial implementation, L3 cache is the only resource type available. Each
supported resource type is represented by a bit in CPUID.(EAX=0FH, ECX=0):EDX[31:1]. The bit position corresponds to the sub-leaf index (ResID) that software must use to query details of the monitoring capability of that resource type (see Figure 18-21 and Figure 18-22). Reserved bits of CPUID.(EAX=0FH, ECX=0):EDX[31:2] correspond to unsupported sub-leaves of the CPUID.0FH leaf. Additionally, CPUID.(EAX=0FH, ECX=0H):EBX reports the highest RMID value of any resource type that supports monitoring in the processor.

![Figure 18-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration](image)

### 18.18.5 Feature-Specific Enumeration

Each additional sub-leaf of CPUID.(EAX=0FH, ECX=ResID) enumerates the specific details for software to program monitoring MSRs using the resource type associated with the given ResID.

Note that in future Monitoring implementations the meanings of the returned registers may vary in other sub-leaves that are not yet defined. The registers will be specified and defined on a per-ResID basis.

![Figure 18-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H) )](image)

CPUID.(EAX=0FH, ECX=1H).EAX[7:0]: Encode counter width as offset from 24b. See Section 18.18.5.2 for details. Bits 31:11 of EAX are reserved.
CPUID.(EAX=0FH, ECX=1H).EAX[bit 8]: If 1, indicates the presence of an overflow bit in the IA32_QM_CTR MSR. See Section 18.18.5.2 for details. Bits 31:11 of EAX are reserved.

CPUID.(EAX=0FH, ECX=1H).EAX[bit 9]: If 1, indicates the presence of non-CPU agent Intel RDT CMT support. See Section 18.20 for details. Bits 31:11 of EAX are reserved.

CPUID.(EAX=0FH, ECX=1H).EAX[bit 10]: If 1, indicates the presence of non-CPU agent Intel RDT MBM support. See Section 18.20 for details. Bits 31:11 of EAX are reserved.

For each supported Cache Monitoring resource type, hardware supports only a finite number of RMIDs. CPUID.(EAX=0FH, ECX=1H).ECX enumerates the highest RMID value that can be monitored with this resource type, see Figure 18-21.

CPUID.(EAX=0FH, ECX=1H).EDX specifies a bit vector that is used to look up the EventID (See Figure 18-22 and Table 18-18) that software must program with IA32_QM_EVTSEL in order to retrieve event data. After software configures IA32_QMEVTSEL with the desired RMID and EventID, it can read the resulting data from IA32_QM_CTR. The raw numerical value reported from IA32_QM_CTR can be converted to the final value (occupancy in bytes or bandwidth in bytes per sampled time period) by multiplying the counter value by the value from CPUID.(EAX=0FH, ECX=1H).EBX, see Figure 18-21.

![Figure 18-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H))](image)

### 18.18.5.1 Cache Monitoring Technology

On processors for which Cache Monitoring Technology supports the L3 cache occupancy event, CPUID.(EAX=0FH, ECX=1H).EDX returns with bit 0 set. The corresponding event ID is shown in Table 18-18. The L3 occupancy data accumulated in the IA32_QM_CTR MSR can be converted to total occupancy (in bytes) by multiplying with CPUID.(EAX=0FH, ECX=1H).EBX.

Event codes for Cache Monitoring Technology are discussed in the next section.

### 18.18.5.2 Memory Bandwidth Monitoring

On processors that monitoring supports Memory Bandwidth Monitoring using ResID=1 (L3), two additional bits are defined in the vector at CPUID.(EAX=0FH, ECX=1H).EDX:

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 1]: indicates the L3 total external bandwidth monitoring event is supported if set. This event monitors the L3 total external bandwidth to the next level of the cache hierarchy, including all demand and prefetch misses from the L3 to the next hierarchy of the memory system. In most platforms, this represents memory bandwidth.

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 2]: indicates L3 local memory bandwidth monitoring event is supported if set. This event monitors the L3 external bandwidth satisfied by the local memory. In most platforms that support this event, L3 requests are likely serviced by a memory system with non-uniform memory architecture. This allows bandwidth to off-package memory resources to be tracked by subtracting local from total bandwidth. (For instance, bandwidth over QPI to a memory controller on another physical processor could be tracked by subtraction). Note that it is not possible to read the local and total bandwidth atomically; multiple operations are needed. Because of this, it is possible for the counters to change in between the two reads.

The corresponding Event ID is shown in Table 18-18. The L3 bandwidth data accumulated in IA32_QM_CTR can be converted to total bandwidth (in bytes) using CPUID.(EAX=0FH, ECX=1H).EBX.
DEBUG, BRANCH PROFILE, TSC, AND INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FEATURES

Table 18-18. Monitoring Supported Event IDs

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event ID</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 Cache Occupancy</td>
<td>01H</td>
<td>Cache Monitoring Technology</td>
</tr>
<tr>
<td>L3 Total External Bandwidth</td>
<td>02H</td>
<td>MBM</td>
</tr>
<tr>
<td>L3 Local External Bandwidth</td>
<td>03H</td>
<td>MBM</td>
</tr>
<tr>
<td>Reserved</td>
<td>All other event codes</td>
<td>N/A</td>
</tr>
</tbody>
</table>

A field is added to CPUID to enumerate the MBM counter width in platforms that support the extensible MBM counter width feature.

- CPUID.(EAX=0FH, ECX=1H).EAX[7:0]: Encode counter width as offset from 24b in bits[7:0]. In EAX bits 7:0, the counter width is encoded as an offset from 24b. A value of zero in this field means 24-bit counters are supported. A value of 8 indicates that 32-bit counters are supported, as in the 3rd generation Intel Xeon Scalable Processor Family. With this enumerable counter width, the requirement that software polls at 1Hz is removed. Software may poll at a varying rate with a reduced risk of rollover. Under typical conditions, rollover will likely require hundreds of seconds (though this value is not explicitly specified and may vary and decrease in future processor generations as memory bandwidths increase). Suppose software seeks to ensure that rollover does not occur more than once between samples. In that case, sampling at 1Hz while consuming the enumerated counter widths' worth of data will provide this guarantee for a specific platform and counter width under all conditions.

- CPUID.(EAX=0FH, ECX=1H).EAX[8]: Enumeration of the presence of an overflow bit in the IA32_QM_CTR MSR via EAX bit[8]. Software that uses the MBM event retrieval MSR interface should be updated to comprehend this new format, which enables up to 62-bit MBM counters to be provided by future platforms. Higher-level software that consumes the resulting bandwidth values is not expected to be affected. An overflow bit is defined in the IA32_QM_CTR MSR, bit 61, if CPUID.(EAX=0FH, ECX=1H).EAX[bit 8] is set. This rollover bit will be set on overflow of the MBM counters and reset upon read. Current processors do not support this capability.

18.18.6 Monitoring Resource RMID Association

After Monitoring and sub-features have been enumerated, software can begin using the monitoring features. The first step is to associate a given software thread (or multiple threads as part of an application, VM, group of applications or other abstraction) with an RMID.

Note that the process of associating an RMID with a given software thread is the same for all shared resource monitoring features (CMT, MBM), and a given RMID number has the same meaning from the viewpoint of any logical processors in a package. Stated another way, a thread may be associated in a 1:1 mapping with an RMID, and that RMID may allow cache occupancy, memory bandwidth information or other monitoring data to be read back later with monitoring event codes (retrieving data is discussed in a previous section).

The association of an application thread with an RMID requires an OS to program the per-logical-processor MSR IA32_PQR_ASSOC at context swap time (updates may also be made at any other arbitrary points during program execution such as application phase changes). The IA32_PQR_ASSOC MSR specifies the active RMID that monitoring hardware will use to tag internal operations, such as L3 cache requests. The layout of the MSR is shown in Figure 18-23. Software specifies the active RMID to monitor in the IA32_PQR_ASSOC.RMID field. The width of the RMID field can vary from one implementation to another, and is derived from \( \text{Ceil} \left( \log_2 \left( \text{CPUID.(EAX=0FH, ECX=0H).EBX}[31:0] + 1 \right) \right) \). The value of IA32_PQR_ASSOC after power-on is 0.

| Width of IA32_PQR_ASSOC.RMID field: \( \log_2 \left( \text{CPUID.(EAX=0FH, ECX=0H).EBX}[31:0] + 1 \right) \) |
|--------------------|--------|
| 63                 | 32     |
| 31                 | 10     |
| 9                  | 0      |
| Reserved for CLOS  | Reserved |
| RMID               |        |

Figure 18-23. IA32_PQR_ASSOC MSR
In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32_PQR_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g., RMID 31) may only be supported by one resource type but not by another resource type.

18.18.7 Monitoring Resource Selection and Reporting Infrastructure

The reporting mechanism for Cache Monitoring Technology and other related features is architecturally exposed as an MSR pair that can be programmed and read to measure various metrics such as the L3 cache occupancy (CMT) and bandwidths (MBM) depending on the level of Monitoring support provided by the platform. Data is reported back on a per-RMID basis. These events do not trigger based on event counts or trigger APIC interrupts (e.g., no Performance Monitoring Interrupt occurs based on counts). Rather, they are used to sample counts explicitly.

The MSR pair for the shared resource monitoring features (CMT, MBM) is separate from and not shared with architectural Perfmon counters, meaning software can use these monitoring features simultaneously with the Perfmon counters.

Access to the aggregated monitoring information is accomplished through the following programmable monitoring MSRs:

- IA32_QM_EVTSEL: This MSR provides a role similar to the event select MSRs for programmable performance monitoring described in Chapter 18. The simplified layout of the MSR is shown in Figure 18-24. IA32_QM_EVTSEL.EvtID (bits 7:0) specifies an event code of a supported resource type for hardware to report monitored data associated with IA32_QM_EVTSEL.RMID (bits 41:32). Software can configure IA32_QM_EVTSEL.RMID with any RMID that is active within the physical processor. The width of IA32_QM_EVTSEL.RMID matches that of IA32_PQR_ASSOC.RMID. Supported event codes for the IA32_QM_EVTSEL register are shown in Table 18-18. Note that valid event codes may not necessarily map directly to the bit position used to enumerate support for the resource via CPUID.

Software can program an RMID / Event ID pair into the IA32_QM_EVTSEL MSR bit field to select an RMID to read a particular counter for a given resource. The currently supported list of Monitoring Event IDs is discussed in Section 18.18.5, which covers feature-specific details.

Thread access to the IA32_QM_EVTSEL and IA32_QM_CTR MSR pair should be serialized (that is, treated as a critical section under lock) to avoid situations where one thread changes the RMID/EvtID just before another thread reads monitoring data from IA32_QM_CTR.

- IA32_QM_CTR: This MSR reports monitored data when available. It contains three bit fields. If software configures an unsupported RMID or event type in IA32_QM_EVTSEL, then IA32_QM_CTR.Error (bit 63) will be set, indicating there is no valid data to report. If IA32_QM_CTR.Unavailable (bit 62) is set, it indicates monitored data for the RMID is not available, and IA32_QM_CTR.data (bits 61:0) should be ignored. Therefore, IA32_QM_CTR.data (bits 61:0) is valid only if bit 63 and 62 are both clear. For Cache Monitoring Technology, software can convert IA32_QM_CTR.data into cache occupancy or bandwidth metrics expressed in bytes by multiplying with the conversion factor from CPUID.(EAX=0FH, ECX=1H).EBX.
18.18.8 Monitoring Programming Considerations

Figure 18-25 illustrates how system software can program IA32_QOSEVTSEL and IA32_QM_CTR to perform resource monitoring.

Though the field provided in IA32_QM_CTR allows for up to 62 bits of data to be returned, often a subset of bits are used. With Cache Monitoring Technology for instance, the number of bits used is the base-two logarithm of the total cache size divided by the Upscaling Factor from CPUID.

In Memory Bandwidth Monitoring, the initial counter size is 24 bits, and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period. Any changes to counter width are enumerated to software; see Section 18.18.5.2 for details.

18.18.8.1 Monitoring Dynamic Configuration

Both the IA32_QM_EVTSEL and IA32_PQR_ASSOC registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- An RMID exceeding the maximum RMID is used.

18.18.8.2 Monitoring Operation With Power Saving Features

Some advanced power management features such as deep package C-states may shrink the L3 cache and cause CMT occupancy count to be reduced. MBM bandwidth counts may increase due to flushing cached data out of L3.
18.18.8.3 Monitoring Operation with Other Operating Modes
The states in IA32_PQR_ASSOC and monitoring counter are unmodified across an SMI delivery. Thus, the execution of SMM handler code and SMM handler’s data can manifest as spurious contribution in the monitored data.

It is possible for an SMM handler to minimize the impact on of spurious contribution in the QOS monitoring counters by reserving a dedicated RMID for monitoring the SMM handler. Such an SMM handler can save the previously configured QOS Monitoring state immediately upon entering SMM, and restoring the QOS monitoring state back to the prev-SMM RMID upon exit.

18.18.8.4 Monitoring Operation with RAS Features
In general, the Reliability, Availability, and Serviceability (RAS) features present in Intel Platforms are not expected to significantly affect shared resource monitoring counts. In cases where software RAS features cause memory copies or cache accesses, these may be tracked and may influence the shared resource monitoring counter values.

18.19 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES
The Intel Resource Director Technology (Intel RDT) feature set provides a set of allocation (resource control) capabilities including Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP). The Intel Xeon processor E5 v4 family (and a subset of communication-focused processors in the Intel Xeon E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Certain Intel Atom processors also provide support for control over the L2 cache, with capabilities as described below. The programming interface for Cache Allocation Technology and for the more general allocation capabilities are described in the rest of this chapter. The CAT and CDP capabilities, where architecturally supported, may be detected and enumerated in software using the CPUID instruction, as described in this chapter.

The Intel Xeon Scalable Processor Family introduces the Memory Bandwidth Allocation (MBA) feature which provides indirect control over the memory bandwidth available to CPU cores, and is discussed later in this chapter.

18.19.1 Introduction to Cache Allocation Technology (CAT)
Cache Allocation Technology enables an Operating System (OS), Hypervisor /Virtual Machine Manager (VMM) or similar system service management agent to specify the amount of cache space into which an application can fill (as a hint to hardware - certain features such as power management may override CAT settings). Specialized user-level implementations with minimal OS support are also possible, though not necessarily recommended (see notes below for OS/Hypervisor with respect to ring 3 software and virtual guests). Depending on the processor family, L2 or L3 cache allocation capability may be provided, and the technology is designed to scale across multiple cache levels and technology generations.

Software can determine which levels are supported in a given platform programmatically using CPUID as described in the following sections.

The CAT mechanisms defined in this document provide the following key features:

- A mechanism to enumerate platform Cache Allocation Technology capabilities and available resource types that provides CAT control capabilities. For implementations that support Cache Allocation Technology, CPUID provides enumeration support to query which levels of the cache hierarchy are supported and specific CAT capabilities, such as the max allocation bitmask size,
- A mechanism for the OS or Hypervisor to configure the amount of a resource available to a particular Class of Service via a list of allocation bitmasks,
- Mechanisms for the OS or Hypervisor to signal the Class of Service to which an application belongs, and
- Hardware mechanisms to guide the LLC fill policy when an application has been designated to belong to a specific Class of Service.

Note that for many usages, an OS or Hypervisor may not want to expose Cache Allocation Technology mechanisms to Ring3 software or virtualized guests.
The Cache Allocation Technology feature enables more cache resources (i.e., cache space) to be made available for high priority applications based on guidance from the execution environment as shown in Figure 18-26. The architecture also allows dynamic resource reassignment during runtime to further optimize the performance of the high priority application with minimal degradation to the low priority app. Additionally, resources can be rebalanced for system throughput benefit across uses cases of OSes, VMMs, containers, and other scenarios by managing the CPUID and MSR interfaces. This section describes the hardware and software support required in the platform including what is required of the execution environment (i.e., OS/VMM) to support such resource control. Note that in Figure 18-26 the L3 Cache is shown as an example resource.

**18.19.2 Cache Allocation Technology Architecture**

The fundamental goal of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (COS or CLOS). The processor exposes a set of Classes of Service into which applications (or individual threads) can be assigned. Cache allocation for the respective applications or threads is then restricted based on the class with which they are associated. Each Class of Service can be configured using capacity bitmasks (CBMs) which represent capacity and indicate the degree of overlap and isolation between classes. For each logical processor there is a register exposed (referred to here as the IA32_PQR_ASSOC MSR or PQR) to allow the OS/VMM to specify a COS when an application, thread or VM is scheduled.

The usage of Classes of Service (COS) are consistent across resources and a COS may have multiple resource control attributes attached, which reduces software overhead at context swap time. Rather than adding new types of COS tags per resource for instance, the COS management overhead is constant. Cache allocation for the indicated application/thread/container/VM is then controlled automatically by the hardware based on the class and the bitmask associated with that class. Bitmasks are configured via the IA32_resourceType_MASK_n MSRs, where resourceType indicates a resource type (e.g., “L3” for the L3 cache) and “n” indicates a COS number.

The basic ingredients of Cache Allocation Technology are as follows:

- An architecturally exposed mechanism using CPUID to indicate whether CAT is supported, and what resource types are available which can be controlled,
- For each available resourceType, CPUID also enumerates the total number of Classes of Services and the length of the capacity bitmasks that can be used to enforce cache allocation to applications on the platform,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to configure the behavior of different classes of service using the bitmasks available,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to assign a COS to an executing software thread (i.e., associating the active CR3 of a logical processor with the COS in IA32_PQR_ASSOC),
- Implementation-dependent mechanisms to indicate which COS is associated with a memory access and to enforce the cache allocation on a per COS basis.

A capacity bitmask (CBM) provides a hint to the hardware indicating the cache space an application should be limited to as well as providing an indication of overlap and isolation in the CAT-capable cache from other applications contending for the cache. The bit length of the capacity mask available generally depends on the configura-
tion of the cache and is specified in the enumeration process for CAT in CPUID (this may vary between models in a processor family as well). Similarly, other parameters such as the number of supported COS may vary for each resource type, and these details can be enumerated via CPUID.

Sample cache capacity bitmasks for a bit length of 8 are shown in Figure 18-27. Note that all (and only) contiguous '1' combinations are allowed (e.g., FFFFH, 0FF0H, 003CH, etc.), unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type. Attempts to program a value without contiguous '1's (including zero) will result in a general protection fault (#GP(0)). It is generally expected that in way-based implementations, one capacity mask bit corresponds to some number of ways in cache, but the specific mapping is implementation-dependent. In all cases, a mask bit set to '1' specifies that a particular Class of Service can allocate into the cache subset represented by that bit. A value of '0' in a mask bit specifies that a Class of Service cannot allocate into the given cache subset. In general, allocating more cache to a given application is usually beneficial to its performance.

Figure 18-27 also shows three examples of sets of Cache Capacity Bitmasks. For simplicity these are represented as 8-bit vectors, though this may vary depending on the implementation and how the mask is mapped to the available cache capacity. The first example shows the default case where all 4 Classes of Service (the total number of COS are implementation-dependent) have full access to the cache. The second case shows an overlapped case, which would allow some lower-priority threads to share cache space with the highest priority threads. The third case shows various non-overlapped partitioning schemes. As a matter of software policy for extensibility, COS0 should typically be considered and configured as the highest priority COS, followed by COS1, and so on, though there is no hardware restriction enforcing this mapping. When the system boots all threads are initialized to COS0, which has full access to the cache by default.

Though the representation of the CBMs looks similar to a way-based mapping they are independent of any specific enforcement implementation (e.g., way partitioning.) Rather, this is a convenient manner to represent capacity, overlap, and isolation of cache space. For example, executing a POPCNT instruction (population count of set bits)
on the capacity bitmask can provide the fraction of cache space that a class of service can allocate into. In addition to the fraction, the exact location of the bits also shows whether the class of service overlaps with other classes of service or is entirely isolated in terms of cache space used.

Figure 18-28 shows how the Cache Capacity Bitmasks and the per-logical-processor Class of Service are logically used to enable Cache Allocation Technology. All (and only) contiguous 1’s in the CBM are permitted, unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type. The length of a CBM may vary from resource to resource or between processor generations and can be enumerated using CPUID. From the available mask set and based on the goals of the OS/VMM (shared or isolated cache, etc.) bitmasks are selected and associated with different classes of service. For the available Classes of Service the associated CBMs can be programmed via the global set of CAT configuration registers (in the case of L3 CAT, via the IA32_L3_MASK_n MSRs, where “n” is the Class of Service, starting from zero). In all architectural implementations supporting CPUID it is possible to change the CBMs dynamically, during program execution, unless stated otherwise by Intel.

The currently running application’s Class of Service is communicated to the hardware through the per-logical-processor PQR MSR (IA32_PQR_ASSOC MSR). When the OS schedules an application thread on a logical processor, the application thread is associated with a specific COS (i.e., the corresponding COS in the PQR) and all requests to the CAT-capable resource from that logical processor are tagged with that COS (in other words, the application thread is configured to belong to a specific COS). The cache subsystem uses this tagged request information to enforce QoS. The capacity bitmask may be mapped into a way bitmask (or a similar enforcement entity based on the implementation) at the cache before it is applied to the allocation policy. For example, the capacity bitmask can be an 8-bit mask and the enforcement may be accomplished using a 16-way bitmask for a cache enforcement implementation based on way partitioning.

The following sections describe extensions of CAT such as Code and Data Prioritization (CDP), followed by details on specific features such as L3 CAT, L3 CDP, L2 CAT, and L2 CDP. Depending on the specific processor a mix of features may be supported, and CPUID provides enumeration capabilities to enable software to dynamically detect the set of supported features.
18.19.3 Code and Data Prioritization (CDP) Technology

Code and Data Prioritization Technology is an extension of CAT. CDP enables isolation and separate prioritization of code and data fetches to the L2 or L3 cache in a software configurable manner, depending on hardware support, which can enable workload prioritization and tuning of cache capacity to the characteristics of the workload. CDP extends Cache Allocation Technology (CAT) by providing separate code and data masks per Class of Service (COS). Support for the L2 CDP feature and the L3 CDP features are separately enumerated (via CPUID) and separately controlled (via remapping the L2 CAT MSRs or L3 CAT MSRs respectively). Section 18.19.6.3 and Section 18.19.7 provide details on enumerating, controlling, and enabling L3 and L2 CDP respectively, while this section provides a general overview.

The L3 CDP feature was first introduced on the Intel Xeon E5 v4 family of server processors, as an extension to L3 CAT. The L2 CDP feature is first introduced on future Intel Atom family processors, as an extension to L2 CAT.

By default, CDP is disabled on the processor. If the CAT MSRs are used without enabling CDP, the processor operates in a traditional CAT-only mode. When CDP is enabled,

- the CAT mask MSRs are re-mapped into interleaved pairs of mask MSRs for data or code fetches (see Figure 18-29),
- the range of COS for CAT is re-indexed, with the lower-half of the COS range available for CDP.

Using the CDP feature, virtual isolation between code and data can be configured on the L2 or L3 cache if desired, similar to how some processor cache levels provide separate L1 data and L1 instruction caches.

Like the CAT feature, CDP may be dynamically configured by privileged software at any point during normal system operation, including dynamically enabling or disabling the feature provided that certain software configuration requirements are met (see Section 18.19.5).

An example of the operating mode of CDP is shown in Figure 18-29. Shown at the top are traditional CAT usage models where capacity masks map 1:1 with a COS number to enable control over the cache space which a given COS (and thus applications, threads or VMs) may occupy. Shown at the bottom are example mask configurations where CDP is enabled, and each COS number maps 1:2 to two masks, one for code and one for data. This enables code and data to be either overlapped or isolated to varying degrees either globally or on a per-COS basis, depending on application and system needs.
When CDP is enabled, the existing mask space for CAT-only operation is split. As an example if the system supports 16 CAT-only COS, when CDP is enabled the same MSR interfaces are used, however half of the masks correspond to code, half correspond to data, and the effective number of COS is reduced by half. Code/Data masks are defined per-COS and interleaved in the MSR space as described in subsequent sections.

In cases where CPUID exposes a non-even number of supported Classes of Service for the CAT or CDP features, software using CDP should use the lower matched pairs of code/data masks, and any upper unpaired masks should not be used. As an example, if CPUID exposes 5 CLOS, when CDP is enabled then two code/data pairs are available (masks 0/1 for CLOS[0] data/code and masks 2/3 for CLOS[1] data/code), however the upper un-paired mask should not be used (mask 4 in this case) or undefined behavior may result.

18.19.4   Enabling Cache Allocation Technology Usage Flow

Figure 18-30 illustrates the key steps for OS/VMM to detect support of Cache Allocation Technology and enable priority-based resource allocation for a CAT-capable resource.

- **On OS/VMM Initialization**: CPUID(7:0):EBX.15
- **Cache Allocation Configuration**: Configure CBM per COS
- **On Context Switch**: Set COS for scheduled thread context

Figure 18-30. Cache Allocation Technology Usage Flow

Enumeration and configuration of L2 CAT is similar to L3 CAT, however CPUID details and MSR addresses differ. Common CLOS are used across the features.

18.19.4.1   Enumeration and Detection Support of Cache Allocation Technology

Software can query processor support of CAT capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software must use CPUID leaf 10H to enumerate additional details of available resource types, classes of services and capability bitmasks. The programming interfaces provided by Cache Allocation Technology include:

- **CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) and its sub-functions provide information on available resource types, and CAT capability for each resource type** (see Section 18.19.4.2).
- **IA32_L3_MASK_n**: A range of MSRs is provided for each resource type, each MSR within that range specifying a software-configured capacity bitmask for each class of service. For L3 with Cache Allocation support, the CBM is specified using one of the IA32_L3_QOS_MASK_n MSR, where ‘n’ corresponds to a number within the supported range of COS, i.e., the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive. See Section 18.19.4.3 for details.
- **IA32_L2_MASK_n**: A range of MSRs is provided for L2 Cache Allocation Technology, enabling software control over the amount of L2 cache available for each CLOS. Similar to L3 CAT, a CBM is specified for each CLOS using the set of registers, IA32_L2_QOS_MASK_n MSR, where ‘n’ ranges from zero to the maximum CLOS number reported for L2 CAT in CPUID. See Section 18.19.4.3 for details.

The L2 mask MSRs are scoped at the same level as the L2 cache (similarly, the L3 mask MSRs are scoped at the same level as the L3 cache). Software may determine which logical processors share an MSR (for instance local...
to a core, or shared across multiple cores) by performing a write to one of these MSRs and noting which logical threads observe the change. Example flows for a similar method to determine register scope are described in Section 16.5.2, “System Software Recommendation for Managing CMCI and Machine Check Resources.” Software may also use CPUID leaf 4 to determine the maximum number of logical processor IDs that may share a given level of the cache.

- **IA32_PQR_ASSOC.CLOS:** The IA32_PQR_ASSOC MSR provides a COS field that OS/VMM can use to assign a logical processor to an available COS. The set of COS are common across all allocation features, meaning that multiple features may be supported in the same processor without additional software COS management overhead at context swap time. See Section 18.19.4.4 for details.

### 18.19.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration

CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) provides two or more sub-functions:

- **CAT Enumeration leaf sub-function 0 enumerates available resource types that support allocation control, i.e., by executing CPUID with EAX=10H and ECX=0H. Each supported resource type is represented by a bit field in CPUID.(EAX=10H, ECX=0H):EBX[31:1]. The bit position of each set bit corresponds to a Resource ID (ResID), for instance ResID=1 is used to indicate L3 CAT support, and ResID=2 indicates L2 CAT support. The ResID is also the sub-leaf index that software must use to query details of the CAT capability of that resource type (see Figure 18-31).

<table>
<thead>
<tr>
<th>EBX</th>
<th>Reserved</th>
<th>M</th>
<th>B</th>
<th>A</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification

- For ECX>0, EAX[4:0] reports the length of the capacity bitmask (ECX=1 or 2 for L3 CAT or L2 CAT respectively). Add one to the return value to get the result, e.g., a value of 15 corresponds to the capacity bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

- Sub-functions of CPUID.EAX=10H with a non-zero ECX input matching a supported ResID enumerate the specific enforcement details of the corresponding ResID. The capabilities enumerated include the length of the capacity bitmasks and the number of Classes of Service for a given ResID. Software should query the capability of each available ResID that supports CAT from a sub-leaf of leaf 10H using the sub-leaf index reported by the corresponding non-zero bit in CPUID.(EAX=10H, ECX=0H):EBX[31:1] in order to obtain additional feature details.
CAT capability for L3 is enumerated by CPUID.(EAX=10H, ECX=1H), see Figure 18-32. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=1) are:

- CPUID.(EAX=10H, ECX=ResID=1):EAX[4:0] reports the length of the capacity bitmask. Add one to the return value to get the result, e.g., a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

- CPUID.(EAX=10H, ECX=1):EBX [31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L3 allocation may be used by other entities in the platform (e.g., an integrated graphics engine or hardware units outside the processor core and have direct access to L3). Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.

- CPUID.(EAX=10H, ECX=1):ECX[bit 1]: If 1, indicates L3 CAT for non-CPU agents is supported. Bits 0 and 31:4 of ECX are reserved. See section 18.20 for details.

- CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2]: If 1, indicates L3 Code and Data Prioritization Technology is supported (see Section 18.19.5). Bits 0 and 31:4 of ECX are reserved.

- CPUID.(EAX=10H, ECX=1):ECX[bit 3]: If 1, indicates non-contiguous capacity bitmask is supported. The bits that are set in the various IA32_L3_MASK_n registers do not have to be contiguous. Bits 0 and 31:4 of ECX are reserved.

- CPUID.(EAX=10H, ECX=1):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.
• CAT capability for L2 is enumerated by CPUID.(EAX=10H, ECX=2H), see Figure 18-33. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=2) are:
  — CPUID.(EAX=10H, ECX=ResID=2):EAX[4:0] reports the length of the capacity bitmask. Add one to the return value to get the result, e.g., a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.
  — CPUID.(EAX=10H, ECX=2):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L2 allocation may be used by other entities in the platform. Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.
  — CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2]: If 1, indicates L2 Code and Data Prioritization Technology is supported (see Section 17.19.6). Bits 1:0 and 31:4 of ECX are reserved.
  — CPUID.(EAX=10H, ECX=2):ECX[bit 3]: If 1, indicates non-contiguous capacity bitmask is supported. The bits which are set in the various IA32_L2_MASK_n registers do not have to be contiguous. Bits 1:0 and 31:4 of ECX are reserved.
  — CPUID.(EAX=10H, ECX=2):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.

A note on migration of Classes of Service (COS): Software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the performance of the Cache Allocation Technology feature may result if COS are migrated frequently. This is aligned with the industry-standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.
18.19.4.3 Cache Allocation Technology: Cache Mask Configuration

After determining the length of the capacity bitmasks (CBM) and number of COS supported using CPUID (see Section 18.19.4.2), each COS needs to be programmed with a CBM to dictate its available cache via a write to the corresponding IA32_resourceType_MASK_n register, where 'n' corresponds to a number within the supported range of COS, i.e., the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive, and 'resource-Type' corresponds to a specific resource as enumerated by the set bits of CPUID.(EAX=10H, ECX=0):EBX[31:1], for instance, 'L2' or 'L3' cache.

A hierarchy of MSRs is reserved for Cache Allocation Technology registers of the form IA32_resource-Type_MASK_n:

- From 0C90H through 0D8FH (inclusive), providing support for multiple sub-ranges to support varying resource types. The first supported resource type is 'L3', corresponding to the L3 cache in a platform. The MSRs range from 0C90H through 0D0FH (inclusive), enables support for up to 128 L3 CAT Classes of Service.

- Within the same CAT range hierarchy, another set of registers is defined for resourceType 'L2', corresponding to the L2 cache in a platform, and MSRs IA32_L2_MASK_n are defined for n=[0,63] at addresses 0D10H through 0D4FH (inclusive).

Figure 18-34 and Figure 18-35 provide an overview of the relevant registers.

All CAT configuration registers can be accessed using the standard RDMSR / WRMSR instructions.

Note that once L3 or L2 CAT masks are configured, threads can be grouped into Classes of Service (COS) using the IA32_PQR_ASSOC MSR as described in Section 18.19.4.4, "Class of Service to Cache Mask Association: Common Across Allocation Features."

18.19.4.4 Class of Service to Cache Mask Association: Common Across Allocation Features

After configuring the available classes of service with the preferred set of capacity bitmasks, the OS/VMM can set the IA32_PQR_ASSOC.COS of a logical processor to the class of service with the desired CBM when a thread
context switch occurs. This allows the OS/VMM to indicate which class of service an executing thread/VM belongs within. Each logical processor contains an instance of the IA32_PQR_ASSOC register at MSR location 0C8FH, and Figure 18-34 shows the bit field layout for this register. Bits[63:32] contain the COS field for each logical processor.

Note that placing the RMID field within the same PQR register enables both RMID and CLOS to be swapped at context swap time for simultaneous use of monitoring and allocation features with a single register write for efficiency.

When CDP is enabled, specifying a COS value in IA32_PQR_ASSOC.COS greater than MAX_COS_CDP = (CPUID.(EAX=10H, ECX=1):EDX[15:0] >> 1) will cause undefined performance impact to code and data fetches. In all cases, code and data masks for L2 and L3 CDP should be programmed with at least one bit set.

Note that if the IA32_PQR_ASSOC.COS is never written then the CAT capability defaults to using COS 0, which in turn is set to the default mask in IA32_L3_MASK_0 - which is all "1"s (on reset). This essentially disables the enforcement feature by default or for legacy operating systems and software.

See Section 18.19.7, "Introduction to Memory Bandwidth Allocation," for important COS programming considerations including maximum values when using CAT and CDP.

### 18.19.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology

L3 CDP is an extension of L3 CAT. The presence of the L3 CDP feature is enumerated via CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] (see Figure 18-32). Most of the CPUID.(EAX=10H, ECX=1) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=1):EDX.COS_MAX_CAT specifies the maximum COS applicable to CAT-only operation. For CDP operations, COS_MAX_CDP is equal to (CPUID.(EAX=10H, ECX=1):EDX.COS_MAX_CAT >>1).

If CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] =1, the processor supports CDP and provides a new MSR IA32_L3_QOS_CFG at address 0C81H. The layout of IA32_L3_QOS_CFG is shown in Figure 18-36. The bit field definition of IA32_L3_QOS_CFG are:

- Bit 0: L3 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32_PQR_ASSOC.COS is COS_MAX_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).

![Figure 18-36. Layout of IA32_L3_QOS_CFG](image)

IA32_L3_QOS_CFG default values are all 0s at RESET, the mask MSRs are all 1s. Hence, all logical processors are initialized in COS0 allocated with the entire L3 with CDP disabled, until software programs CAT and CDP. The scope of the IA32_L3_QOS_CFG MSR is defined to be the same scope as the L3 cache (e.g., typically per processor socket). Refer to Section 18.19.7 for software considerations while enabling or disabling L3 CDP.

### 18.19.5.1 Mapping Between L3 CDP Masks and CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per COS. The re-mapping is shown in Table 18-19.
Table 18-19. Re-indexing of COS Numbers and Mapping to CAT/CDP Mask MSRs

<table>
<thead>
<tr>
<th>Mask MSR</th>
<th>CAT-only Operation</th>
<th>CDP Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_L3_QOS_Mask_0</td>
<td>COS0</td>
<td>COS0.Data</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_1</td>
<td>COS1</td>
<td>COS0.Code</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_2</td>
<td>COS2</td>
<td>COS1.Data</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_3</td>
<td>COS3</td>
<td>COS1.Code</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_4</td>
<td>COS4</td>
<td>COS2.Data</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_5</td>
<td>COS5</td>
<td>COS2.Code</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_’2n’</td>
<td>COS’2n’</td>
<td>COS’n’.Data</td>
</tr>
<tr>
<td>IA32_L3_QOS_Mask_’2n+1’</td>
<td>COS’2n+1’</td>
<td>COS’n’.Code</td>
</tr>
</tbody>
</table>

One can derive the MSR address for the data mask or code mask for a given COS number ‘n’ by:

- data_mask_address (n) = base + (n <<1), where base is the address of IA32_L3_QOS_MASK_0.
- code_mask_address (n) = base + (n <<1) +1.

When CDP is enabled, each COS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 18-29).

Mask MSR field definitions remain the same. Capacity masks must be formed of contiguous set bits, unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003, and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

18.19.6 Code and Data Prioritization (CDP): Enumerating and Enabling L2 CDP Technology

L2 CDP is an extension of the L2 CAT feature. The presence of the L2 CDP feature is enumerated via CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] (see Figure 17-33). Most of the CPUID.(EAX=10H, ECX=2) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=2):EDX.COS_MAX_CAT specifies the maximum COS applicable to CAT-only operation. For CDP operations, COS_MAX_CDP is equal to (CPUID.(EAX=10H, ECX=2):EDX.COS_MAX_CAT >>1).

If CPUID.(EAX=10H, ECX=2):ECX.CDP[bit 2] = 1, the processor supports L2 CDP and provides a new MSR IA32_L2_QOS_CFG at address 0C82H. The layout of IA32_L2_QOS_CFG is shown in Figure 18-37. The bit field definition of IA32_L2_QOS_CFG are:

- Bit 0: L2 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32_PQR_ASSOC.COS is COS_MAX_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).
IA32_L2_QOS_CFG default values are all 0s at RESET, and the mask MSRs are all 1s. Hence all logical processors are initialized in COS0 allocated with the entire L2 available and with CDP disabled, until software programs CAT and CDP. The IA32_L2_QOS_CFG MSR is defined at the same scope as the L2 cache, typically at the module level for Intel Atom processors for instance. In processors with multiple modules present it is recommended to program the IA32_L2_QOS_CFG MSR consistently across all modules for simplicity.

18.19.6.1 Mapping Between L2 CDP Masks and L2 CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per COS. This remapping is the same as the remapping shown in Table 18-19 for L3 CDP, but for the L2 MSR block (IA32_L2_QOS_MASK_n) instead of the L3 MSR block (IA32_L3_QOS_MASK_n). The same code / data mask mapping algorithm applies to remapping the MSR block between code and data masks.

As with L3 CDP, when L2 CDP is enabled, each COS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over code placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 18-29).

Mask MSR field definitions for L2 CDP remain the same as for L2 CAT. Capacity masks must be formed of contiguous set bits, unless otherwise non-contiguous capacity bitmask support is specified in CPUID enumeration for the resource type with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003, and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

18.19.6.2 Common L2 and L3 CDP Programming Considerations

Before enabling or disabling L2 or L3 CDP, software should write all 1’s to all of the corresponding CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs for the L3 CAT feature). When enabling CDP, software should also ensure that only COS number which are valid in CDP operation is used, otherwise undefined behavior may result. For instance in a case with 16 CAT COS, since COS are reduced by half when CDP is enabled, software should ensure that only COS 0-7 are in use before enabling CDP (along with writing 1’s to all mask bits before enabling or disabling CDP).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given COS may become a code mask for a different Class of Service when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to COS[0] before enabling or disabling CDP.

18.19.6.3 Cache Allocation Technology Dynamic Configuration

All Intel Resource Director Technology (Intel RDT) interfaces including the IA32_PQR_ASSOC MSR, CAT/CDP masks, MBA delay values, and CQM/MBM registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

• A reserved bit is modified,

• Accessing a QOS mask register outside the supported COS (the max COS number is specified in CPUID.(EAX=10H, ECX=ResID):EDX[15:0]), or

• Writing a COS greater than the supported maximum (specified as the maximum value of CPUID.(EAX=10H, ECX=ResID):EDX[15:0] for all valid ResID values) is written to the IA32_PQR_ASSOC.CLOS field.

When CDP is enabled, specifying a COS value in IA32_PQR_ASSOC.COS outside of the lower half of the COS space will cause undefined performance impact to code and data fetches due to MSR space re-indexing into code/data masks when CDP is enabled.

When reading the IA32_PQR_ASSOC register the currently programmed COS on the core will be returned.

When reading an IA32_resourceType_MASK_n register the current capacity bit mask for COS 'n' will be returned.
As noted previously, software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the accuracy of the Cache Allocation feature may result if COS are migrated frequently. This is aligned with the industry standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

18.19.6.4 Cache Allocation Technology Operation With Power Saving Features

Note that the Cache Allocation Technology feature cannot be used to enforce cache coherency, and that some advanced power management features such as C-states which may shrink or power off various caches within the system may interfere with CAT hints - in such cases the CAT bitmasks are ignored and the other features take precedence. If the highest possible level of CAT differentiation or determinism is required, disable any power-saving features which shrink the caches or power off caches. The details of the power management interfaces are typically implementation-specific, but can be found at Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

If software requires differentiation between threads but not absolute determinism then in many cases it is possible to leave power-saving cache shrink features enabled, which can provide substantial power savings and increase battery life in mobile platforms. In such cases when the caches are powered off (e.g., package C-states) the entire cache of a portion thereof may be powered off. Upon resuming an active state any new incoming data to the cache will be filled subject to the cache capacity bitmasks. Any data in the cache prior to the cache shrink or power off may have been flushed to memory during the process of entering the idle state, however, and is not guaranteed to remain in the cache. If differentiation between threads is the goal of system software then this model allows substantial power savings while continuing to deliver performance differentiation. If system software needs optimal determinism then power saving modes which flush portions of the caches and power them off should be disabled.

NOTE
IA32_PQR_ASSOC is saved and restored across C6 entry/exit. Similarly, the mask register contents are saved across package C-state entry/exit and are not lost.

18.19.6.5 Cache Allocation Technology Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and mask registers are unmodified across an SMI delivery. Thus, the execution of SMM handler code can interact with the Cache Allocation Technology resource and manifest some degree of non-determinism to the non-SMM software stack. An SMM handler may also perform certain system-level or power management practices that affect CAT operation.

It is possible for an SMM handler to minimize the impact on data determinism in the cache by reserving a COS with a dedicated partition in the cache. Such an SMM handler can switch to the dedicated COS immediately upon entering SMM, and switching back to the previously running COS upon exit.

18.19.6.6 Associating Threads with CAT/CDP Classes of Service

Threads are associated with Classes of Service (CLOS) via the per-logical-processor IA32_PQR_ASSOC MSR. The same COS concept applies to both CAT and CDP (for instance, COS[5] means the same thing whether CAT or CDP is in use, and the COS has associated resource usage constraint attributes including cache capacity masks). The mapping of COS to mask MSRs does change when CDP is enabled, according to the following guidelines:

- In CAT-only Mode - one set of bitmasks in one mask MSR control both code and data.
  - Each COS number map 1:1 with a capacity mask on the applicable resource (e.g., L3 cache).
- When CDP is enabled,
  - Two mask sets exist for each COS number, one for code, one for data.
  - Masks for code/data are interleaved in the MSR address space (see Table 18-19).
18.19.7 Introduction to Memory Bandwidth Allocation

The Memory Bandwidth Allocation (MBA) feature provides indirect and approximate control over memory bandwidth available per-core. It was introduced in the Intel Xeon Scalable Processor Family. This feature provides a method to control applications that may be over-utilizing bandwidth relative to their priority in environments such as the data-center.

The MBA feature uses existing constructs from the Intel RDT feature set, including Classes of Service (CLOS). A given CLOS used for L3 CAT, for instance, means the same thing as a CLOS used for MBA. Infrastructure, such as the MSR used to associate a thread with a CLOS (the IA32_PQR_ASSOC_MSR) and some elements of the CPUID enumeration (such as CPUID leaf 10H), are shared. Certain generations include advanced hardware controllers for efficiency. For more information, refer to the “Intel® Resource Director Technology Architecture Specification.”

The following sections describe CPU interfaces to Memory Bandwidth Allocation, such as CPUID enumeration and configuration interfaces (MSRs).

18.19.7.1 Memory Bandwidth Allocation Enumeration

Similar to other Intel RDT features, enumeration of the presence and details of the MBA feature is provided via a sub-leaf of the CPUID instruction.

Key components of the enumeration are as follows.

- Support for the MBA feature on the processor, and if MBA is supported, the following details:
  - Number of supported Classes of Service (CLOS) for the processor.
  - The maximum MBA delay value supported (which also implicitly provides a definition of the granularity).
  - An indication of whether the delay values which can be programmed are linearly spaced or not.

The presence of any of the Intel RDT features which enable control over shared platform resources is enumerated by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software may then use CPUID leaf 10H to enumerate additional details on the specific controls provided.

Through CPUID leaf 10H software may determine whether MBA is supported on the platform. Specifically, as shown in Figure 18-31, bit 3 of the EBX register indicates whether MBA is supported on the processor, and the bit position (3) constitutes a Resource ID (ResID) which allows enumeration of MBA details. For instance, if bit 3 is supported this implies the presence of CPUID.10H.[ResID=3] as shown in Figure 18-38 which provides the following details.

- CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] reports the maximum MBA throttling value supported, minus one. For instance, a value of 89 indicates that a maximum throttling value of 90 is supported. Additionally, in cases where a linear interface (see below) is supported then one hundred minus the maximum throttling value indicates the granularity, 10% in this example.

- CPUID.(EAX=10H, ECX=ResID=3):EBX is reserved.

- CPUID.(EAX=10H, ECX=ResID=3):ECX[2] reports whether the response of the delay values is linear (see text).

- CPUID.(EAX=10H, ECX=ResID=3):EDX[15:0] reports the number of Classes of Service (CLOS) supported for the feature (minus one). For instance, a reported value of 15 implies a maximum of 16 supported MBA CLOS.

The number of CLOS supported for the MBA feature may or may not align with other resources such as L3 CAT. In cases where the Intel RDT features support different numbers of CLOS the lowest numerical CLOS support the common set of features, while higher CLOS may support a subset. For instance, if L3 CAT supports 8 CLOS while MBA supports 4 CLOS, all 8 CLOS would have L3 CAT masks available for cache control, but the upper 4 CLOS would not offer MBA support. In this case the upper 4 CLOS would not be subject to any throttling control. Software can manage supported resources / CLOS in order to either have consistent capabilities across CLOS by using the common subset or enable more flexibility by selectively applying resource control where needed based on careful CLOS and thread mapping. In all cases, CLOS[0] supports all Intel RDT resource control features present on the platform.

Discussion on the interpretation and usage of the MBA delay values is provided in Section 18.19.7.2 on MBA configuration.
18.19.7.2 Memory Bandwidth Allocation Configuration

The configuration of MBA takes consists of two processes once enumeration is complete.

- Association of threads to Classes of Service (CLOS) - accomplished in a common fashion across Intel RDT features as described in Section 18.19.7.1 via the IA32_PQR_ASSOC MSR. As with features such as L3 CAT, software may update the CLOS field of the PQR MSR at context swap time in order to maintain the proper association of software threads to Classes of Service on the hardware. While logical processors may each be associated with independent CLOS, see Section 18.19.7.3 for important usage model considerations (initial versions of the MBA feature select the maximum delay value across threads).

- Configuration of the per-CLOS delay values, accomplished via the IA32_L2_QoS_Ext_BW_Thrtl_n MSR set shown in Table 18-20.

The MBA delay values which may be programmed range from zero (implying zero delay, and full bandwidth available) to the maximum (MBA_MAX) specified in CPUID as discussed in Section 18.19.7.1. The throttling values are approximate and do not sum to 100% across CLOS, rather they should be viewed as a maximum bandwidth "cap" per-CLOS.

Software may select an MBA delay value then write the value into one or more of the IA32_L2_QoS_Ext_BW_Thrtl_n MSRs to update the delay values applied for a specific CLOS. As shown in Table 18-20 the base address of the MSRs is at D50H, and the range corresponds to the maximum supported CLOS from CPUID.(EAX=10H, ECX=ResID=1):EDX[15:0] as described in Section 18.19.7.1. For instance, if 16 CLOS are supported then the valid MSR range will extend from D50H through D5F inclusive.
Table 18-20. MBA Delay Value MSRs

<table>
<thead>
<tr>
<th>Delay Value MSR</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_L2_QoS_Ext_BW_Thrtl_0</td>
<td>D50H</td>
</tr>
<tr>
<td>IA32_L2_QoS_Ext_BW_Thrtl_1</td>
<td>D51H</td>
</tr>
<tr>
<td>IA32_L2_QoS_Ext_BW_Thrtl_2</td>
<td>D52H</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>IA32_L2_QoS_Ext_BW_Thrtl_'COS_MAX'</td>
<td>D50H + COS_MAX from CPUID.10H.3</td>
</tr>
</tbody>
</table>

The definition for the MBA delay value MSRs is provided in Figure 17.39. The lower 16 bits are used for MBA delay values, and values from zero to the maximum from the CPUID MBA_MAX-1 value are supported. Values outside this range will generate #GP(0).

If linear input throttling values are indicated by CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] then values from zero through the MBA_MAX field from CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] are supported as inputs. In the linear mode the input precision is defined as 100- (MBA_MAX). For instance, if the MBA_MAX value is 90, the input precision is 10%. Values not an even multiple of the precision (e.g., 12%) will be rounded down (e.g., to 10% delay applied).

- If linear values are not supported (CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] = 0) then input delay values are powers-of-two from zero to the MBA_MAX value from CPUID. In this case any values not a power of two will be rounded down the next nearest power of two.

Note that the throttling values provided to software are calibrated through specific traffic patterns, however as workload characteristics may vary the response precision and linearity of the delay values will vary across products and should be treated as approximate values only.

18.19.7.3 Memory Bandwidth Allocation Usage Considerations

Different versions of Memory Bandwidth Allocation have various usage considerations and improving efficiency over time. See the “Intel® Resource Director Technology Architecture Specification” for additional details.

18.20 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FOR NON-CPU AGENTS

This section describes Intel RDT features for non-CPU agents. CPU agents are threads running on IA cores. Non-CPU agents include PCIe and CXL devices and integrated accelerators, thus broadly encompassing the set of agents that read from and write to either caches or memory, excluding IA cores. The non-CPU agent Intel RDT features enable monitoring of I/O device shared cache and memory bandwidth and cache allocation control. This provides features for I/O devices equivalent to the CPU agent Intel RDT capabilities CMT, MBM, and CAT (discussed in
Section 18.18 and Section 18.19). Refer to the "Intel® Resource Director Technology Architecture Specification" regarding design goals, use cases, software architecture, ACPI enumeration, and MMIO register interfaces.

“Non-CPU agent Intel RDT” refers to capabilities that monitor and control non-CPU agents’ resource utilization, including PCIe and CXL devices and integrated accelerators. Non-CPU agent Intel RDT may be called I/O RDT in some literature. In this document, the term “non-CPU agent Intel RDT” is used.

### 18.20.1 Non-CPU Agent Intel® RDT Features Enumeration Details

CPU agent Intel RDT features use the CPUID instruction to enumerate supported features and the level of support. Architectural Model-Specific Registers (MSRs) are interfaces to the monitoring and allocation features, as described in Sections 18.18 and 18.19.

Non-CPU agent Intel RDT builds on CPU agent Intel RDT by extending CPUID to indicate the presence and integration of non-CPU agent Intel RDT and by providing rich enumeration information in vendor-specific extensions to the Advanced Configuration and Power Interface (ACPI), in particular in the I/O RDT (IRDT) table. The ACPI extensions detailed in the "Intel® Resource Director Technology Architecture Specification" provide mechanisms to comprehend the structure of devices attached behind I/O blocks to particular links and what forms of tagging are supported on a per-link basis.

It is recommended that software parse CPUID and ACPI to obtain a detailed understanding of platform support and capabilities before attempting to use non-CPU agent Intel RDT.

#### 18.20.1.1 CPUID-Based Enumeration for Non-CPU Agent Intel® RDT Feature

CPUID-based enumeration provides a method by which all architectural Intel RDT features may be enumerated.

For CPU agent Intel RDT, monitoring details are enumerated in a CPUID sub-leaf denoted as CPUID.(EAX=0FH, ECX=ResID), where ResID corresponds to a resource ID bit index from the CPUID.(EAX=0FH, ECX=0) sub-leaf. Similarly, Intel RDT allocation features are described in CPUID.(EAX=10H, ECX=ResID). (Note that the ResID bit positions are not guaranteed to be symmetric or have the same encodings.)

No CPUID leaves or sub-leaves are created for non-CPU agent Intel RDT. Rather, non-CPU agent Intel RDT extends the existing Intel RDT CPUID sub-leaves with a bit per resource type, indicating whether non-CPU agent Intel RDT monitoring or control is present. CPUID.(EAX=0FH, ECX=ResID=1):EAX[bits 9, 10] represents the presence of CMT and MBM features for non-CPU agents. CPUID.(EAX=10H, ECX=ResID=1):ECX[bit 1] represents the presence of the CAT feature for non-CPU agents.

Specifically, for non-CPU Agent Intel RDT Monitoring (see Figure 18-21):

- Bits are added in the CPU Agent Intel RDT CMT/MBM leaf: CPUID.(EAX=0FH, ECX=ResID=1):EAX[bits 9, 10].
  - EAX[bit 9]: If set, indicates the presence of non-CPU Agent Cache Occupancy Monitoring (the equivalent of CPU Agent Intel RDT’s CMT feature).
  - EAX[bit 10]: If set, indicates the presence of non-CPU Agent memory L3 external BW monitoring (the equivalent of CPU Agent Intel RDT’s MBM feature).

For non-CPU Agent Intel RDT Allocation (see Figure 18-32):

  - ECX[bit 1]: If set, indicates the presence of non-CPU Agent Cache Allocation Technology (the equivalent of CPU Agent Intel RDT’s L3 CAT feature).
- As before, ECX[bit 2] indicates that L3 CDP is supported if set.

Note that no equivalent bits are defined in CPUID.(EAX=10H, ECX=ResID=2) as there is no ability for devices to fill into core L2 caches.

If any of these non-CPU agent Intel RDT enumeration bits are set, indicating that a monitoring feature or allocation feature is present, it also indicates the presence of the IA32_L3_10_RDT_CFG architectural MSR. This MSR may be used to enable the non-CPU agent Intel RDT features. See Section 18.20.2 for MSR details.

The presence of Intel RDT is a prerequisite for using the equivalent non-CPU agent Intel RDT feature. If a particular CPU agent Intel RDT feature is absent, any attempt to use non-CPU agent Intel RDT equivalents will result in...
general protection faults in the MSR interface. Attempts to enable unsupported features in the I/O complex will result in writes to the corresponding MMIO enable or configuration interfaces being ignored.

Software may use the existing CPUID leaves to gather the maximum number of RMID and CLOS tags for each resource level (e.g., L3 cache), and non-CPU agent Intel RDT is also subject to these limits.

Some platforms may support a mix of features, for instance, supporting L3 CAT architectural controls and the non-CPU agent Intel RDT equivalent, but no CMT/MBM monitoring or non-CPU agent monitoring equivalent, and these capabilities should be enumerated on a per-platform basis.

### 18.20.1.2 ACPI Enumeration

When support for non-CPU agent Intel RDT features is detected using CPUID, ACPI may be consulted for further details on the level of feature support, device structures behind various I/O ports, and the specific MMIO interfaces used to control a given device.

Non-CPU agent Intel RDT enumeration is via the “IRDT” ACPI table. For more information, refer to the “Intel® Resource Director Technology Architecture Specification.”

### 18.20.2 Non-CPU Agent Intel® RDT Feature Enable MSR

Before configuring non-CPU agent Intel RDT through MMIO, the feature should be enabled using the non-CPU agent Intel RDT Feature Enable MSR, IA32_L3_1O_RDT_CFG (MSR address 0C83H). As described in Section 18.20.1.1, the presence of one or more CPUID bits indicating support for one or more non-CPU agent Intel RDT features also indicates the presence of this MSR. This MSR may be used to enable the non-CPU agent Intel RDT features.

Two bits are defined in this MSR. Bit 0, when set, enables non-CPU agent RDT resource allocation features. Bit 1, when set, enables non-CPU agent Intel RDT monitoring features.

The L3 Non-CPU agent Intel RDT Monitoring Enable bit is supported if CPUID indicates that one or more non-CPU agent Intel RDT resource monitoring features are present.

The L3 Non-CPU agent Intel RDT Allocation Enable bit is supported if CPUID indicates that one or more non-CPU agent Intel RDT resource allocation features are present.

The default value is 0x0, so both classes of features are disabled by default. All bits not defined are reserved. Writing a non-zero value to any reserved bit will generate a General Protection Fault (#GP(0)).

This MSR is scoped at the L3 cache level and is cleared on system reset. It is expected that the software will configure this MSR consistently across all L3 caches that may be present on that package.

The definition of the IA32_L3_1O_RDT_CFG MSR is shown in Figure 18-40.

![Figure 18-40. Layout of the IA32_L3_1O_QOS_CFG MSR for Enabling Non-CPU Agent Intel® RDT](image-url)
This chapter defines a last-branch recording (LBR) facility that is architectural and part of the Intel 64 architecture. This facility is an enhancement of but distinct from earlier LBR facilities that were not architectural. Those earlier facilities are documented in Chapter 18.

Support of the architectural LBR feature in a logical processor is reported in CPUID.(EAX=07H, ECX=0H):EDX[19]=1. When the architectural LBR feature is supported, capability details like the number of LBR records that are available is indicated in CPUID.1CH:EAX[7:0]. The number of LBR records available varies across processor generations, so software should only access the available LBR records indicated by CPUID.1CH:EAX[7:0].

Last Branch Records (LBRs) enable recording of software path history by logging taken branches and other control flow transfers within processor registers. Each LBR record or entry is comprised of three MSRs:

- **IA32_LBR_x_FROM_IP** — Holds the source IP of the operation.
- **IA32_LBR_x_TO_IP** — Holds the destination IP of the operation.
- **IA32_LBR_x_INFO** — Holds metadata for the operation, including mispredict, TSX, and elapsed cycle time information.

LBR records are stored in age order. The most recent LBR entry is stored in IA32_LBR_0_*, the next youngest in IA32_LBR_1_*, and so on. When an operation to be recorded completes (retires) with LBRs enabled (IA32_LBR_CTL.LBREn=1), older LBR entries are shifted in the LBR array by one entry, then a record of the new operation is written into entry 0. See Section 19.1.1 for the list of recorded operations.

The number of LBR entries available for recording operations is dictated by the value in IA32_LBR_DEPTH.DEPTH. By default, the DEPTH value matches the maximum number of LBRs supported by the processor, but software may opt to use fewer in order to achieve reduced context switch latency.

In addition to the LBRs, there is a single Last Event Record (LER). It records the last taken branch preceding the last exception, hardware interrupt, or software interrupt. Like LBRs, the LER is comprised of three MSRs (IA32_LER_FROM_IP, IA32_LER_TO_IP, IA32_LER_INFO), and is subject to the same dependencies on enabling and filtering.

Which operations are recorded in LBRs depends upon a series of factors:

- **Branch Type Filtering** — Software must opt in to the types of branches to be logged; see Section 19.1.2.3.
- **Current Privilege Level (CPL)** — LBRs can be filtered based on CPL; see Section 19.1.2.5.
- **LBR Freeze** — LBR and LER recording can be suspended by setting IA32_PERF_GLOBAL_STATUS.LBR_FRZ to 1. See Section 18.4.7 for details on LBR_FRZ.

On some implementations, recording LBRs may require constraining the number of operations that can complete in a cycle. As a result, on these implementations, enabling LBRs may have some performance overhead.

## 19.1 BEHAVIOR

### 19.1.1 Logged Operations

LBRs can log most control flow transfer operations.

The source IP recorded for a branch instruction is the IP of that instruction. For events that take place between instructions, the source IP recorded is the IP of the next sequential instruction.

The destination IP recorded is always the target of the branch or event, the next instruction that will execute.

The full list of operations and the respective IPs recorded is shown in Table 19-1.
19.1.2 Configuration

19.1.2.1 Enabling and Disabling
LBRs are enabled by setting IA32_LBR_CTL.LBREn to 1.
Some operations, such as entry to a secure mode like SMM or Intel SGX, can cause LBRs to be temporarily disabled. Other operations, such as debug exceptions or some SMX operations, disable LBRs and require software to re-enable them. Details on these interactions can be found in Section 19.1.4.

19.1.2.2 LBR Depth
The number of LBRs used by the processor can be constrained by modifying the IA32_LBR_DEPTH.DEPTH value. DEPTH defaults to the maximum number of LBRs supported by the processor. Allowed DEPTH values can be found in CPUID.1CH:EAX[7:0].
Reducing the LBR depth can result in improved performance, by reducing the number of LBRs that need to be read and/or context switched.
On a software write to IA32_LBR_DEPTH, all LBR entries are reset to 0. LERs are not impacted.
A RDMSR or WRMSR to any IA32_LBR_x_* MSRs, such that x ≥ DEPTH, will generate a #GP exception. Note that the XSAVES and XRSTORS instructions access only the LBRs associated with entries 0 to DEPTH-1.
By clearing the LBR entries on writes to IA32_LBR_DEPTH, and forbidding any software writes to LBRs ≥ DEPTH, it is thereby guaranteed that any LBR entries equal to or above DEPTH will have value 0.

19.1.2.3 Branch Type Enabling and Filtering
Software must opt in to the types of branches that are desired to be recorded. These elections are made in IA32_LBR_CTL; see Section 19.2. Branch type options are listed in Table 19-2; only those enabled will be recorded.

<table>
<thead>
<tr>
<th>Operation</th>
<th>FROM_IP</th>
<th>TO_IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taken Branch^1, Exception, INT3, INTn, INTO, TSX Abort</td>
<td>Current IP</td>
<td>Target IP</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Next IP</td>
<td>Target IP</td>
</tr>
<tr>
<td>INIT (BSP)</td>
<td>Next IP</td>
<td>Reset Vector</td>
</tr>
<tr>
<td>INIT (AP) + SIPI</td>
<td>Next IP</td>
<td>SIPI Vector</td>
</tr>
<tr>
<td>EENTER/ERESUME + EEXIT/AEX</td>
<td>Current IP</td>
<td>Target or Trampoline IP</td>
</tr>
<tr>
<td>RSM^2</td>
<td>Target IP</td>
<td>Target IP</td>
</tr>
<tr>
<td>#DB, #SMI, VM exit, VM entry</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Direct CALLs with displacement zero, for which the target is typically the next sequential IP, are not treated as taken branches by LBRs.
2. RSM is only recorded in LBRs when IA32_DEBUGCTL.FREEZE_WHILE_SMM is set to 0.
These encodings match those in IA32_LBR_x_INFO.BR_TYPE.

Control flow transfers that are not recorded include #DB, VM exit, VM entry, and #SMI.

### 19.1.2.4 Call-Stack Mode

The LBR array is, by default, treated as a ring buffer that captures control flow transitions. However, the finite depth of the LBR array can be limiting when profiling certain high-level languages (e.g., C++), where a transition of the execution flow is accompanied by a large number of leaf function calls. These calls to leaf functions, and their returns, are likely to displace the main execution context from the LBRs.

When call-stack mode is enabled, the LBR array can capture unfiltered call data normally, but as return instructions are executed the last captured branch (call) record is flushed from the LBRs in a last-in first-out (LIFO) manner. Thus, branch information pertaining to completed leaf functions will not be retained, while preserving the call stack information of the main line execution path.

Table-19.2. Branch Type Filtering Details

<table>
<thead>
<tr>
<th>Branch Type</th>
<th>Operations Recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>Jcc, <em>CXZ, and LOOP</em></td>
</tr>
<tr>
<td>NEAR_IND_JMP</td>
<td>JMP r/m*</td>
</tr>
<tr>
<td>NEAR_REL_JMP</td>
<td>JMP rel*</td>
</tr>
<tr>
<td>NEAR_IND_CALL</td>
<td>CALL r/m* (excluding CALLs to the next sequential IP)</td>
</tr>
<tr>
<td>NEAR_REL_CALL</td>
<td>CALL rel* (excluding CALLs to the next sequential IP)</td>
</tr>
<tr>
<td>NEAR_RET</td>
<td>RET (0C3H)</td>
</tr>
<tr>
<td>OTHER_BRANCH</td>
<td>JMP/CALL ptr*, JMP/CALL m*, RET (0C8H), SYS*, interrupts, exceptions (other than debug exceptions), IRET, INT3, INTn, INTO, TSX Abort, EENTER, ERESUME, EEXIT, AEX, INIT, SIPI, RSM</td>
</tr>
</tbody>
</table>

Call-stack mode should be used with branch type enabling configured to capture only CALLs (NEAR_REL_CALL and NEAR_IND_CALL) and RETs (NEAR_RET). When configured in this manner, the LBR array emulates a call stack, where CALLs are “pushed” and RETs “pop” them off the stack. If other branch types (JCC, NEAR_*_JMP, or OTHER_BRANCH) are enabled for recording with call-stack mode, LBR behavior may be undefined.

It is recommended that call-stack mode be used along with CPL filtering, by setting at most one of the OS and USR bits in the IA32_LBR_CTL MSR. Call-stack mode does not emulate the stack switch that can occur on CPL transitions, and hence monitoring all CPLs may result in a corrupted LBR call stack.

**Call-Stack Mode and LBR Freeze**

When IA32_DEBUGCTL.FREEZE_LBRS_ON_PMI=1, IA32_PERF_GLOBAL_STATUS.LBR_FRZ will be set to 1 when a PMI is pended. That will cause LBRs and LERs to cease recording branches until LBR_FRZ is cleared. Because there may be some “skid”, or instructions retiring, in between the PMI being pended and the PMI being taken, it is possible that some branches may be missing from the LBRs. In the case of call-stack mode, if a CALL or RET is missed, that can lead to confusing results where CALL entries fail to get “popped” off the stack, and RETs “pop” the wrong CALLs.

An alternative is to utilize CPL filtering to limit LBR recording to less privileged modes only (CPL>3) instead of using the FREEZE_LBRS_ON_PMI=1 feature. This will record branches in the "skid", but avoid recording any branches in the privilege level 0 handler.
19.1.2.5 CPL Filtering

Software must opt in to which CPL(s) will have branches recorded. If IA32_LBR_CTL.OS=1, then branches in CPL=0 can be recorded. If IA32_LBR_CTL.USR=1, then branches in CPL>0 can be recorded. For operations which change the CPL, the operation is recorded in LBRs only if the CPL at the end of the operation is enabled for LBR recording. In cases where the CPL transitions from a value that is filtered out to a value that is enabled for LBR recording, the FROM_IP address for the recorded CPL transition branch or event will be 0xFFFFFFFFFFFFFH.

19.1.3 Record Data

19.1.3.1 IP Fields

The source and destination IP values in IA32_LBR_x_[FROM|TO]_IP and IA32_LER_x_[FROM|TO]_IP may hold effective IPs or linear IPs (LIPs), depending on the processor generation. The effective IP is the offset from the CS base address, while LIP includes the CS base address. Which IP type is used is indicated in CPUID.1CH:EAX[bit 31]. The value read from this field will always be canonical. Note that this includes the case where a canonical violation (#GP) results from executing sequential code that runs precisely to the end of the lower canonical address space (where IP[63:MAXLINADDR-1] is 0, but IP[MAXLINADDR-2:0] is all ones). In this case, the FROM_IP will hold the lowest canonical address in the upper canonical space, such that IP[63:MAXLINADDR-1] is all ones, and IP[MAXLINADDR-2:0] is 0.

In some cases, due to CPL filtering, the FROM_IP of the recorded operation may be filtered out. In this case 0xFFFFFFFFFFFFFH will be recorded. See Section 19.1.2.5 for details.

Writes of these fields will be forced canonical, such that the processor ignores the value written to the upper bits (IP[63:MAXLINADDR-1]).

19.1.3.2 Branch Types

The IA32_LBR_x_INFO.BR_TYPE and IA32_LER_INFO.BR_TYPE fields encode the branch types as shown in Table 19-3.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Branch Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000B</td>
<td>COND</td>
</tr>
<tr>
<td>0011B</td>
<td>NEAR_IND_CALL</td>
</tr>
<tr>
<td>0010B</td>
<td>NEAR_REL_JMP</td>
</tr>
<tr>
<td>0011B</td>
<td>NEAR_REL_CALL</td>
</tr>
<tr>
<td>0101B</td>
<td>NEAR_RET</td>
</tr>
<tr>
<td>0100B</td>
<td>NEAR_REL_CALL</td>
</tr>
<tr>
<td>011xB</td>
<td>Reserved</td>
</tr>
<tr>
<td>1xxxB</td>
<td>OTHER_BRANCH</td>
</tr>
</tbody>
</table>

Table 19-3. IA32_LBR_x_INFO and IA32_LER_INFO Branch Type Encodings

For a list of branch operations that fall into the categories above, see Table 19-2. In future generations, BR_TYPE bits 2:0 may be used to distinguish between differing types of OTHER_BRANCH.

19.1.3.3 Cycle Time

Each time an operation is recorded in an LBR, the value of the LBR cycle timer is recorded in IA32_LBR_x_INFO.CYC_CNT. The LBR cycle timer is a saturating counter that counts at the processor clock rate. Each time an operation is recorded in an LBR, the counter is reset but continues counting.
There is an LBR cycle counter valid bit, IA32_LBR_x_INFO.CYC_CNT_VALID. When set, the CYC_CNT field holds a valid value, the number of elapsed cycles since the last operation recorded in an LBR (up to 0xFFFFH).

Some implementations may opt to reduce the granularity of the CYC_CNT field for larger values. The implication of this is that the least significant bits may be forced to 1 in cases where the count has reached some minimum threshold. It is guaranteed that this reduced granularity will never result in an inaccuracy of more than 10%.

19.1.3.4 Mispredict Information
IA32_LBR_x_INFO.MISPRED provides an indication of whether the recorded branch was predicted incorrectly by the processor. The bit is set if either the taken/not-taken direction of a conditional branch was mispredicted, or if the target of an indirect branch was mispredicted.

19.1.3.5 Intel® TSX Information
IA32_LBR_x_INFO.IN_TSX indicates whether the operation recorded retired during a TSX transaction. IA32_LBR_x_INFO.TSX_ABORT indicates that the operation is a TSX Abort.

19.1.4 Interaction with Other Processor Features

19.1.4.1 SMM
IA32_LBR_CTL.LBREn is saved and cleared on #SMI, and restored on RSM. As a result of disabling LBRs, the #SMI is not recorded. RSM is recorded only if IA32_DEBUGCTL.FREEZE_WHILE_SMM is set to 0, and the FROM_IP will be set to the same value as the TO_IP.

19.1.4.2 SMM Transfer Monitor (STM)
LBREn is not cleared on #SMI when it causes SMM VM exit. Instead, the STM should use the VMCS controls described in Section 19.1.4.3 to disable LBRs while in SMM, and to restore them on VM entries that exit SMM. On VMCALL to configure STM, IA32_LBR_CTL is cleared.

19.1.4.3 VMX
By default, LBR operation persists across VMX transitions. However, VMCS fields have been added to enable constraining LBR usage to within non-root operation only. See details in Table 19-4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Bit Position</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest IA32_LBR_CTL</td>
<td>Guest State Field</td>
<td>NA</td>
<td>The guest value of IA32_LBR_CTL is written to this field on all VM exits.</td>
</tr>
<tr>
<td>Load Guest IA32_LBR_CTL</td>
<td>Entry Control</td>
<td>21</td>
<td>When set, VM entry will write the value from the “Guest IA32_LBR_CTL” guest state field to IA32_LBR_CTL.</td>
</tr>
<tr>
<td>Clear IA32_LBR_CTL</td>
<td>Exit Control</td>
<td>26</td>
<td>When set, VM exit will clear IA32_LBR_CTL after the value has been saved to the “Guest IA32_LBR_CTL” guest state field.</td>
</tr>
</tbody>
</table>

To enable “guest-only” LBR use, a VMM should set both the “Load Guest IA32_LBR_CTL” entry control and the “Clear IA32_LBR_CTL” exit control. For “system-wide” LBR use, where LBRs remain enabled across host and guest(s), a VMM should keep both new VMCS controls clear.

VM entry checks that, if the “Load Guest IA32_LBR_CTL” entry control is 1, bits reserved in the IA32_LBR_CTL MSR must be 0 in the field for that register.
LAST BRANCH RECORDS

For additional information relating to VMX transitions, see Chapter 25, Chapter 27, and Chapter 28 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

19.1.4.4 Intel® SGX
On entry to an enclave, via EENTER or ERESUME, logging of LBR entries is suspended. On enclave exit, via EEXIT or AEX, logging resumes. The cycle counter will continue to run during enclave execution.
An exception to the above is made for opt-in debug enclaves. For such enclaves, LBR logging is not impacted.

19.1.4.5 Debug Exceptions
When a branch happens because of a #DB exception, IA32_LBR_CTL.LBREn is cleared. As a result, the operation is not recorded.

19.1.4.6 SMX
On GETSEC leaves SENTER or ENTERACCS, IA32_LBR_CTL is cleared. As a result, the operation is not recorded.

19.1.4.7 MWAIT
On an MWAIT that requests a C-state deeper than C1, IA32_LBR_x_* MSRs may be cleared to 0. IA32_LBR_CTL, IA32_LBR_DEPTH, and IA32_LER_* MSRs will be preserved.
For an MWAIT that enters a C-state equal to or less deep than C1, and all C-states that enter as a result of Hardware Duty Cycling (HDC), all LBR MSRs are preserved.

19.1.4.8 Processor Event-Based Sampling (PEBS)
PEBS records can be configured to include LBRs, by setting PEBS_DATA_CFG.LBREn[3]=1. The number of LBRs to include in the record is also configurable, via PEBS_DATA_CFG.NUM_LBRS[28:24]. For details on PEBS, see Section 20.9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
If NUM_LBRS is set to a value greater than LBR_DEPTH, then only LBR_DEPTH entries will be written into the PEBS record. Further, the Record Size field will be decreased to match the actual size of the record to be written, and the Record Format field will replace the value of NUM_LBRS with the value of LBR_DEPTH. These adjustments ensure that software is able to properly interpret the PEBS record.

19.2 MSRS
The MSRs that represent the LBR entries (IA32_LBR_x_[TO][FROM][INFO]) and the LER entry (IA32_LER_[TO][FROM][INFO]) do not fault on writes. Any address field written will force sign-extension based on the maximum linear address width supported by the processor, and any non-zero value written to undefined bits may be ignored such that subsequent reads return 0.
On a warm reset, all LBR MSRs, including IA32_LBR_DEPTH, have their values preserved. However, IA32_LBR_CTL.LBREn is cleared to 0, disabling LBRs. If a warm reset is triggered while the processor is in the C6 idle state, also known as warm init, all LBR MSRs will be reset to their initial values.
See Table 2-2 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, for details on LBR MSRs.

19.3 FAST LBR READ ACCESS
XSAVE provides a faster means than RDMSR for software to read all LBRs. When using XSAVE for reading LBRs rather than for context switch, software should take care to ensure that XSAVE does not write LBR state to an area of memory that has been or will be used by XRSTORS. This could corrupt INIT tracking.
19.4 OTHER IMPACTS

19.4.1 Branch Trace Store on Intel Atom® Processors

Branch Trace Store (BTS) on Intel Atom processors that support the architectural form of the LBR feature has dependencies on the LBR configuration. BTS will store out the LBR_0 (TOS) record each time a taken branch or event retires. If any filtering of LBRs is employed, or if LBRs are disabled, some duplicate entries may be stored by BTS. Like LBRs and LERs, BTS is suspended when IA32_PERF_GLOBAL_STATUS.LBR_FRZ is set to 1.

BTS will change to cease issuing branch records for direct near CALLs with displacement zero to align with LBR behavior.

19.4.2 IA32_DEBUGCTL

On processors that do not support model-specific LBRs, IA32_DEBUGCTL[bit 0] has no meaning. It can be written to 0 or 1, but reads will always return 0.

19.4.3 IA32_PERF_CAPABILITIES

On processors that do not support model-specific LBRs, IA32_PERF_CAPABILITIES.LBR_FMT will have the value 03FH.
Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

NOTE
Performance monitoring events can be found here: https://perfmon-events.intel.com/.
Additionally, performance monitoring event files for Intel processors are hosted by the Intel Open Source Technology Center. These files can be downloaded here: https://download.01.org/perfmon/.

20.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, and Intel processors based on Intel NetBurst microarchitecture are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Newer Intel processor generations support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 20.6.3, “Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture).” Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they can be found at: https://perfmon-events.intel.com/.

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and Interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 20.2.

See also:
— Section 20.2, “Architectural Performance Monitoring.”
— Section 20.3, “Performance Monitoring (Intel® Core™ Processors and Intel® Xeon® Processors).”
  • Section 20.3.1, “Performance Monitoring for Processors Based on Nehalem Microarchitecture.”
  • Section 20.3.2, “Performance Monitoring for Processors Based on Westmere Microarchitecture.”
  • Section 20.3.3, “Intel® Xeon® Processor E7 Family Performance Monitoring Facility.”
  • Section 20.3.4, “Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture.”
  • Section 20.3.5, “3rd Generation Intel® Core™ Processor Performance Monitoring Facility.”
Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T
7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture. Intel Atom processors based on the Goldmont and Goldmont Plus microarchitectures support versionID 4. Intel Atom processors starting with processors based on the Tremont microarchitecture support versionID 5.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microarchitectures support version ID 3. Intel processors based on the Skylake through Coffee Lake microarchitectures support versionID 4. Intel processors starting with processors based on the Ice Lake microarchitecture support versionID 5.

### 20.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- The bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures. A non-zero write of a field that is introduced after the initial implementation of architectural performance monitoring (Version 1) results in #GP if that field is not supported.
- Addresses of IA32_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available to software in a logical processor (each IA32_PERFEVTSELx MSR is paired to the corresponding IA32_PMCx MSR).
- Number of bits supported in each IA32_PMCx.
- Number of architectural performance monitoring events supported in a logical processor.

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32_PERFEVTSELx/IA32_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

### 20.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8]. Note that this may vary from the number
of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

- IA32_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block. Note the number of IA32_PERFEVTSELx MSRs may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.

- The bit width of an IA32_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.

- Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally.

See Figure 20-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7)** — Selects the event logic unit used to detect microarchitectural conditions (see Table 20-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition. A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 20-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX.

- **USR (user mode) flag (bit 16)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.

- **OS (operating system mode) flag (bit 17)** — Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.

- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19)** — Beginning with Sandy Bridge microarchitecture, this bit is reserved (not writeable). On processors based on previous microarchitectures, the logical processor toggles the PMI pins and...
increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

- **INT (APIC interrupt enable) flag (bit 20)** — When set, the logical processor generates an exception through its local APIC on counter overflow.

- **EN (Enable Counters) Flag (bit 22)** — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.

- **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.

- **Counter mask (CMASK) field (bits 24 through 31)** — When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

### 20.2.1.2 Pre-defined Architectural Performance Events

Table 20-1 lists architecturally defined events.

<table>
<thead>
<tr>
<th>Bit Position CPUID.AH.EBX</th>
<th>Event Name</th>
<th>UMask</th>
<th>Event Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UnHalted Core Cycles</td>
<td>00H</td>
<td>3CH</td>
</tr>
<tr>
<td>1</td>
<td>Instruction Retired</td>
<td>00H</td>
<td>C0H</td>
</tr>
<tr>
<td>2</td>
<td>UnHalted Reference Cycles¹</td>
<td>01H</td>
<td>3CH</td>
</tr>
<tr>
<td>3</td>
<td>LLC Reference</td>
<td>4FH</td>
<td>2EH</td>
</tr>
<tr>
<td>4</td>
<td>LLC Misses</td>
<td>41H</td>
<td>2EH</td>
</tr>
<tr>
<td>5</td>
<td>Branch Instruction Retired</td>
<td>00H</td>
<td>C4H</td>
</tr>
<tr>
<td>6</td>
<td>Branch Misses Retired</td>
<td>00H</td>
<td>C5H</td>
</tr>
<tr>
<td>7</td>
<td>Topdown Slots</td>
<td>01H</td>
<td>A4H</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Implementations prior to the 12th generation Intel® Core™ processor P-cores count at core crystal clock, TSC, or bus clock frequency.

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 20-1). The number of architectural events is reported through CPUID.0AH:EAX[31:24], while non-zero bits in CPUID.0AH:EBX indicate any architectural events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles** — Event select 3CH, Umask 00H
  
  This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:
  
  - An ACPI C-state other than C0 for normal operation.
  - HLT.
  - STPCLK# pin asserted.
PERFORMANCE MONITORING

— Being throttled by TM1.
— During the frequency switching phase of a performance state transition (see Chapter 15, “Power and Thermal Management”).

The performance counter for this event counts across performance state transitions using different core clock frequencies.

• **Instructions Retired** — Event select C0H, Umask 00H
  This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

  This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

• **UnHalted Reference Cycles** — Event select 3CH, Umask 01H
  This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

• **Last Level Cache References** — Event select 2EH, Umask 4FH
  This event counts requests originating from the core that reference a cache line in the last level on-die cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

• **Last Level Cache Misses** — Event select 2EH, Umask 41H
  This event counts each cache miss condition for references to the last level on-die cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

• **Branch Instructions Retired** — Event select C4H, Umask 00H
  This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

• **All Branch Mispredict Retired** — Event select C5H, Umask 00H
  This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

  Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

• **Topdown Slots** — Event select A4H, Umask 01H
  This event counts the total number of available slots for an unhalted logical processor.

  The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core, in processors that support Intel Hyper-Threading Technology.

  Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.

**NOTE**

Programming decisions or software precisans on functionality should not be based on the event values or dependent on the existence of performance monitoring events.
20.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32_FIXED_CTR0 through IA32_FIXED_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

  Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32_FIXED_CTR_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32_PMCx) via UMASK field in (IA32_PERFEVTSELx), configuring, programming IA32_FIXED_CTR_CTRL for fixed-function PMCs do not require any UMASK.

- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - IA32_PERF_GLOBAL_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or any general-purpose PMCs via a single WRMSR.
  - IA32_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - IA32_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.

- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - IA32_DEBUGCTL.Freeze_LBR_On_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 18.4.7 for details of the legacy Freeze LBRs on PMI control.
  - IA32_DEBUGCTL.Freeze_PerfMon_On_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 18.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

**NOTE**

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32_FIXED_CTR_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 20-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:
**Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.

**PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 20-3 shows the layout of IA32_PERF_GLOBAL_CTRL. Each enable bit in IA32_PERF_GLOBAL_CTRL is AND’ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND’ed results is true; counting is disabled when the result is false.

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.
PERFORMANCE MONITORING

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 20-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status, and sets the “OvfBuffer” bit in IA32_PERF_GLOBAL_STATUS.

Table 20-2. Association of Fixed-Function Performance Counters with Architectural Performance Events

<table>
<thead>
<tr>
<th>Fixed-Function Performance Counter</th>
<th>Address</th>
<th>Event Mask Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_FIXED_CTR0</td>
<td>309H</td>
<td>INST_RETIRED.ANY</td>
<td>This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.</td>
</tr>
<tr>
<td>IA32_FIXED_CTR1</td>
<td>30AH</td>
<td>CPU_CLK_UNHALTED.THREAD, CPU_CLK_UNHALTED.CORE</td>
<td>The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.</td>
</tr>
<tr>
<td>IA32_FIXED_CTR2</td>
<td>30BH</td>
<td>CPU_CLK_UNHALTED.REF_TSC</td>
<td>This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.</td>
</tr>
<tr>
<td>IA32_FIXED_CTR3</td>
<td>30CH</td>
<td>TOPDOWN.SLOTS</td>
<td>This event counts the number of available slots for an unhalted logical processor. The event increments by machine-width of the narrowest pipeline as employed by the Top-down Microarchitecture Analysis method. The count is distributed among unhalted logical processors (hyper-threads) who share the same physical core. Software can use this event as the denominator for the top-level metrics of the Top-down Microarchitecture Analysis method.</td>
</tr>
</tbody>
</table>

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 20-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status, and sets the “OvfBuffer” bit in IA32_PERF_GLOBAL_STATUS.
PERFORMANCE MONITORING

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32_PERF_GLOBAL_OVF_CTL is shown in Figure 20-5.

20.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e., a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- AnyThread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  - Each IA32_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 20-6.
Bit 21 (AnyThread) of IA32_PERFEVTSELx is supported in architectural performance monitoring version 3 for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions (including matching the thread’s CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread’s CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring in the logical processor which programmed the IA32_PERFEVTSELx MSR.

Each fixed-function performance counter IA32_FIXED_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32_PERF_FIXED_CTR_CTRL MSR. The control block also allows thread-specificity configuration using an AnyThread bit for fixed-function counters 0, 1, and 2. The layout of IA32_PERF_FIXED_CTR_CTRL MSR is shown.

Each control block for a fixed-function performance counter provides an AnyThread (bit position 2 + 4*N, N=0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread’s CPL with the ENABLE setting of the corresponding control block of IA32_PERF_FIXED_CTR_CTRL) occurring across all logical processors sharing a processor core. When an AnyThread bit is 0 in IA32_PERF_FIXED_CTR_CTRL, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the IA32_PERF_FIXED_CTR_CTRL MSR.

The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 20-8 and Figure 20-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.
The number of general-purpose performance monitoring counters (i.e., $N$ in Figure 20-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g., $N=2$ for 45 nm Intel Atom processors; $N=4$ for processors based on the Nehalem microarchitecture; for processors based on the Sandy Bridge microarchitecture, $N=4$ if Intel Hyper Threading Technology is active and $N=8$ if not active). In addition, the number of counters may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters.
20.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL. While AnyThread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 24, "Introduction to Virtual Machine Extensions") where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow an individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- Configure the MSR bitmap to cause VM-exits for WRMSR to IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in VMX non-Root operation (see Chapter 25 for additional information),
- Clear the AnyThread bit of IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in the MSR-load lists for VM exits and VM entries (see Chapter 25, Chapter 27, and Chapter 28).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted.

20.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32_DEBUGCTL.Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI. Specifically version 4 provides the following enhancements:

- New indicators (LBR_FRZ, CTR_FRZ) in IA32_PERF_GLOBAL_STATUS, see Section 20.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32_DEBUGCTL.Freeze_LBRs_On_PMI and IA32_DEBUGCTL.Freeze_PerfMon_On_PMI: see Section 20.2.4.1. Legacy semantics of Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32_PERF_GLOBAL_STATUS and read-only performance counter enabling interface in IA32_PERF_GLOBAL_STATUS: see Section 20.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

20.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS

The IA32_PERF_GLOBAL_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32_PERF_GLOBAL_STATUS.LBR_FRZ[bit 58]: This bit is set due to the following conditions:
  - IA32_DEBUGCTL.FREEZE_LBR_ON_PMI has been set by the profiling agent, and
  - A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.LBR_FRZ bit also serves as a control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:

    \((\text{IA32_DEBUGCTL.LBR} \& (\neg \text{IA32_PERF_GLOBAL_STATUS.LBR_FRZ}) ) = 1\)

- IA32_PERF_GLOBAL_STATUS.CTR_FRZ[bit 59]: This bit is set due to the following conditions:
— IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI has been set by the profiling agent, and
— A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

Effectively, the IA32_PERF_GLOBAL_STATUS.CTR_FRZ bit also serve as a read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:

- \( (IA32\_PERF\_EVTSELn.\text{EN} \& IA32\_PERF\_GLOBAL\_CTRL.PMCn \& (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ) ) = 1 \) for programmable counter ‘n’, or
- \( (IA32\_PERF\_FIXED\_CTRL.\text{ENi} \& IA32\_PERF\_GLOBAL\_CTRL.FCi \& (!IA32\_PERF\_GLOBAL\_STATUS.CTR\_FRZ) ) = 1 \) for fixed counter ‘i’

The read-only enable interface IA32_PERF_GLOBAL_STATUS.CTR_FRZ provides a more efficient flow for a PMI handler to use IA32_DEBUGCTL.Freeze_Perfmon_On_PMI to filter out data that may distort target workload analysis, see Table 18-3. It should be noted the IA32_PERF_GLOBAL_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32_PERF_GLOBAL_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls, and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g., race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32_PERF_GLOBAL_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support it (for additional information about Intel SGX, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D):

- IA32_PERF_GLOBAL_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e., IA32_PMCx or IA32_FIXED_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32_PERF_GLOBAL_STATUS.ASCI was cleared).

Note, a processor’s support for IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32_PERF_GLOBAL_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.
### 20.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32_PERF_GLOBAL_STATUS MSR by software is done via IA32_PERF_GLOBAL_OVF_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32_PERF_GLOBAL_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32_PERF_GLOBAL_OVF_CTRL is inherited by IA32_PERF_GLOBAL_STATUS_RESET (see Figure 20-11). Further, IA32_PERF_GLOBAL_STATUS_RESET provides additional bit fields to clear the new indicators in IA32_PERF_GLOBAL_STATUS described in Section 20.2.4.1.

![Figure 20-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural Perfmon Version 4](image)

The IA32_PERF_GLOBAL_STATUS_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32_PERF_GLOBAL_STATUS. The IA32_PERF_GLOBAL_STATUS_SET interface can be used by a VMM to virtualize the state of IA32_PERF_GLOBAL_STATUS across VMs.

![Figure 20-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural Perfmon Version 4](image)

### 20.2.4.3 IA32_PERF_GLOBAL_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor’s performance monitoring facilities. The IA32_MISC_ENABLE.PERFMON_AVAILABLE[bit 7] interface could not serve
the need of multiple agent adequately. A white paper, "Performance Monitoring Unit Sharing Guideline"\(^1\), proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32_PERF_GLOBAL_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32_PERF_GLOBAL_INUSE is shown in Figure 20-13.

The IA32_PERF_GLOBAL_INUSE MSR provides an “InUse” bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- **IA32_PERF_GLOBAL_INUSE.PERFEVTSEL0_InUse[bit 0]**: This bit reflects the logical state of (IA32_PERFVTSEL0[7:0] != 0).
- **IA32_PERF_GLOBAL_INUSE.PERFEVTSEL1_InUse[bit 1]**: This bit reflects the logical state of (IA32_PERFVTSEL1[7:0] != 0).
- **IA32_PERF_GLOBAL_INUSE.PERFEVTSEL2_InUse[bit 2]**: This bit reflects the logical state of (IA32_PERFVTSEL2[7:0] != 0).
- **IA32_PERF_GLOBAL_INUSE.PERFEVTSELn_InUse[bit n]**: This bit reflects the logical state of (IA32_PERFVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- **IA32_PERF_GLOBAL_INUSE.FC0_InUse[bit 32]**: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[1:0] != 0).
- **IA32_PERF_GLOBAL_INUSE.FC1_InUse[bit 33]**: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[5:4] != 0).
- **IA32_PERF_GLOBAL_INUSE.FC2_InUse[bit 34]**: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[9:8] != 0).
- **IA32_PERF_GLOBAL_INUSE.PMI_InUse[bit 63]**: This bit is set if any one of the following bit is set:
  - IA32_PERFVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8].
  - IA32_FIXED_CTR_CTRL.ENi_PMI, i = 0, 1, 2.
  - Any IA32_PEBS_ENABLES bit which enables PEBS for a general-purpose or fixed-function performance counter.

---

1. Available at http://www.intel.com/sdm
20.2.5 Architectural Performance Monitoring Version 5

Processors supporting architectural performance monitoring version 5 also support versions 1, 2, 3, and 4, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 5 provides the following enhancements:

- Deprecation of AnyThread mode, see Section 20.2.5.1.
- Individual enumeration of Fixed counters in CPUID.0AH, see Section 20.2.5.2.
- Domain separation, see Section 20.2.5.3.

20.2.5.1 AnyThread Mode Deprecation

With Architectural Performance Monitoring Version 5, a processor that supports AnyThread mode deprecation is enumerated by CPUID.0AH.EDX[15]. If set, software will not have to follow guidelines in Section 20.2.3.1.

20.2.5.2 Fixed Counter Enumeration

With Architectural Performance Monitoring Version 5, register CPUID.0AH.ECX indicates Fixed Counter enumeration. It is a bit mask which enumerates the supported Fixed Counters in a processor. If bit 'i' is set, it implies that Fixed Counter 'i' is supported. Software is recommended to use the following logic to check if a Fixed Counter is supported on a given processor:

\[
\text{FxCtr[i]_is_supported} := \text{ECX}[i] || (\text{EDX}[4:0] > i);
\]

20.2.5.3 Domain Separation

When the INV flag in IA32_PERFEVTSELx is used, a counter stops counting when the logical processor exits the C0 ACPI C-state.

20.2.6 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability. See Figure 20-65.

If IA32_PERF_CAPABILITIES.FW_WRITE[bit 13] =1, each IA32_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32_A_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32_A_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32_A_PMCi will cause IA32_PMCi to be updated by:

\[
\text{COUNTERWIDTH} = \text{CPUID.0AH:EAX}[23:16] \text{ bit width of the performance monitoring counter}
\]

\[
\text{IA32_PMCi[COUNTERWIDTH-1:32]} := \text{EDX}[\text{COUNTERWIDTH-33:0})];
\]

\[
\text{IA32_PMCi[31:0]} := \text{EAX}[31:0];
\]

\[
\text{EDX}[63:\text{COUNTERWIDTH}] \text{ are reserved}
\]

20.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS)

20.3.1 Performance Monitoring for Processors Based on Nehalem Microarchitecture

Intel Core i7 processor family supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based...
PERFORMANCE MONITORING

on Nehalem microarchitecture, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as “uncore”.
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3.

![Figure 20-14. IA32_PERF_GLOBAL_STATUS MSR](image)

20.3.1.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32_PMCx, associated counter configuration MSRs, IA32_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Nehalem

2. Intel Xeon processor 5500 series and 3400 series are also based on Nehalem microarchitecture; the performance monitoring facilities described in this section generally also apply.
microarchitecture has been enhanced to include new data format to capture additional information, such as load latency.

- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Nehalem microarchitecture. This field measures the load latency from load’s first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.

- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

NOTE

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 20.2.1.

20.3.1.1.1 Processor Event Based Sampling (PEBS)

All general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEB_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEB_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEB_ENABLE for processors based on Nehalem microarchitecture is shown in Figure 20-15.

When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32_PERF_CAPABILITIES[11:8] (see Figure 20-65).
The behavior of PEBS assists is reported by IA32_PERF_CAPABILITIES[6] (see Figure 20-65). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

```
mov eax, [eax] ; causes PEBS assist
nop
```

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 20-3, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor’s DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

### Table 20-3. PEBS Record Format for Intel Core i7 Processor Family

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field</th>
<th>Byte Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>R/EFLAGS</td>
<td>58H</td>
<td>R9</td>
</tr>
<tr>
<td>08H</td>
<td>R/EIP</td>
<td>60H</td>
<td>R10</td>
</tr>
<tr>
<td>10H</td>
<td>R/EAX</td>
<td>68H</td>
<td>R11</td>
</tr>
<tr>
<td>18H</td>
<td>R/EBX</td>
<td>70H</td>
<td>R12</td>
</tr>
<tr>
<td>20H</td>
<td>R/ECX</td>
<td>78H</td>
<td>R13</td>
</tr>
<tr>
<td>28H</td>
<td>R/EDX</td>
<td>80H</td>
<td>R14</td>
</tr>
<tr>
<td>30H</td>
<td>R/ESI</td>
<td>88H</td>
<td>R15</td>
</tr>
<tr>
<td>38H</td>
<td>R/EDI</td>
<td>90H</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
</tr>
<tr>
<td>40H</td>
<td>R/EBP</td>
<td>98H</td>
<td>Data Linear Address</td>
</tr>
<tr>
<td>48H</td>
<td>R/ESP</td>
<td>A0H</td>
<td>Data Source Encoding</td>
</tr>
<tr>
<td>50H</td>
<td>R8</td>
<td>A8H</td>
<td>Latency value (core cycles)</td>
</tr>
</tbody>
</table>

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32_PERF_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32_PERF_GLOBAL_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

### Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 20-84. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in the IA32_PBS_ENABLE register for the respective counter, the software must also initialize the DS_BUFFER_MANAGEMENT_AREA data structure in memory to support capturing PEBS records for precise events.

The recording of PEBS records may not operate properly if accesses to the linear addresses in the DS buffer management area or in the PEBS buffer (see below) cause page faults, VM exits, or the setting of accessed or dirty flags in the paging structures (ordinary or EPT). For that reason, system software should establish paging structures (both ordinary and EPT) to prevent such occurrences. Implications of this may be that an operating system should allocate this memory from a non-paged pool and that system software cannot do “lazy” page-table entry propagation for these pages. A virtual-machine monitor may choose to allow use of PEBS by guest software only if EPT maps all guest-physical memory as present and read/write.
NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS_BUFFER_MANAGEMENT_AREA data structure must be programmed into the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 20-16.

- **PEBS Buffer Base**: This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer.

- **PEBS Index**: This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.

- **PEBS Absolute Maximum**: This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when PEBS Index equals PEBS Absolute Maximum. No signaling is generated when PEBS buffer is full. Software must reset the PEBS Index field to the beginning of the PEBS buffer address to continue capturing PEBS records.

---

**Figure 20-16. PEBS Programming Environment**
PERFORMANCE MONITORING

- **PEBS Interrupt Threshold**: This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the PEBS Index reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the IA32_PERF_GLOBAL_STATUS.PEBS_Ovf bit will be set.

- **PEBS CounterX Reset**: This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32_PEBS_ENABLED was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter IA32_PMC0 caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter IA32_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

**Performance Counter Prioritization**

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming IA32_PerfEvtSelx.INT is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.

Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32_PMC0 takes precedence over all other counters. Counter IA32_PMC1 takes precedence over counters IA32_PMC2 and IA32_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32_PMC1 cause an overflow interrupt and IA32_PMC2 causes a PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 20.3.1.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

**20.3.1.2 Load Latency Performance Monitoring Facility**

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 20-3. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_INST_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelx[15:0] = 100H). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelx register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).

- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 20-3, bytes AFH:98H) consists of:

- **Data Linear Address**: This is the linear address of the target of the load operation.
- **Latency Value**: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.
- **Data Source**: The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 20-4. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory refers to system memory physically attached to another processor package.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>Unknown L3 cache miss.</td>
</tr>
<tr>
<td>01H</td>
<td>Minimal latency core cache hit. This request was satisfied by the L1 data cache.</td>
</tr>
<tr>
<td>02H</td>
<td>Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.</td>
</tr>
<tr>
<td>03H</td>
<td>This data request was satisfied by the L2.</td>
</tr>
<tr>
<td>04H</td>
<td>L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).</td>
</tr>
<tr>
<td>05H</td>
<td>L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found. (clean).</td>
</tr>
<tr>
<td>06H</td>
<td>L3 HIT. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found.</td>
</tr>
<tr>
<td>07H</td>
<td>Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and were serviced by another core with a cross core snoop where modified copies were found.</td>
</tr>
<tr>
<td>08H</td>
<td>Reserved/L3 MISS. Local homed requests that missed the L3 cache and were serviced by forwarded data following a cross package snoop where no modified copies were found. (Remote home requests are not counted).</td>
</tr>
<tr>
<td>09H</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0AH</td>
<td>L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).</td>
</tr>
<tr>
<td>0BH</td>
<td>L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to shared state).</td>
</tr>
<tr>
<td>0CH</td>
<td>L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to exclusive state).</td>
</tr>
<tr>
<td>0DH</td>
<td>L3 MISS. Remote home requests that missed the L3 cache and were serviced by remote DRAM (go to exclusive state).</td>
</tr>
<tr>
<td>0EH</td>
<td>I/O, Request of input/output operation.</td>
</tr>
<tr>
<td>0FH</td>
<td>The request was to un-cacheable memory.</td>
</tr>
</tbody>
</table>
NOTES:
1. Bit 7 is supported only for processors with a CPUID DisplayFamily_DisplayModel signature of 06_2A, and 06_2E; otherwise it is reserved.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 20-17.

![Figure 20-17. Layout of MSR_PEBS_LD_LAT MSR](image)

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

20.3.1.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32_PERFEVTSELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_0. There is only one off-core response configuration MSR. Table 20-5 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 20-5. Off-Core Response Event Encoding

<table>
<thead>
<tr>
<th>Event code in IA32_PERFEVTSELx</th>
<th>Mask Value in IA32_PERFEVTSELx</th>
<th>Required Off-core Response MSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7H</td>
<td>01H</td>
<td>MSR_OFFCORE_RSP_0 (address 1A6H)</td>
</tr>
</tbody>
</table>

The layout of MSR_OFFCORE_RSP_0 is shown in Figure 20-18. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.
Table 20-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMND_DATA_RD</td>
<td>0</td>
<td>Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.</td>
</tr>
<tr>
<td>DMND_RFO</td>
<td>1</td>
<td>Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.</td>
</tr>
<tr>
<td>DMND_IFETCH</td>
<td>2</td>
<td>Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.</td>
</tr>
<tr>
<td>WB</td>
<td>3</td>
<td>Counts the number of writeback (modified to exclusive) transactions.</td>
</tr>
<tr>
<td>PF_DATA_RD</td>
<td>4</td>
<td>Counts the number of data cacheline reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_RFO</td>
<td>5</td>
<td>Counts the number of RFO requests generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_IFETCH</td>
<td>6</td>
<td>Counts the number of code reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>OTHER</td>
<td>7</td>
<td>Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.</td>
</tr>
<tr>
<td>UNCORE_HIT</td>
<td>8</td>
<td>L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).</td>
</tr>
<tr>
<td>OTHER_CORE_HIT_SNP</td>
<td>9</td>
<td>L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoo where no modified copies were found (clean).</td>
</tr>
<tr>
<td>OTHER_CORE_HIT_TM</td>
<td>10</td>
<td>L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoo where modified copies were found (HITM).</td>
</tr>
<tr>
<td>Reserved</td>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>REMOTE_CACHE_FWD</td>
<td>12</td>
<td>L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoo where no modified copies found. (Remote home requests are not counted)</td>
</tr>
<tr>
<td>REMOTE_DRAM</td>
<td>13</td>
<td>L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.</td>
</tr>
<tr>
<td>LOCAL_DRAM</td>
<td>14</td>
<td>L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.</td>
</tr>
</tbody>
</table>
The “uncore” in Nehalem microarchitecture refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06_1AH, 06_1EH, 06_1FH (see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- **Eight General-purpose counters (MSR_UNCORE_PerfCntr0 through MSR_UNCORE_PerfCntr7).** The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR_UNCORE_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.

- **Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.**

- **One fixed-function counter, MSR_UNCORE_FixedCntr0.** The fixed-function uncore counter increments at the rate of the U-clock when enabled.

The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

### 20.3.1.2.1 Uncore Performance Monitoring Management Facility

MSR_UNCORE_PERF_GLOBAL_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 20-19 shows the layout of MSR_UNCORE_PERF_GLOBAL_CTRL for an uncore that is shared by four processor cores in a physical package.

- **EN_PCn** (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR_UNCORE_PerfCntr n.

- **EN_FC0** (bit 32): When set, enables counting for the fixed-function uncore counter MSR_UNCORE_FixedCntr0.

- **EN_PMI_COREn** (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32_DEBUGCTL.Offcore_PMI_EN to 1.

- **PMI_FRZ** (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR_UNCORE_PERF_GLOBAL_CTRL upon exit from the ISR.
MSR_UNCORE_PERF_GLOBAL_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 20-20 shows the layout of MSR_UNCORE_PERF_GLOBAL_STATUS.

- OVF_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR_UNCORE_PerfCntr n has overflowed.
- OVF_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR_UNCORE_FixedCntr0 has overflowed.
- OVF_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR_UNCORE_PERF_GLOBAL_STATUS register has changed state.

MSR_UNCORE_PERF_GLOBAL_OVF_CTRL allows software to clear the status bits in the UNCORE_PERF_GLOBAL_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

Figure 20-19. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

Vol. 3B 20-27
PERFORMANCE MONITORING

Figure 20-21 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>CLR_CHG (WO1)</td>
</tr>
<tr>
<td>62</td>
<td>CLR_OVF_PMI (WO1)</td>
</tr>
<tr>
<td>61</td>
<td>CLR_OVF_FC0 (WO1)</td>
</tr>
<tr>
<td>32</td>
<td>CLR_OVF_PC0 (R/O)</td>
</tr>
<tr>
<td>31</td>
<td>CLR_OVF_PC1 (R/O)</td>
</tr>
<tr>
<td>30</td>
<td>CLR_OVF_PC2 (R/O)</td>
</tr>
<tr>
<td>29</td>
<td>CLR_OVF_PC3 (R/O)</td>
</tr>
<tr>
<td>28</td>
<td>CLR_OVF_PC4 (R/O)</td>
</tr>
<tr>
<td>27</td>
<td>CLR_OVF_PC5 (R/O)</td>
</tr>
<tr>
<td>26</td>
<td>CLR_OVF_PC6 (R/O)</td>
</tr>
<tr>
<td>25</td>
<td>CLR_OVF_PC7 (R/O)</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
```

RESET Value — 00000000_00000000H

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>CHG (R/W)</td>
</tr>
<tr>
<td>62</td>
<td>OVF_PMI (R/W)</td>
</tr>
<tr>
<td>61</td>
<td>OVF_FC0 (R/O)</td>
</tr>
<tr>
<td>32</td>
<td>OVF_PC0 (R/O)</td>
</tr>
<tr>
<td>31</td>
<td>OVF_PC1 (R/O)</td>
</tr>
<tr>
<td>30</td>
<td>OVF_PC2 (R/O)</td>
</tr>
<tr>
<td>29</td>
<td>OVF_PC3 (R/O)</td>
</tr>
<tr>
<td>28</td>
<td>OVF_PC4 (R/O)</td>
</tr>
<tr>
<td>27</td>
<td>OVF_PC5 (R/O)</td>
</tr>
<tr>
<td>26</td>
<td>OVF_PC6 (R/O)</td>
</tr>
<tr>
<td>25</td>
<td>OVF_PC7 (R/O)</td>
</tr>
<tr>
<td>63</td>
<td>CLR_OVF_PMI (WO1)</td>
</tr>
<tr>
<td>62</td>
<td>CLR_OVF_FC0 (WO1)</td>
</tr>
<tr>
<td>61</td>
<td>CLR_OVF_PC0 (WO1)</td>
</tr>
<tr>
<td>32</td>
<td>CLR_OVF_PC1 (WO1)</td>
</tr>
<tr>
<td>31</td>
<td>CLR_OVF_PC2 (WO1)</td>
</tr>
<tr>
<td>30</td>
<td>CLR_OVF_PC3 (WO1)</td>
</tr>
<tr>
<td>29</td>
<td>CLR_OVF_PC4 (WO1)</td>
</tr>
<tr>
<td>28</td>
<td>CLR_OVF_PC5 (WO1)</td>
</tr>
<tr>
<td>27</td>
<td>CLR_OVF_PC6 (WO1)</td>
</tr>
<tr>
<td>26</td>
<td>CLR_OVF_PC7 (WO1)</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
```

RESET Value — 00000000_00000000H

20.3.1.2.2 Uncore Performance Event Configuration Facility

MSR_UNCORE_PerfEvtSel0 through MSR_UNCORE_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corre-
sponding MSR_UNCORE_PerfEvtSelx and counting must be enabled using the respective EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL. Figure 20-22 shows the layout of MSR_UNCORE_PERFEVTSELx.

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8): Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC_CTR_RST (bit 17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 20-23 shows the layout of MSR_UNCORE_FIXED_CTR_CTRL.

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN_FC0 bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.
Both the general-purpose counters (MSR_UNCORE_PerfCntr) and the fixed-function counter (MSR_UNCORE_FixedCntr0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

### 20.3.1.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR_UNCORE_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR_OPCODE_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR_UNCORE_ADDR_OPCODE_MATCH register. The layout is shown in Figure 20-24.

- **Addr (bits 39:3)**: The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- **Opcode (bits 47:40)**: Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - Bits 43:40 specify the QPI packet header opcode.
  - Bits 47:44 specify the QPI message classes.

Table 20-7 lists the encodings supported in the opcode field.

<table>
<thead>
<tr>
<th>Opcode [43:40]</th>
<th>QPI Message Class</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Home Request</td>
<td>Snoop Response</td>
<td>Data Response</td>
</tr>
<tr>
<td>DMND_IFETCH</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PF_DATA_RD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PF_RFO</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>PF_IFETCH</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>OTHER</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>NON_DRAM</td>
<td>15</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
• MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  — 000B: Disable addr_opcode match hardware.
  — 100B: Count if only the address field matches.
  — 010B: Count if only the opcode field matches.
  — 110B: Count if either opcode field matches or the address field matches.
  — 001B: Count only if both opcode and address field match.
  — Other encoding are reserved.

20.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 20-25.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

Table 20-8 summarizes the number MSRs for uncore PMU for each box.
The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the "global control" programming interface, IA32_PERF_GLOBAL_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR_U_PMON_GLOBAL_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 2, "Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, Table 2-17 under the general naming style of MSR_%box#%_PMON_%scope_function%, where %box#% designates the type of box and zero-based index if there are more one box of the same type, %scope_function% follows the examples below:

- Multi-counter enabling MSRs: MSR_U_PMON_GLOBAL_CTL, MSR_S0_PMON_BOX_CTL, MSR_C7_PMON_BOX_CTL, etc.
- Multi-counter status MSRs: MSR_U_PMON_GLOBAL_STATUS, MSR_S0_PMON_BOX_STATUS, MSR_C7_PMON_BOX_STATUS, etc.
- Multi-counter overflow control MSRs: MSR_U_PMON_GLOBAL_OVF_CTL, MSR_S0_PMON_BOX_OVF_CTL, MSR_C7_PMON_BOX_OVF_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g., MSR_U_PMON_CTR, MSR_S0_PMON_CTR0, MSR_C7_PMON_CTR5, etc.
- Event select MSRs: the scope is implicitly per-counter, e.g., MSR_U_PMON_EVNT_SEL, MSR_S0_PMON_EVNT_SEL0, MSR_C7_PMON_EVNT_SEL5, etc.
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g., MSR_M0_PMON_TIMESTAMP, MSR_R0_PMON_IPERF0_P1, MSR_S1_PMON_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document "Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide”.

### 20.3.2 Performance Monitoring for Processors Based on Westmere Microarchitecture

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 20.6.3 also apply to processors based on Westmere microarchitecture.

Table 20-5 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR_OFFCORE_RSP_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore
response event using event code 0BBH and MSR_OFFCORE_RSP_1 (address 1A7H) are supported in processors based on Westmere microarchitecture. The event code and event mask definitions of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

The load latency facility is the same as described in Section 20.3.1.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB_MISS and LOCK, see Table 20-13.

### 20.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore sub-system is similar to that shown in Figure 20-25, with the additional capability that up to 10 C-Box units are supported.

Table 20-9 summarizes the number MSRs for uncore PMU for each box.

#### Table 20-9. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

<table>
<thead>
<tr>
<th>Box</th>
<th># of Boxes</th>
<th>Counters per Box</th>
<th>Counter Width</th>
<th>General Purpose</th>
<th>Global Enable</th>
<th>Sub-control MSRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Box</td>
<td>10</td>
<td>6</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>None</td>
</tr>
<tr>
<td>S-Box</td>
<td>2</td>
<td>4</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>Match/Mask</td>
</tr>
<tr>
<td>B-Box</td>
<td>2</td>
<td>4</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>Match/Mask</td>
</tr>
<tr>
<td>M-Box</td>
<td>2</td>
<td>6</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>Yes</td>
</tr>
<tr>
<td>R-Box</td>
<td>1</td>
<td>16 (2 port, 8 per port)</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>Yes</td>
</tr>
<tr>
<td>W-Box</td>
<td>1</td>
<td>4</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>48</td>
<td>No</td>
<td>per-box</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U-Box</td>
<td>1</td>
<td>1</td>
<td>48</td>
<td>Yes</td>
<td>uncore</td>
<td>None</td>
</tr>
</tbody>
</table>

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the “Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual”.

### 20.3.4 Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Sandy Bridge microarchitecture; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 20.2.3.

The core PMU’s capability is similar to those described in Section 20.3.1.1 and Section 20.6.3, with some differences and enhancements relative to Westmere microarchitecture summarized in Table 20-10.

3. Exceptions are indicated for event code 0FH in the event list for this processor (https://perfmon-events.intel.com/); and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 20-13.
20.3.4.1 Global Counter Control Facilities in Sandy Bridge Microarchitecture

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Sandy Bridge microarchitecture. Software must use CPUID to determine the number performance counters/event select registers (See Section 20.2.1.1).

### Table 20-10. Core PMU Comparison

<table>
<thead>
<tr>
<th>Box</th>
<th>Sandy Bridge Microarchitecture</th>
<th>Westmere Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Fixed counters per thread</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of general-purpose counters per core</td>
<td>8</td>
<td>8</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of programmable counters per thread</td>
<td>4 or (8 if a core not shared by two threads)</td>
<td>4</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
</tbody>
</table>
| PMU Overhead Mitigation | • Freeze_Perfmon_on_PMI with legacy semantics.  
• Freeze_LBR_on_PMI with legacy semantics for branch profiling.  
• Freeze_while_SMM. | • Freeze_Perfmon_on_PMI with legacy semantics.  
• Freeze_LBR_on_PMI with legacy semantics for branch profiling.  
• Freeze_while_SMM. | See Section 18.4.7. |
| Processor Event Based Sampling (PEBS) Events | See Table 20-12. | See Table 20-84. | IA32_PMC4-IA32_PMC7 do not support PEBS. |
| PEBS-Load Latency | See Section 20.3.4.4.2;  
• Data source encoding  
• STLB miss encoding  
• Lock transaction encoding | Data source encoding |
| PEBS-Precise Store | Section 20.3.4.4.3 | No |
| PEBS-PDIR | Yes (using precise INST_RETIRED.ALL). | No |
| Off-core Response Event | MSR 1A6H and 1A7H, extended request and response types. | MSR 1A6H and 1A7H, limited response types. | Nehalem supports 1A6H only. |

20.3.4.1 Global Counter Control Facilities in Sandy Bridge Microarchitecture

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Sandy Bridge microarchitecture. Software must use CPUID to determine the number performance counters/event select registers (See Section 20.2.1.1).
Figure 20-44 depicts the layout of IA32_PERF_GLOBAL_CTRL MSR. The enable bits (PMC4_EN, PMC5_EN, PMC6_EN, PMC7_EN) corresponding to IA32_PMC4-IA32_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of ‘8’. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32_PERF_GLOBAL_CTRL is AND’ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND’ed results is true; counting is disabled when the result is false.

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 20-27). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

![Figure 20-27. IA32_PERF_GLOBAL_STATUS MSR in Sandy Bridge Microarchitecture](image)

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 18.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 20-28). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling.
- Reloading counter values to continue sampling.
- Disabling event counting or interrupt based sampling.
20.3.4.2 Counter Coalescence

In processors based on Sandy Bridge microarchitecture, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report the number of counters visible to software.

If a processor core is shared by two logical processors, each logical processor can access up to four counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Nehalem microarchitecture.

If a processor core is not shared by two logical processors, up to eight general-purpose counters are visible. If CPUID.0AH:EAX[15:8] reports 8 counters, then IA32_PMC4-IA32_PMC7 would occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_GLOBAL_OVF_CTL will also cause #GP.

20.3.4.3 Full Width Writes to Performance Counters

Processors based on Sandy Bridge microarchitecture support full-width writes to the general-purpose counters, IA32_PMCx. Support of full-width writes are enumerated by IA32_PERF_CAPABILITIES.FW_WRITES[13] (see Section 20.2.4).

The default behavior of IA32_PMCx is unchanged, i.e., WRMSR to IA32_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32_A_PMCx by testing IA32_PERF_CAPABILITIES[13].

20.3.4.4 PEBS Support in Sandy Bridge Microarchitecture

Processors based on Sandy Bridge microarchitecture support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Westmere microarchitecture is summarized in Table 20-11.
Performance Monitoring

Only IA32_PMC0 through IA32_PMC3 support PEBS.

**NOTE**

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32_PEPS_ENABLE MSR, bit 63 is defined as PS_ENABLE: When set, this enables IA32_PMC3 to capture precise store information. Only IA32_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32_PEPS_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.

---

**Table 20-11. PEBS Facility Comparison**

<table>
<thead>
<tr>
<th>Box</th>
<th>Sandy Bridge Microarchitecture</th>
<th>Westmere Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid IA32_PMCx</td>
<td>PMC0-PMC3</td>
<td>PMC0-PMC3</td>
<td>No PEBS on PMC4-PMC7.</td>
</tr>
<tr>
<td>PEBS Buffer Programming</td>
<td>Section 20.3.1.1.1</td>
<td>Section 20.3.1.1.1</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_PEPS_ENABLE</td>
<td>Figure 20-29</td>
<td>Figure 20-15</td>
<td></td>
</tr>
<tr>
<td>Layout</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PEBS record layout</td>
<td>Physical Layout same as Table 20-3.</td>
<td>Table 20-3</td>
<td>Enhanced fields at offsets 98H, A0H, A8H.</td>
</tr>
<tr>
<td>PEBS Events</td>
<td>See Table 20-12.</td>
<td>See Table 20-84.</td>
<td>IA32_PMC4-IA32_PMC7 do not support PEBS.</td>
</tr>
<tr>
<td>PEBS-Load Latency</td>
<td>See Table 20-13.</td>
<td>Table 20-4</td>
<td></td>
</tr>
<tr>
<td>PEBS-Precise Store</td>
<td>Yes; see Section 20.3.4.4.3.</td>
<td>No</td>
<td>IA32_PMC3 only</td>
</tr>
<tr>
<td>PEBS-PDIR</td>
<td>Yes</td>
<td>No</td>
<td>IA32_PMC1 only</td>
</tr>
<tr>
<td>PEBS skid from EventingIP</td>
<td>1 (or 2 if micro+macro fusion)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SAMPLING Restriction</td>
<td>Small SAV(CountDown) value incur higher overhead than prior generation.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only IA32_PMC0 through IA32_PMC3 support PEBS.
20.3.4.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 20-3, but the fields at offsets 98H, A0H, and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H): When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 20-4. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32_PERF_GLOBAL_STATUS corresponding to those counters that both overflowed and were enabled in the IA32_PEBS_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Sandy Bridge microarchitecture is shown in Table 20-12.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Select</th>
<th>Sub-event</th>
<th>UMask</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST_RETIRED</td>
<td>C0H</td>
<td>PREC_DIST</td>
<td>01H</td>
</tr>
<tr>
<td>UOPS_RETIRED</td>
<td>C2H</td>
<td>All</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Retire_Slots</td>
<td>02H</td>
</tr>
<tr>
<td>BR_INST_RETIRED</td>
<td>C4H</td>
<td>Conditional</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near_Call</td>
<td>02H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All_branches</td>
<td>04H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near_Return</td>
<td>08H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near_Taken</td>
<td>20H</td>
</tr>
<tr>
<td>BR_MISP_RETIRED</td>
<td>C5H</td>
<td>Conditional</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Near_Call</td>
<td>02H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All_branches</td>
<td>04H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not_Taken</td>
<td>10H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Taken</td>
<td>20H</td>
</tr>
</tbody>
</table>

Table 20-12. PEBS Performance Events for Sandy Bridge Microarchitecture
Load Latency Performance Monitoring Facility

The load latency facility in Sandy Bridge microarchitecture is similar to that in prior microarchitectures. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 20-3 and Section 20.3.4.4.1. This field measures the load latency from load’s first dispatch till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_TRANS_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.

- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).

- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.

- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM_TRANS_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

Vol. 3B 20-39
When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 20-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

<table>
<thead>
<tr>
<th>Field</th>
<th>Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>3:0</td>
<td>See Table 20-4</td>
</tr>
<tr>
<td>STLB_MISS</td>
<td>4</td>
<td>0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.</td>
</tr>
<tr>
<td>Lock</td>
<td>5</td>
<td>0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.</td>
</tr>
<tr>
<td>Reserved</td>
<td>63:6</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The layout of MSR_PEBS_LD_LAT_THRESHOLD is the same as shown in Figure 20-17.

20.3.4.4.3 Precise Store Facility

Processors based on Sandy Bridge microarchitecture offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.
- Program the MEM_TRANS_RETIRED.PRECISE_STORE event in IA32_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBs_ENABLE[3] and IA32_PEBs_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offsets 98H, A0H, and A8H of Table 20-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.
20.3.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable “skids” in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Sandy Bridge microarchitecture include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the “skid” problem by providing an early indication of when the INST_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow. On processors based on Sandy Bridge microarchitecture, skid is significantly reduced and can be as little as one instruction. On future implementations, PDIR may eliminate skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and processors based on Sandy Bridge microarchitecture must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily_DisplayModel signatures of 06_2A and 06_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

20.3.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Sandy Bridge microarchitecture provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Table 20-15 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

<table>
<thead>
<tr>
<th>Counter</th>
<th>Event code</th>
<th>UMask</th>
<th>Required Off-core Response MSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMC0-3</td>
<td>B7H</td>
<td>01H</td>
<td>MSR_OFFCORE_RSP_0 (address 1A6H)</td>
</tr>
<tr>
<td>PMC0-3</td>
<td>BBH</td>
<td>01H</td>
<td>MSR_OFFCORE_RSP_1 (address 1A7H)</td>
</tr>
</tbody>
</table>

Table 20-14. Layout of Precise Store Information In PEBS Record

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Data Linear Address</td>
<td>98H</td>
<td>The linear address of the destination of the store.</td>
</tr>
<tr>
<td>Store Status</td>
<td>A0H</td>
<td>L1D Hit (bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. STLB Miss (bit 4): The store missed the STLB if set, otherwise the store hit the STLB Locked Access (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.</td>
</tr>
<tr>
<td>Reserved</td>
<td>A8H</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
## Table 20-16. MSR_OFFCORE_RSP_x Request_Type Field Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMND_DATA_RD</td>
<td>0</td>
<td>Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.</td>
</tr>
<tr>
<td>DMND_RFO</td>
<td>1</td>
<td>Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.</td>
</tr>
<tr>
<td>DMND_IFETCH</td>
<td>2</td>
<td>Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.</td>
</tr>
<tr>
<td>WB</td>
<td>3</td>
<td>Counts the number of writeback (modified to exclusive) transactions.</td>
</tr>
<tr>
<td>PF_DATA_RD</td>
<td>4</td>
<td>Counts the number of data cacheline reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_RFO</td>
<td>5</td>
<td>Counts the number of RFO requests generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_IFETCH</td>
<td>6</td>
<td>Counts the number of code reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_LLCDATA_RD</td>
<td>7</td>
<td>L2 prefetcher to L3 for loads.</td>
</tr>
<tr>
<td>PF_LLCRFO</td>
<td>8</td>
<td>RFO requests generated by L2 prefetcher</td>
</tr>
<tr>
<td>PF_LLCIFETCH</td>
<td>9</td>
<td>L2 prefetcher to L3 for instruction fetches.</td>
</tr>
<tr>
<td>BUS_LOCKS</td>
<td>10</td>
<td>Bus lock and split lock requests</td>
</tr>
<tr>
<td>STRM_ST</td>
<td>11</td>
<td>Streaming store requests</td>
</tr>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Any other request that crosses IDI, including I/O.</td>
</tr>
</tbody>
</table>

### Figure 20-30. Request_Type Fields for MSR_OFFCORE_RSP_x
To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSP_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

\[ \text{ANY} \mid (\text{‘OR’ of Supplier Info Bits}) \& (\text{‘OR’ of Snoop Info Bits}) \]

If "ANY" bit is set, the supplier and snoop info bits are ignored.
PERFORMANCE MONITORING

20.3.4.6 Uncore Performance Monitoring Facilities in the Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, and Intel® Core™ i3-2xxx Processor Series

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 20.3.1.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 20-32.

Table 20-18. MSR_OFFCORE_RSP_x Snoop Info Field Definition

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoop Info</td>
<td>SNP_NONE</td>
<td>31</td>
<td>No details on snoop-related information.</td>
</tr>
<tr>
<td></td>
<td>SNP_NOT_NEEDED</td>
<td>32</td>
<td>No snoop was needed to satisfy the request.</td>
</tr>
<tr>
<td>SNP_MISS</td>
<td>SNP_MISS</td>
<td>33</td>
<td>A snoop was needed and it missed all snooped caches:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- For LLC Hit, ResiHitl was returned by all cores</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.</td>
</tr>
<tr>
<td>SNP_NO_FWD</td>
<td>SNP_NO_FWD</td>
<td>34</td>
<td>A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Hit w/ Invalidation (LLC Hit, RFO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>In the LLC Miss case, data is returned from DRAM.</td>
</tr>
<tr>
<td>SNP_FWD</td>
<td>SNP_FWD</td>
<td>35</td>
<td>A snoop was needed and data was forwarded from a remote socket. This includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).</td>
</tr>
<tr>
<td>HITM</td>
<td>HITM</td>
<td>36</td>
<td>A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop HitM w/ WB (LLC miss, IFetch/Data_RD)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop MtoS (LLC Hit, IFetch/Data_RD).</td>
</tr>
<tr>
<td>NON_DRAM</td>
<td>NON_DRAM</td>
<td>37</td>
<td>Target was non-DRAM system address. This includes MMIO transactions.</td>
</tr>
</tbody>
</table>

Figure 20-32. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit
The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- **Event>Select (bits 7:0) and UMASK (bits 15:8):** Specifies the microarchitectural condition to count in a local uncore PMU counter, see the event list at: https://perfmon-events.intel.com/
- **E (bit 18):** Enables edge detection filtering, if 1.
- **OVF_EN (bit 20):** Enables the overflow indicator from the uncore counter forwarded to MSR_UNC_PERF_-GLOBAL_CTRL, if 1.
- **EN (bit 22):** Enables the local counter associated with this event select MSR.
- **INV (bit 23):** Event count increments with non-negative value if 0, with negated value if 1.
- **CMASK (bits 28:24):** Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 20-33 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR_UNC_PERF_-GLOBAL_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32_DEBUGCTL (at address 1D9H) to enable this capability.

- **PMI_SEL_Core#:** Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is ‘1’, a wake request is sent to the respective processor core prior to sending the PMI.
- **EN:** Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- **WakePMI:** Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- **FREEZE:** Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 20-19 summarizes the number MSRs for uncore PMU for each box.

### Table 20-19. Uncore PMU MSR Summary

<table>
<thead>
<tr>
<th>Box</th>
<th># of Boxes</th>
<th>Counters per Box</th>
<th>Counter Width</th>
<th>General Purpose</th>
<th>Global Enable</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Box</td>
<td>SKU specific</td>
<td>2</td>
<td>44</td>
<td>Yes</td>
<td>Per-box</td>
<td>Up to 4, see Table 2-21 MSR_UNC_CBO_CONFIG</td>
</tr>
<tr>
<td>ARB</td>
<td>1</td>
<td>2</td>
<td>44</td>
<td>Yes</td>
<td>Uncore</td>
<td></td>
</tr>
</tbody>
</table>
PERFORMANCE MONITORING

20.3.4.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.
- Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic cannot associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

20.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Sandy Bridge-E microarchitecture. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 20.3.4.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 20.6.3 through Section 20.3.4.5. However, the MSR_OFFCORE_RSP_0/MSR_OFFCORE_RSP_1 Response Supplier Info field shown in Table 20-17 applies to Intel Core Processors with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2AH; Intel Xeon processor with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2DH supports an additional field for remote DRAM controller shown in Table 20-20. Additionally, there are some small differences in the non-architectural performance monitoring events (see event list available at: https://perfmon-events.intel.com/).

Table 20-19. Uncore PMU MSR Summary (Contd.)

<table>
<thead>
<tr>
<th>Box</th>
<th># of Boxes</th>
<th>Counters per Box</th>
<th>Counter Width</th>
<th>General Purpose</th>
<th>Global Enable</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Counter</td>
<td>N.A.</td>
<td>N.A.</td>
<td>48</td>
<td>No</td>
<td>Uncore</td>
<td></td>
</tr>
</tbody>
</table>

Table 20-20. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>NO_SUPP</td>
<td>17</td>
<td>No Supplier Information available.</td>
</tr>
<tr>
<td></td>
<td>LLC_HITM</td>
<td>18</td>
<td>M-state initial lookup stat in L3.</td>
</tr>
<tr>
<td></td>
<td>LLC_HITE</td>
<td>19</td>
<td>E-state</td>
</tr>
<tr>
<td></td>
<td>LLC_HITS</td>
<td>20</td>
<td>S-state</td>
</tr>
<tr>
<td></td>
<td>LLC_HITF</td>
<td>21</td>
<td>F-state</td>
</tr>
<tr>
<td></td>
<td>LOCAL</td>
<td>22</td>
<td>Local DRAM Controller.</td>
</tr>
<tr>
<td>Remote</td>
<td>30:23</td>
<td></td>
<td>Remote DRAM Controller (either all Os or all 1s).</td>
</tr>
</tbody>
</table>
20.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 20-21 summarizes the uncore PMU facilities providing MSR interfaces.

<table>
<thead>
<tr>
<th>Box</th>
<th># of Boxes</th>
<th>Counters per Box</th>
<th>Counter Width</th>
<th>General Purpose</th>
<th>Global Enable</th>
<th>Sub-control MSRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Box</td>
<td>8</td>
<td>4</td>
<td>44</td>
<td>Yes</td>
<td>per-box</td>
<td>None</td>
</tr>
<tr>
<td>PCU</td>
<td>1</td>
<td>4</td>
<td>48</td>
<td>Yes</td>
<td>per-box</td>
<td>Match/Mask</td>
</tr>
<tr>
<td>U-Box</td>
<td>1</td>
<td>2</td>
<td>44</td>
<td>Yes</td>
<td>uncore</td>
<td>None</td>
</tr>
</tbody>
</table>

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in "Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-24.

20.3.5 3rd Generation Intel® Core™ Processor Performance Monitoring Facility

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 20.6.3 through Section 20.3.4.5. The non-architectural performance monitoring events supported by the processor core can be found at: https://perfmon-events.intel.com/.

20.3.5.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in the "Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-28.

20.3.6 4th Generation Intel® Core™ Processor Performance Monitoring Facility

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 20.2.3.

The core PMU’s capability is similar to those described in Section 20.6.3 through Section 20.3.4.5, with some differences and enhancements summarized in Table 20-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 20.3.6.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.
20.3.6.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Sandy Bridge microarchitecture, with several enhanced features. The key components and differences of PEBS facility relative to Sandy Bridge microarchitecture is summarized in Table 20-23.

### Table 20-22. Core PMU Comparison

<table>
<thead>
<tr>
<th>Box</th>
<th>Haswell Microarchitecture</th>
<th>Sandy Bridge Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Fixed counters per thread</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of general-purpose counters per core</td>
<td>8</td>
<td>8</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of programmable counters per thread</td>
<td>4 or (8 if a core not shared by two threads)</td>
<td>4 or (8 if a core not shared by two threads)</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>PMI Overhead Mitigation</td>
<td>• Freeze_Perfmon_on_PMI with legacy semantics. • Freeze_LBR_on_PMI with legacy semantics for branch profiling. • Freeze_while_SMM.</td>
<td>• Freeze_Perfmon_on_PMI with legacy semantics. • Freeze_LBR_on_PMI with legacy semantics for branch profiling. • Freeze_while_SMM.</td>
<td>See Section 18.4.7.</td>
</tr>
<tr>
<td>Processor Event Based Sampling (PEBS) Events</td>
<td>See Table 20-12 and Section 20.3.6.5.1.</td>
<td>See Table 20-12.</td>
<td>IA32_PMC4-IA32_PMC7 do not support PEBS.</td>
</tr>
<tr>
<td>PEBS-Load Latency</td>
<td>See Section 20.3.4.4.2.</td>
<td>See Section 20.3.4.4.2.</td>
<td></td>
</tr>
<tr>
<td>PEBS-Precise Store</td>
<td>No, replaced by Data Address profiling.</td>
<td>Section 20.3.4.4.3</td>
<td></td>
</tr>
<tr>
<td>PEBS-PDIR</td>
<td>Yes (using precise INST_RETIRED.ALL)</td>
<td>Yes (using precise INST_RETIRED.ALL)</td>
<td></td>
</tr>
<tr>
<td>PEBS-EventingIP</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Data Address Profiling</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LBR Profiling</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Call Stack Profiling</td>
<td>Yes, see Section 18.11.</td>
<td>No</td>
<td>Use LBR facility.</td>
</tr>
<tr>
<td>Off-core Response Event</td>
<td>MSR 1A6H and 1A7H; extended request and response types.</td>
<td>MSR 1A6H and 1A7H; extended request and response types.</td>
<td></td>
</tr>
<tr>
<td>Intel TSX support for Perfmon</td>
<td>See Section 20.3.6.5.</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

### Table 20-23. PEBS Facility Comparison

<table>
<thead>
<tr>
<th>Box</th>
<th>Haswell Microarchitecture</th>
<th>Sandy Bridge Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid IA32_PMCx</td>
<td>PMCO-PMC3</td>
<td>PMCO-PMC3</td>
<td>No PEBS on PMC4-PMC7</td>
</tr>
<tr>
<td>PEBS Buffer Programming</td>
<td>Section 20.3.1.1.1</td>
<td>Section 20.3.1.1.1</td>
<td>Unchanged</td>
</tr>
<tr>
<td>IA32_PEB_ENABLE Layout</td>
<td>Figure 20-15</td>
<td>Figure 20-29</td>
<td></td>
</tr>
<tr>
<td>PEBS record layout</td>
<td>Table 20-24; enhanced fields at offsets 98H, A0H, A8H, B0H.</td>
<td>Table 20-3; enhanced fields at offsets 98H, A0H, A8H.</td>
<td></td>
</tr>
</tbody>
</table>
Only IA32_PMC0 through IA32_PMC3 support PEBS.

**NOTE**

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

### 20.3.6.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 20-24. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

#### Table 20-24. PEBS Record Format for 4th Generation Intel Core Processor Family

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte Offset</th>
<th>Field</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/EFLAGS</td>
<td>00H</td>
<td>R10</td>
<td>60H</td>
</tr>
<tr>
<td>R/EIP</td>
<td>08H</td>
<td>R11</td>
<td>68H</td>
</tr>
<tr>
<td>R/EAX</td>
<td>10H</td>
<td>R12</td>
<td>70H</td>
</tr>
<tr>
<td>R/EBX</td>
<td>18H</td>
<td>R13</td>
<td>78H</td>
</tr>
<tr>
<td>R/ECX</td>
<td>20H</td>
<td>R14</td>
<td>80H</td>
</tr>
<tr>
<td>R/EDX</td>
<td>28H</td>
<td>R15</td>
<td>88H</td>
</tr>
<tr>
<td>R/ESI</td>
<td>30H</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td>90H</td>
</tr>
<tr>
<td>R/EDI</td>
<td>38H</td>
<td>Data Linear Address</td>
<td>98H</td>
</tr>
<tr>
<td>R/EBP</td>
<td>40H</td>
<td>Data Source Encoding</td>
<td>A0H</td>
</tr>
<tr>
<td>R/ESP</td>
<td>48H</td>
<td>Latency value (core cycles)</td>
<td>A8H</td>
</tr>
<tr>
<td>R</td>
<td>50H</td>
<td>EventingIP</td>
<td>B0H</td>
</tr>
<tr>
<td>R9</td>
<td>58H</td>
<td>TX Abort Information (Section 20.3.6.5.1)</td>
<td>B8H</td>
</tr>
</tbody>
</table>
The layout of PEBS records are almost identical to those shown in Table 20-3. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 20.3.4.4.2), PDIR (Section 20.3.4.4.4), and the equivalent capability of precise store in prior generation (see Section 20.3.6.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

### 20.3.6.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM_UOPS_RETIRED.STLB_MISS_LOADS</td>
<td>MEM_UOPS_RETIRED.STLB_MISS_STORES</td>
</tr>
<tr>
<td>MEM_UOPS_RETIRED.LOCK_LOADS</td>
<td>MEM_UOPS_RETIRED.SPLIT_STORES</td>
</tr>
<tr>
<td>MEM_UOPS_RETIRED.SPLIT_LOADS</td>
<td>MEM_UOPS_RETIRED.ALL_STORES</td>
</tr>
<tr>
<td>MEM_UOPS_RETIRED.ALL LOADS</td>
<td>MEM_LOAD_UOPSALLED_MISS_RETIRED.LOCAL_DRAM</td>
</tr>
<tr>
<td>MEM_LOAD_UOPS_RETIRED.L1_HIT</td>
<td>MEM_LOAD_UOPS_RETIRED.L2_HIT</td>
</tr>
<tr>
<td>MEM_LOAD_UOPS_RETIRED.L3_HIT</td>
<td>MEM_LOAD_UOPS_RETIRED.L1_MISS</td>
</tr>
<tr>
<td>MEM_LOAD_UOPS_RETIRED.L2_MISS</td>
<td>MEM_LOAD_UOPS_RETIRED.L3_MISS</td>
</tr>
<tr>
<td>MEM_LOAD_UOPS_RETIRED.HIT_LFB</td>
<td>MEM_LOAD_UOPS_RETIRED.L3_HIT_RETIRED.XSNP_MISS</td>
</tr>
<tr>
<td>MEM_LOAD_UOPS_RETIRED.XSNP_HIT</td>
<td>MEM_LOAD_UOPS_RETIRED.L3_HIT_RETIRED.XSNP_HITM</td>
</tr>
<tr>
<td>UOPS_RETIRED.ALL (if load or store is tagged)</td>
<td>MEM_LOAD_UOPS_RETIRED.L3_HIT_RETIRED.XSNP_NONE</td>
</tr>
</tbody>
</table>

DataLA can use any one of the IA32_PMC0-IA32_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program an event listed in Table 20-25 using any one of IA32_PERFEVTSEL0-IA32_PERFEVTSEL3.
- Set the corresponding IA32_PEB_ENABLE.PEBS_EN_CTRx bit. This enables the corresponding IA32_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H, and A8H, as shown in Table 20-26.
20.3.6.3.1 EventingIP Record

The PEBS record layout for processors based on Haswell microarchitecture adds a new field at offset 0B0H. This is the EventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

20.3.6.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 20.3.4.5. The event codes are listed in Table 20-15. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 20-27.
- Supplier information (bits 30:16): see Table 20-28.
- Snoop response information (bits 37:31): see Table 20-18.

Table 20-26. Layout of Data Linear Address Information In PEBS Record

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Linear Address</td>
<td>98H</td>
<td>The linear address of the load or the destination of the store.</td>
</tr>
<tr>
<td>Store Status</td>
<td>A0H</td>
<td>• DCU Hit (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES • Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 20-25.</td>
</tr>
<tr>
<td>Reserved</td>
<td>A8H</td>
<td>Always zero.</td>
</tr>
</tbody>
</table>

Table 20-27. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMND_DATA_RD</td>
<td>0</td>
<td>Counts the number of demand data reads and page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.</td>
</tr>
<tr>
<td>DMND_RFO</td>
<td>1</td>
<td>Counts demand read (RFO) and software prefetches (PREFETCHW) for exclusive ownership in anticipation of a write.</td>
</tr>
<tr>
<td>DMND_IFETCH</td>
<td>2</td>
<td>Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.</td>
</tr>
<tr>
<td>COREWB</td>
<td>3</td>
<td>Counts the number of modified cachelines written back.</td>
</tr>
<tr>
<td>PF_DATA_RD</td>
<td>4</td>
<td>Counts the number of data cacheline reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_RFO</td>
<td>5</td>
<td>Counts the number of RFO requests generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_IFETCH</td>
<td>6</td>
<td>Counts the number of code reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_L3_DATA_RD</td>
<td>7</td>
<td>Counts the number of data cacheline reads generated by L3 prefetchers.</td>
</tr>
<tr>
<td>PF_L3_RFO</td>
<td>8</td>
<td>Counts the number of RFO requests generated by L3 prefetchers.</td>
</tr>
<tr>
<td>PF_L3_CODE_RD</td>
<td>9</td>
<td>Counts the number of code reads generated by L3 prefetchers.</td>
</tr>
<tr>
<td>SPLIT_LOCK_UC_LOCK</td>
<td>10</td>
<td>Counts the number of lock requests that split across two cachelines or are to UC memory.</td>
</tr>
<tr>
<td>STRM_ST</td>
<td>11</td>
<td>Counts the number of streaming store requests electronically.</td>
</tr>
<tr>
<td>Reserved</td>
<td>14:12</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
PERFORMANCE MONITORING

Table 20-27. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Any other request that crosses IDI, including I/O.</td>
</tr>
</tbody>
</table>

The supplier information field listed in Table 20-28. The fields vary across products (according to CPUID signatures) and is noted in the description.

Table 20-28. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_3CH, 06_46H)

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>NO_SUPP</td>
<td>17</td>
<td>No Supplier Information available.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HITM</td>
<td>18</td>
<td>M-state initial lookup stat in L3.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HITE</td>
<td>19</td>
<td>E-state</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HITS</td>
<td>20</td>
<td>S-state</td>
</tr>
<tr>
<td>Reserved</td>
<td>21</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>LOCAL</td>
<td>22</td>
<td>Local DRAM Controller.</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>30:23</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Table 20-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_45H)

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>NO_SUPP</td>
<td>17</td>
<td>No Supplier Information available.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HITM</td>
<td>18</td>
<td>M-state initial lookup stat in L3.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HITE</td>
<td>19</td>
<td>E-state</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HITS</td>
<td>20</td>
<td>S-state</td>
</tr>
<tr>
<td>Reserved</td>
<td>21</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>L4_HIT_LOCAL_L4</td>
<td></td>
<td>22</td>
<td>L4 Cache</td>
</tr>
<tr>
<td>L4_HITREMOTE_HOP0_L4</td>
<td></td>
<td>23</td>
<td>L4 Cache</td>
</tr>
<tr>
<td>L4_HITREMOTE_HOP1_L4</td>
<td></td>
<td>24</td>
<td>L4 Cache</td>
</tr>
<tr>
<td>L4_HITREMOTE_HOP2P_L4</td>
<td></td>
<td>25</td>
<td>L4 Cache</td>
</tr>
<tr>
<td>Reserved</td>
<td>30:26</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

20.3.6.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 20-28 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CPUID signature 06_3FH).
20.3.6.5 Performance Monitoring and Intel® TSX

Chapter 16 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the details of Intel® Transactional Synchronization Extensions (Intel® TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances its IA32_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32_PERFEVTSELx is shown in Figure 20-34. The two additional bit fields are:

- **IN_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32_PERFEVTSEL2.

When the IA32_PERFEVTSELx MSR is programmed with both IN_TX=0 and IN_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN_TXCP bit. See Table 2-29.

---

### Table 20-30. MSR_OFFCORE_RSP_x Supplier Info Field Definition

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>NO_SUPP</td>
<td>17</td>
<td>No Supplier Information available.</td>
</tr>
<tr>
<td></td>
<td>L3_HITM</td>
<td>18</td>
<td>M-state initial lookup stat in L3.</td>
</tr>
<tr>
<td></td>
<td>L3_HITE</td>
<td>19</td>
<td>E-state</td>
</tr>
<tr>
<td></td>
<td>L3_HITS</td>
<td>20</td>
<td>S-state</td>
</tr>
<tr>
<td></td>
<td>L3_HITF</td>
<td>21</td>
<td>F-state</td>
</tr>
<tr>
<td></td>
<td>LOCAL</td>
<td>22</td>
<td>Local DRAM Controller.</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>26:23</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>L3_MISS_REMOTE_HOP0</td>
<td>27</td>
<td>Hop 0 Remote supplier.</td>
</tr>
<tr>
<td></td>
<td>L3_MISS_REMOTE_HOP1</td>
<td>28</td>
<td>Hop 1 Remote supplier.</td>
</tr>
<tr>
<td></td>
<td>L3_MISS_REMOTE_HOP2P</td>
<td>29</td>
<td>Hop 2 or more Remote supplier.</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>30</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
A common usage of setting IN_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32_PERFEVTSELx.IN_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32_PERFEVTSELx but occurring outside a transactional region are discarded.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they can be found at: https://perfmon-events.intel.com/.

### 20.3.6.5.1 Intel® TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events also support using the PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORTED (encoding C8H mask 04H),
- RTM_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE_RETIRED.ABORTED,RTM_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pended at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 20-31.
### 20.3.6.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 20.3.4.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 20-32.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 20-33 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 20-19 summarizes the number MSRs for uncore PMU for each box.

![Table 20-31. TX Abort Information Field Definition](image)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles_Last_TX</td>
<td>31:0</td>
<td>The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.</td>
</tr>
<tr>
<td>HLE_Abort</td>
<td>32</td>
<td>If set, the abort information corresponds to an aborted HLE execution</td>
</tr>
<tr>
<td>RTM_Abort</td>
<td>33</td>
<td>If set, the abort information corresponds to an aborted RTM execution</td>
</tr>
<tr>
<td>Instruction_Abort</td>
<td>34</td>
<td>If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.</td>
</tr>
<tr>
<td>Non_Instruction_Abort</td>
<td>35</td>
<td>If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.</td>
</tr>
<tr>
<td>Retry</td>
<td>36</td>
<td>If set, retrying the transactional execution may have succeeded.</td>
</tr>
<tr>
<td>Data_Conflict</td>
<td>37</td>
<td>If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.</td>
</tr>
<tr>
<td>Capacity Writes</td>
<td>38</td>
<td>If set, the transactional region aborted due to exceeding resources for transactional writes.</td>
</tr>
<tr>
<td>Capacity Reads</td>
<td>39</td>
<td>If set, the transactional region aborted due to exceeding resources for transactional reads.</td>
</tr>
<tr>
<td>In_Suspend</td>
<td>40</td>
<td>Transaction was aborted while in a suspend region. This is an Intel Xeon processor only feature, available beginning with 4th generation Intel Xeon Scalable Processor Family; otherwise reserved.</td>
</tr>
<tr>
<td>Reserved</td>
<td>63:41</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### 20.3.6.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in “Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual”. The MSR-based uncore PMU interfaces are listed in Table 2-33.
20.3.7 5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility

The 5th Generation Intel® Core™ processor and the Intel® Core™ M processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 20.2.3. The core PMU has the same capability as those described in Section 20.3.6. IA32_PERF_GLOBAL_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.

Details of Intel Processor Trace is described in Chapter 33, “Intel® Processor Trace.” The IA32_PERF_GLOBAL_OVF_CTRL MSR provides a corresponding reset control bit.

---

**Figure 20-35. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture**

**Figure 20-36. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture**
The specifics of non-architectural performance events can be found at: https://perfmon-events.intel.com/.

20.3.8 6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture. For these microarchitectures, the core PMU supports architectural performance monitoring capability with version ID 4 (see Section 20.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 20.2.4.

The core PMU’s capability is similar to those described in Section 20.6.3 through Section 20.3.4.5, with some differences and enhancements summarized in Table 20-33. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 20.3.6.5. For details of Intel TSX, see Chapter 16, “Programming with Intel® Transactional Synchronization Extensions,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 40, “Enclave Code Debug and Profiling.”

Table 20-33. Core PMU Comparison

<table>
<thead>
<tr>
<th>Box</th>
<th>Skylake, Kaby Lake and Coffee Lake Microarchitectures</th>
<th>Haswell and Broadwell Microarchitectures</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Fixed counters per thread</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of general-purpose counters per core</td>
<td>8</td>
<td>8</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of programmable counters per thread</td>
<td>4 or (8 if a core not shared by two threads)</td>
<td>4 or (8 if a core not shared by two threads)</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>Architectural Perfmon version</td>
<td>4</td>
<td>3</td>
<td>See Section 20.2.4</td>
</tr>
<tr>
<td>PMI Overhead Mitigation</td>
<td>- Freeze_Perfmon_on_PMI with streamlined semantics. - Freeze_LBR_on_PMI with streamlined semantics. - Freeze_while_SMM.</td>
<td>- Freeze_Perfmon_on_PMI with legacy semantics. - Freeze_LBR_on_PMI with legacy semantics for branch profiling. - Freeze_while_SMM.</td>
<td>See Section 18.4.7. Legacy semantics not supported with version 4 or higher.</td>
</tr>
<tr>
<td>Counter and Buffer Overflow Status Management</td>
<td>- Query via IA32_PERF_GLOBAL_STATUS - Reset via IA32_PERF_GLOBAL_STATUS_RESET - Set via IA32_PERF_GLOBAL_STATUS_SET</td>
<td>- Query via IA32_PERF_GLOBAL_STATUS - Reset via IA32_PERF_GLOBAL_OVF_CTRL</td>
<td>See Section 20.2.4.</td>
</tr>
</tbody>
</table>
PERFORMANCE MONITORING

20.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 20-34.

<table>
<thead>
<tr>
<th>Box</th>
<th>IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference</th>
<th>Haswell and Broadwell Microarchitectures</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Individual counter overflow • PEBS buffer overflow • ToPA buffer overflow • CTR_Frz, LBR_Frz, ASCI</td>
<td>• Individual counter overflow • PEBS buffer overflow • ToPA buffer overflow (applicable to Broadwell microarchitecture)</td>
<td>See Section 20.2.4.</td>
</tr>
</tbody>
</table>

| Enable control in IA32_PERF_GLOBAL_STATUS | CTR_Frz • LBR_Frz | NA | See Section 20.2.4.1. |

| Perfmon Counter In-Use Indicator | Query IA32_PERF_GLOBAL_INUSE | NA | See Section 20.2.4.3. |

| Precise Events | See Table 20-36. | See Table 20-12. | IA32_PMC4-PMC7 do not support PEBS. |

| PEBS for front end events | See Section 20.3.8.2. | No |

| LBR Record Format Encoding | 000101b | 000100b | Section 18.4.8.1 |

| LBR Size | 32 entries | 16 entries |

| LBR Entry | From_IP/To_IP/LBR_Info triplet | From_IP/To_IP pair | Section 18.12 |

| LBR Timing | Yes | No | Section 18.12.1 |

| Call Stack Profiling | Yes, see Section 18.11 | Yes, see Section 18.11 | Use LBR facility. |

| Off-core Response Event | MSR 1A6H and 1A7H; Extended request and response types. | MSR 1A6H and 1A7H; Extended request and response types. |

| Intel TSX support for Perfmon | See Section 20.3.6.5. | See Section 20.3.6.5. |

20.3.8.2 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th generation, 7th generation and 8th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 20-34.

<table>
<thead>
<tr>
<th>Box</th>
<th>IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference</th>
<th>Haswell and Broadwell Microarchitectures</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Individual counter overflow • PEBS buffer overflow • ToPA buffer overflow • CTR_Frz, LBR_Frz, ASCI</td>
<td>• Individual counter overflow • PEBS buffer overflow • ToPA buffer overflow (applicable to Broadwell microarchitecture)</td>
<td>See Section 20.2.4.</td>
</tr>
</tbody>
</table>

| Enable control in IA32_PERF_GLOBAL_STATUS | CTR_Frz • LBR_Frz | NA | See Section 20.2.4.1. |

| Perfmon Counter In-Use Indicator | Query IA32_PERF_GLOBAL_INUSE | NA | See Section 20.2.4.3. |

| Precise Events | See Table 20-36. | See Table 20-12. | IA32_PMC4-PMC7 do not support PEBS. |

| PEBS for front end events | See Section 20.3.8.2. | No |

| LBR Record Format Encoding | 000101b | 000100b | Section 18.4.8.1 |

| LBR Size | 32 entries | 16 entries |

| LBR Entry | From_IP/To_IP/LBR_Info triplet | From_IP/To_IP pair | Section 18.12 |

| LBR Timing | Yes | No | Section 18.12.1 |

| Call Stack Profiling | Yes, see Section 18.11 | Yes, see Section 18.11 | Use LBR facility. |

| Off-core Response Event | MSR 1A6H and 1A7H; Extended request and response types. | MSR 1A6H and 1A7H; Extended request and response types. |

| Intel TSX support for Perfmon | See Section 20.3.6.5. | See Section 20.3.6.5. |

Table 20-34. PEBS Facility Comparison

<table>
<thead>
<tr>
<th>Box</th>
<th>Skylake, Kaby Lake and Coffee Lake Microarchitectures</th>
<th>Haswell and Broadwell Microarchitectures</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid IA32_PMCx</td>
<td>PMCO-PMC3</td>
<td>PMCO-PMC3</td>
<td>No PEBS on PMC4-PMC7.</td>
</tr>
</tbody>
</table>

| PEBS Buffer Programming | Section 20.3.1.1.1 | Section 20.3.1.1.1 | Unchanged |

| IA32_PES_ENABLE Layout | Figure 20-15 | Figure 20-15 |

| PEBS-EventingIP | Yes | Yes |

| PEBS record format encoding | 0011b | 0010b |

| PEBS record layout | Table 20-35; enhanced fields at offsets 98H- B8H; and TSC record field at COH. | Table 20-24; enhanced fields at offsets 98H, A0H, A8H, B0H. |

| Multi-counter PEBS resolution | PEBS record 90H resolves the eventing counter overflow. | PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS. |

| Precise Events | See Table 20-36. | See Table 20-12. | IA32_PMC4-IA32_PMC7 do not support PEBS. |
Only IA32_PMC0 through IA32_PMC3 support PEBS.

**NOTES**

Precise events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

### 20.3.8.1.1 PEBS Data Format

The PEBS record format for the 6th generation, 7th generation and 8th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The layout is shown in Table 20-35. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

**Table 20-35. PEBS Record Format for the 6th Generation, 7th Generation, and 8th Generation Intel Core Processor Families**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field</th>
<th>Byte Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>R/EFLAGS</td>
<td>68H</td>
<td>R11</td>
</tr>
<tr>
<td>08H</td>
<td>R/EIP</td>
<td>70H</td>
<td>R12</td>
</tr>
<tr>
<td>10H</td>
<td>R/EAX</td>
<td>78H</td>
<td>R13</td>
</tr>
<tr>
<td>18H</td>
<td>R/EBX</td>
<td>80H</td>
<td>R14</td>
</tr>
<tr>
<td>20H</td>
<td>R/ECX</td>
<td>88H</td>
<td>R15</td>
</tr>
<tr>
<td>28H</td>
<td>R/EDX</td>
<td>90H</td>
<td>Applicable Counter</td>
</tr>
<tr>
<td>30H</td>
<td>R/ESI</td>
<td>98H</td>
<td>Data Linear Address</td>
</tr>
<tr>
<td>38H</td>
<td>R/EDI</td>
<td>A0H</td>
<td>Data Source Encoding</td>
</tr>
<tr>
<td>40H</td>
<td>R/EBP</td>
<td>A8H</td>
<td>Latency value (core cycles)</td>
</tr>
<tr>
<td>48H</td>
<td>R/ESP</td>
<td>80H</td>
<td>EventingIP</td>
</tr>
<tr>
<td>50H</td>
<td>R8</td>
<td>88H</td>
<td>TX Abort Information (Section 20.3.6.5.1)</td>
</tr>
<tr>
<td>58H</td>
<td>R9</td>
<td>COH</td>
<td>TSC</td>
</tr>
<tr>
<td>60H</td>
<td>R10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The layout of PEBS records are largely identical to those shown in Table 20-24.
The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 20.3.4.4.2), PDIR (Section 20.3.4.4.4), and data address profiling (Section 20.3.6.3).

In the core PMU of the 6th generation, 7th generation and 8th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th generation and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32_PERF_GLOBAL_STATUS may be useful to resolve the situations when more than one of IA32_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

### 20.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake, Kaby Lake, and Coffee Lake microarchitectures is shown in Table 20-36.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Select</th>
<th>Sub-event</th>
<th>UMask</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST_RETIRED</td>
<td>C0H</td>
<td>PREC_DIST(^1)</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALL_CYCLES(^2)</td>
<td>01H</td>
</tr>
<tr>
<td>OTHER_ASSISTS</td>
<td>C1H</td>
<td>ANY</td>
<td>3FH</td>
</tr>
<tr>
<td>BR_INST_RETIRED</td>
<td>C4H</td>
<td>CONDITIONAL</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEAR_CALL</td>
<td>02H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALL_BRANCHES</td>
<td>04H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEAR_RETURN</td>
<td>08H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEAR_TAKEN</td>
<td>20H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR_BRANCHES</td>
<td>40H</td>
</tr>
<tr>
<td>BR_MISP_RETIRED</td>
<td>C5H</td>
<td>CONDITIONAL</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALL_BRANCHES</td>
<td>04H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NEAR_TAKEN</td>
<td>20H</td>
</tr>
<tr>
<td>FRONTEND_RETIRED</td>
<td>C6H</td>
<td>&lt;Programmable(^2)&gt;</td>
<td>01H</td>
</tr>
<tr>
<td>HLE_RETIRED</td>
<td>C8H</td>
<td>ABORTED</td>
<td>04H</td>
</tr>
<tr>
<td>RTM_RETIRED</td>
<td>C9H</td>
<td>ABORTED</td>
<td>04H</td>
</tr>
<tr>
<td>MEM_INST_RETIRED</td>
<td>D0H</td>
<td>LOCK_LOADS</td>
<td>21H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SPLIT_LOADS</td>
<td>41H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SPLIT_STORES</td>
<td>42H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALL_LOADS</td>
<td>81H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALL_STORES</td>
<td>82H</td>
</tr>
</tbody>
</table>
20.3.8.1.3 Data Address Profiling

The PEBS Data address profiling on the 6th generation, 7th generation and 8th generation Intel Core processors is largely unchanged from the prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H, and A8H, as shown in Table 20-26.

### Table 20-37. Layout of Data Linear Address Information In PEBS Record

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Linear Address</td>
<td>98H</td>
<td>The linear address of the load or the destination of the store.</td>
</tr>
</tbody>
</table>
| Store Status    | A0H    | • **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES.  
  • Other bits are zero. |
| Reserved         | A8H    | Always zero.                                     |

20.3.8.2 Frontend Retired Facility

The Skylake Core PMU has been extended to cover common microarchitectural conditions related to the front end pipeline in addition to providing a generic latency mechanism that can locate fetch bubbles without necessarily attributing them to a particular condition. The facility counts the events if the associated instruction reaches retirement (architecturally committed). Additionally, the user may opt to enable the PEBS facility to obtain precise information on the context of the event, e.g., EventingIP.

The supported frontend microarchitectural conditions require the following interfaces:

- The IA32_PERFEVTSELx MSR must select the FRONTEND_RETIRED event, EventSelect = C6H and UMask = 01H.
• This event employs a new MSR, MSR_PEBS_FRONTEND, to specify the supported frontend event details, see Table 20-38.

• If precise information is desired, program the PEBS_EN_PMCx field of IA32_PEBS_ENABLE MSR as required. Note the AnyThread field of IA32_PERFEVTSELX is ignored by the processor for the "FRONTEND_RETIRED" event. The sub-event encodings supported by MSR_PEBS_FRONTEND.EVTSEL is given in Table 20-38.

### Table 20-38. FrontEnd_Retired Sub-Event Encodings Supported by MSR_PEBS_FRONTEND.EVTSEL

<table>
<thead>
<tr>
<th>Sub-Event Name</th>
<th>EVTSEL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_DSB_MISS</td>
<td>1H</td>
<td>Retired Instructions which experienced any decode stream buffer (DSB) miss.</td>
</tr>
<tr>
<td>DSB_MISS</td>
<td>11H</td>
<td>Retired Instructions which experienced a DSB miss that caused a fetch starvation cycle.</td>
</tr>
<tr>
<td>L1I_MISS</td>
<td>12H</td>
<td>The fetch of retired Instructions which experienced Instruction L1 Cache true miss. Additional requests to the same cache line as an in-flight L1I cache miss will not be counted.</td>
</tr>
<tr>
<td>L2_MISS</td>
<td>13H</td>
<td>The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.</td>
</tr>
<tr>
<td>ITLB_MISS</td>
<td>14H</td>
<td>The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.</td>
</tr>
<tr>
<td>STL1_MISS</td>
<td>15H</td>
<td>The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.</td>
</tr>
<tr>
<td>IDQ_READ_BUBBLES</td>
<td>6H</td>
<td>An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles. The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR_PEBS_FRONTEND is given in Table 20-39.

### Table 20-39. MSR_PEBS_FRONTEND Layout

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVTSEL</td>
<td>7:0</td>
<td>Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 20-38.</td>
</tr>
<tr>
<td>IDQ_Bubble_Length</td>
<td>19:8</td>
<td>Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.</td>
</tr>
<tr>
<td>IDQ_Bubble_Width</td>
<td>22:20</td>
<td>Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.</td>
</tr>
<tr>
<td>Reserved</td>
<td>63:23</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The FRONTEND_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

• The event counts only retired (non-speculative) front-end events, i.e., events from just true program execution path are counted.

• The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.

• If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
• If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
• If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
• If a front-end (miss) event occurs outside instruction boundary (e.g., due to processor handling of architectural event), it may be reported for the next instruction to retire.

20.3.8.3 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 20.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:
• Transaction request type encoding (bits 15:0): see Table 20-40.
• Supplier information (bits 29:16): see Table 20-41.
• Snoop response information (bits 37:30): see Table 20-42.

Table 20-40. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake, Kaby Lake, and Coffee Lake Microarchitectures)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMND_DATA_RD</td>
<td>0</td>
<td>Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.</td>
</tr>
<tr>
<td>DMND_RFO</td>
<td>1</td>
<td>Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.</td>
</tr>
<tr>
<td>DMND_IFETCH</td>
<td>2</td>
<td>Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.</td>
</tr>
<tr>
<td>Reserved</td>
<td>14:3</td>
<td>Reserved</td>
</tr>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Counts miscellaneous requests, such as I/O and uncacheable accesses.</td>
</tr>
</tbody>
</table>

Table 20-41 lists the supplier information field that applies to 6th generation, 7th generation and 8th generation Intel Core processors. (6th generation Intel Core processor CPUID signatures: 06_4EH and 06_5EH; 7th generation and 8th generation Intel Core processor CPUID signatures: 06_8EH and 06_9EH).

Table 20-41. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9EH)

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>NO_SUPP</td>
<td>17</td>
<td>No Supplier Information available.</td>
</tr>
<tr>
<td></td>
<td>L3_HITM</td>
<td>18</td>
<td>M-state initial lookup stat in L3.</td>
</tr>
<tr>
<td></td>
<td>L3_HITE</td>
<td>19</td>
<td>E-state</td>
</tr>
<tr>
<td></td>
<td>L3_HITS</td>
<td>20</td>
<td>S-state</td>
</tr>
<tr>
<td>Reserved</td>
<td>21</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L4_HIT</td>
<td>22</td>
<td>L4 Cache (if L4 is present in the processor).</td>
</tr>
<tr>
<td>Reserved</td>
<td>25:23</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>26</td>
<td>Local Node</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>29:27</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>SPL_HIT</td>
<td>30</td>
<td>L4 cache super line hit (if L4 is present in the processor).</td>
<td></td>
</tr>
</tbody>
</table>
Table 20-42 lists the snoop information field that apply to processors with CPUID signatures 06_4EH, 06_5EH, 06_8EH, 06_9E, and 06_55H.

### Table 20-42. MSR_OFFCORE_RSP_x Snoop Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9E, 06_55H)

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoop Info</td>
<td>SPL_HIT</td>
<td>30</td>
<td>L4 cache super line hit (if L4 is present in the processor).</td>
</tr>
<tr>
<td></td>
<td>SNOOP_NONE</td>
<td>31</td>
<td>No details on snoop-related information.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_NOT_NEEDED</td>
<td>32</td>
<td>No snoop was needed to satisfy the request.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_MISS</td>
<td>33</td>
<td>A snoop was needed and it missed all snooped caches:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- For LLC Hit, ResHi Hit was returned by all cores.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- For LLC Miss, Rspl was returned by all sockets and data was returned from</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DRAM.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_HIT_NO_FWD</td>
<td>34</td>
<td>A snoop was needed and it hits in at least one snooped cache. Hit denotes a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cache-line was valid before snoop effect. This includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Hit w/ Invalidation (LLC Hit, RFO).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>In the LLC Miss case, data is returned from DRAM.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_HIT_WITH_FWD</td>
<td>35</td>
<td>A snoop was needed and data was forwarded from a remote socket. This</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).</td>
</tr>
<tr>
<td></td>
<td>SNOOP_HITM</td>
<td>36</td>
<td>A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cache-line was in modified state before effect as a results of snoop. This</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop HitM w/ WB (LLC miss, IFetch/Data_RD).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Snoop MtoS (LLC Hit, IFetch/Data_RD).</td>
</tr>
<tr>
<td></td>
<td>SNOOP_NON_DRAM</td>
<td>37</td>
<td>Target was non-DRAM system address. This includes MMIO transactions.</td>
</tr>
</tbody>
</table>

#### 20.3.8.3.1 Off-core Response Performance Monitoring for the Intel® Xeon® Scalable Processor Family

The following tables list the requestor and supplier information fields that apply to the Intel® Xeon® Scalable Processor Family.

- Transaction request type encoding (bits 15:0): see Table 20-43.
- Supplier information (bits 29:16): see Table 20-44.
- Supplier information (bits 29:16) with support for Intel® Optane™ DC Persistent Memory support: see Table 20-45.
- Snoop response information has not been changed and is the same as in (bits 37:30): see Table 20-42.
Table 20-43.  MSR_OFFCORE_RSP_x Request_Type Definition (Intel® Xeon® Scalable Processor Family)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMAND_DATA_RD</td>
<td>0</td>
<td>Counts the number of demand data reads and page table entry cacheline reads. Does not count hw or sw prefetches.</td>
</tr>
<tr>
<td>DEMAND_RFO</td>
<td>1</td>
<td>Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.</td>
</tr>
<tr>
<td>DEMAND_CODE_RD</td>
<td>2</td>
<td>Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.</td>
</tr>
<tr>
<td>Reserved</td>
<td>3</td>
<td>Reserved.</td>
</tr>
<tr>
<td>PF_L2_DATA_RD</td>
<td>4</td>
<td>Counts the number of prefetch data reads into L2.</td>
</tr>
<tr>
<td>PF_L2_RFO</td>
<td>5</td>
<td>Counts the number of RFO Requests generated by the MLC prefetches to L2.</td>
</tr>
<tr>
<td>Reserved</td>
<td>6</td>
<td>Reserved.</td>
</tr>
<tr>
<td>PF_L3_DATA_RD</td>
<td>7</td>
<td>Counts the number of MLC data read prefetches into L3.</td>
</tr>
<tr>
<td>PF_L3_RFO</td>
<td>8</td>
<td>Counts the number of RFO requests generated by MLC prefetches to L3.</td>
</tr>
<tr>
<td>Reserved</td>
<td>9</td>
<td>Reserved.</td>
</tr>
<tr>
<td>PF_L1D_AND_SW</td>
<td>10</td>
<td>Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.</td>
</tr>
<tr>
<td>Reserved</td>
<td>14:11</td>
<td>Reserved.</td>
</tr>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Counts miscellaneous requests, such as I/O and un-cacheable accesses.</td>
</tr>
</tbody>
</table>

Table 20-44 lists the supplier information field that applies to the Intel Xeon Scalable Processor Family (CPUID signature: 06_55H).

Table 20-44.  MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H)

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>SUPPLIER_NONE</td>
<td>17</td>
<td>No Supplier Information available.</td>
</tr>
<tr>
<td></td>
<td>L3_HIT_M</td>
<td>18</td>
<td>M-state initial lookup stat in L3.</td>
</tr>
<tr>
<td></td>
<td>L3_HIT_E</td>
<td>19</td>
<td>E-state</td>
</tr>
<tr>
<td></td>
<td>L3_HIT_S</td>
<td>20</td>
<td>S-state</td>
</tr>
<tr>
<td></td>
<td>L3_HIT_F</td>
<td>21</td>
<td>F-state</td>
</tr>
<tr>
<td>Reserved</td>
<td>25:22</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3_MISS_LOCAL_DRAM</td>
<td>26</td>
<td>L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.</td>
</tr>
<tr>
<td></td>
<td>L3_MISS_REMOTE_HOPO_DRAM</td>
<td>27</td>
<td>Hop 0 Remote supplier.</td>
</tr>
<tr>
<td></td>
<td>L3_MISS_REMOTE_HOP1_DRAM</td>
<td>28</td>
<td>Hop 1 Remote supplier.</td>
</tr>
<tr>
<td></td>
<td>L3_MISS_REMOTE_HOP2P_DRAM</td>
<td>29</td>
<td>Hop 2 or more Remote supplier.</td>
</tr>
<tr>
<td>Reserved</td>
<td>30</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Table 20-45 lists the supplier information field that applies to the Intel Xeon Scalable Processor Family (CPUID signature: 06_55H, Steppings 0x5H - 0xFH).
20.3.8.4 Uncore Performance Monitoring Facilities on Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Cannon Lake microarchitecture introduces LLC support of up to six processor cores. To support six processor cores and eight LLC slices, existing MSRs have been rearranged and new CBo MSRs have been added. Uncore performance monitoring software drivers from prior generations of Intel Core processors will need to update the MSR addresses. The new MSRs and updated MSR addresses have been added to the Uncore PMU listing in Section 2.17.2, “MSRs Specific to 8th Generation Intel® Core™ i3 Processors,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

20.3.9 10th Generation Intel® Core™ Processor Performance Monitoring Facility

Some 10th generation Intel® Core™ processors and some 3rd generation Intel® Xeon® Scalable Processor Family are based on Ice Lake microarchitecture. Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture. For these processors, the core PMU supports architectural performance monitoring capability with version Id 5 (see Section 20.2.5) and a host of non-architectural monitoring capabilities.

The core PMU’s capability is similar to those described in Section 20.3.1 through Section 20.3.8, with some differences and enhancements summarized in Table 20-46.
**20.3.9.1 Processor Event Based Sampling (PEBS) Facility**

The PEBS facility in the 10th generation Intel Core processors provides a number of enhancements relative to PEBS in processors based on the Skylake, Kaby Lake, and Coffee Lake microarchitectures. Enhancement of the PEBS facility with Extended PEBS and Adaptive PEBS features is described in detail in Section 20.9.

The 3rd generation Intel Xeon Scalable Family of processors based on the Ice Lake microarchitecture introduce EPT-friendly PEBS. This allows EPT violations and other VM Exits to be taken on PEBS accesses to the DS Area. See Section 20.9.5 for details.

**20.3.9.2 Off-core Response Performance Monitoring**

The core PMU facility to collect off-core response events are similar to those described in Section 20.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:
- Transaction request type encoding (bits 15:0): see Table 18-[N1].
- Response type encoding (bits 16-37) of
  - Supplier information: see Table [18-N2].
  - Snoop response information: see Table [18-N3].
- All transactions are tracked at cacheline granularity except some in request type OTHER.

### Table 20-46. Core PMU Summary of the Ice Lake Microarchitecture

<table>
<thead>
<tr>
<th>Box</th>
<th>Ice Lake Microarchitecture</th>
<th>Skylake, Kaby Lake and Coffee Lake Microarchitectures</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Perfmon version</td>
<td>5</td>
<td>4</td>
<td>See Section 20.2.5.</td>
</tr>
<tr>
<td>Number of programmable counters per thread</td>
<td>8</td>
<td>4</td>
<td>Use CPUID to determine number of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>PEBS: Basic functionality</td>
<td>Yes</td>
<td>Yes</td>
<td>See Section 20.3.9.1.</td>
</tr>
<tr>
<td>PEBS record format encoding</td>
<td>0100b</td>
<td>0011b</td>
<td>See Section 20.6.2.4.2.</td>
</tr>
<tr>
<td>Extended PEBS</td>
<td>PEBS is extended to all Fixed and General Purpose counters and to all performance monitoring events.</td>
<td>No</td>
<td>See Section 20.9.1.</td>
</tr>
<tr>
<td>Adaptive PEBS</td>
<td>Yes</td>
<td>No</td>
<td>See Section 20.9.2.</td>
</tr>
<tr>
<td>Performance Metrics</td>
<td>Yes (4)</td>
<td>No</td>
<td>See Section 20.3.9.3.</td>
</tr>
<tr>
<td>PEBS-PDIR</td>
<td>IA32 FIXED0 only (Corresponding counter control MSRs must be enabled.)</td>
<td>IA32_PMC1 only.</td>
<td></td>
</tr>
</tbody>
</table>

### Table 20-47. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMAND_DATA_RD</td>
<td>0</td>
<td>Counts demand data and page table entry reads.</td>
</tr>
<tr>
<td>DEMAND_RFO</td>
<td>1</td>
<td>Counts demand read (RFO) and software prefetches (PREFETCHw) for exclusive ownership in anticipation of a write.</td>
</tr>
<tr>
<td>DEMAND_CODE_RD</td>
<td>2</td>
<td>Counts demand instruction fetches and instruction prefetches targeting the L1 instruction cache.</td>
</tr>
<tr>
<td>Reserved</td>
<td>3</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
PERFORMANCE MONITORING

Table 20-47. MSR_OFFCORE_RSP_x Request_Type Definition
(Processors Based on Ice Lake Microarchitecture)

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>Any</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Combined Response Info</td>
<td>DRAM</td>
<td>26, 31, 32</td>
<td>Requests that are satisfied by DRAM.</td>
</tr>
<tr>
<td></td>
<td>NON_DRAM</td>
<td>26, 37</td>
<td>Requests that are satisfied by a NON_DRAM system component. This includes MMIO transactions.</td>
</tr>
<tr>
<td></td>
<td>L3_MISS</td>
<td>22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37</td>
<td>Requests that were not supplied by the L3 Cache. The event includes some currently reserved bits in anticipation of future memory designs.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>L3_HIT</td>
<td>18, 19, 20</td>
<td>Requests that hit in L3 cache. Depending on the snoop response the L3 cache may have retrieved the cacheline from another core's cache.</td>
</tr>
<tr>
<td>Reserved</td>
<td>17, 21:25, 27:29</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. All bits need to be set to 1 to count this type.

Table 20-48 lists the supplier information field that applies to processors based on Ice Lake microarchitecture.

Table 20-48. MSR_OFFCORE_RSP_x Supplier Info Field Definition
(Processors Based on Ice Lake Microarchitecture)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HWPF_L2_DATA_RD</td>
<td>4</td>
<td>Counts hardware generated data read prefetches targeting the L2 cache.</td>
</tr>
<tr>
<td>HWPF_L2_RFO</td>
<td>5</td>
<td>Counts hardware generated prefetches for exclusive ownership (RFO) targeting the L2 cache.</td>
</tr>
<tr>
<td>Reserved</td>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>HWPF_L3</td>
<td>9:7 and 13</td>
<td>Counts hardware generated prefetches of any type targeting the L3 cache.</td>
</tr>
<tr>
<td>HWPF_L1D_AND_SWP</td>
<td>10</td>
<td>Counts hardware generated data read prefetches targeting the L1 data cache and the following software prefetches (PREFETCHNTA, PREFETCHTO/1/2).</td>
</tr>
<tr>
<td>STREAMING_WR</td>
<td>11</td>
<td>Counts streaming stores.</td>
</tr>
<tr>
<td>Reserved</td>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>14</td>
<td>Reserved</td>
</tr>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Counts miscellaneous requests, such as I/O and un-cacheable accesses.</td>
</tr>
</tbody>
</table>

NOTES:
1. All bits need to be set to 1 to count this type.

Ice Lake microarchitecture has added a new category of Response subtype, called a Combined Response Info. To count a feature in this type, all the bits specified must be set to 1.

A valid response type must be a non-zero value of the following expression:

```
Any | ['OR' of Combined Response Info Bits | ['OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]
```

If "ANY" bit[16] is set, other response type bits [17-39] are ignored.

Table 20-48 lists the supplier information field that applies to processors based on Ice Lake microarchitecture.

Table 20-49 lists the snoop information field that applies to processors based on Ice Lake microarchitecture.
20.3.9.3 Performance Metrics

The Ice Lake core PMU provides built-in support for Top-down Microarchitecture Analysis (TMA) method level 1 metrics. These metrics are always available to cross-validate performance observations, freeing general purpose counters to count other events in high counter utilization scenarios. For more details about the method, refer to Top-Down Analysis Method chapter (Appendix B.1) of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

A new MSR called MSR_PERF_METRICS reports the metrics directly. Software can check (and/or expose to its guests) the availability of the PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15). For additional details on this MSR, refer to Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoop Info</td>
<td>Reserved</td>
<td>30</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_NOT_NEEDED</td>
<td>32</td>
<td>No snoop was needed to satisfy the request.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_MISS</td>
<td>33</td>
<td>A snoop was sent and none of the snooped caches contained the cacheline.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_HIT_NO_FWD</td>
<td>34</td>
<td>A snoop was sent and hit in at least one snooped cache. The unmodified cacheline was not forwarded back, because the L3 already has a valid copy.</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>35</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>SNOOP_HITM</td>
<td>36</td>
<td>A snoop was sent and the cacheline was found modified in another core’s caches. The modified cacheline was forwarded to the requesting core.</td>
</tr>
</tbody>
</table>

Figure 20-37. MSR_PERF_METRICS Definition

This register exposes the four TMA Level 1 metrics. The lower 32 bits are divided into four 8-bit fields, as shown by the above figure, each of which is an integer fraction of 255.
To support built-in performance metrics, new bits have been added to the following MSRs:

- **IA32_PERF_GLOBAL_CTRL. EN_PERF_METRICS[48]**: If this bit is set and fixed-function performance-monitoring counter 3 is enabled, built-in performance metrics are enabled.

- **IA32_PERF_GLOBAL_STATUS_SET. SET_OVF_PERF_METRICS[48]**: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.

- **IA32_PERF_GLOBAL_STATUS_RESET. RESET_OVF_PERF_METRICS[48]**: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for PERF_METRICS.

- **IA32_PERF_GLOBAL_STATUS. OVF_PERF_METRICS[48]**: If this bit is set, it indicates that a PERF_METRICS-related resource has overflowed and a PMI is triggered\(^4\). If this bit is clear, no such overflow has occurred.

**NOTE**
Software has to synchronize, e.g., re-start, fixed-function performance-monitoring counter 3 as well as PERF_METRICS when either bit 35 or 48 in IA32_PERF_GLOBAL_STATUS is set. Otherwise, PERF_METRICS may return undefined values.

The values in MSR_PERF_METRICS are derived from fixed-function performance-monitoring counter 3. Software should start both registers, PERF_METRICS and fixed-function performance-monitoring counter 3, from zero. Additionally, software is recommended to periodically clear both registers in order to maintain accurate measurements for certain scenarios that involve sampling metrics at high rates.

In order to save/restore PERF_METRICS, software should follow these guidelines:

- PERF_METRICS and fixed-function performance-monitoring counter 3 should be saved and restored together.
- To ensure that PERF_METRICS and fixed-function performance-monitoring counter 3 remain synchronized, both should be disabled during both save and restore. Software should enable/disable them atomically, with a single write to IA32_PERF_GLOBAL_CTRL to set/clear both EN_PERF_METRICS[bit 48] and EN_FIXED_CTR3[bit 35].
- On state restore, fixed-function performance-monitoring counter 3 must be restored **before** PERF_METRICS, otherwise undefined results may be observed.

### 20.3.10 12th and 13th Generation Intel® Core™ Processors, and 4th Generation Intel® Xeon® Scalable Processor Family Performance Monitoring Facility

The 12th generation Intel® Core™ processor supports Alder Lake performance hybrid architecture. These processors offer a unique combination of Performance and Efficient-cores (P-core and E-core). The P-core is based on Golden Cove microarchitecture and the E-core is based on Gracemont microarchitecture. The 13th generation Intel® Core™ processor supports Raptor Lake performance hybrid architecture, utilizing both Raptor Cove cores and enhanced Gracemont cores. The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture utilizing Golden Cove cores. These processors all report architectural performance monitoring version ID = 5 and support non-architectural monitoring capabilities described in this section.

#### 20.3.10.1 P-core Performance Monitoring Unit

The P-core PMU’s capability is similar to those described in Section 20.3.1 through Section 20.3.9, with some differences and enhancements summarized in Table 20-50.

---

4. An overflow of fixed-function performance-monitoring counter 3 should normally happen first if software follows Intel’s recommendations.
### Table 20-50. Core PMU Summary of the Golden Cove Microarchitecture

<table>
<thead>
<tr>
<th>Box</th>
<th>Golden Cove Microarchitecture</th>
<th>Ice Lake Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Perfmon version</td>
<td>5</td>
<td>5</td>
<td>See Section 20.2.5.</td>
</tr>
<tr>
<td>Event-Counter Restrictions</td>
<td>Simplified identification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Metrics</td>
<td>Yes (12)</td>
<td>Yes (4)</td>
<td></td>
</tr>
<tr>
<td>PEBS: Baseline, record format</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PEBS: EPT-friendly</td>
<td>Yes</td>
<td>No; debuts in Ice Lake server microarchitecture</td>
<td></td>
</tr>
<tr>
<td>PEBS: Precise Distribution</td>
<td>IA32_FIXED0 instruction-granularity</td>
<td>IA32_FIXED0 cycle-granularity</td>
<td>See Section 20.9.6.</td>
</tr>
<tr>
<td>PEBS: Store Latency</td>
<td>Cache latency Access info fields (3)</td>
<td>None</td>
<td>See Section 20.9.8.</td>
</tr>
<tr>
<td>PEBS: Intel TSX support</td>
<td>Abort info fields (9)</td>
<td>Abort info fields (8)</td>
<td>See Section 20.3.6.5.1. (Intel Xeon processor only feature.)</td>
</tr>
</tbody>
</table>

#### 20.3.10.1.1 P-core Perf Metrics Extensions

For 12th generation Intel Core processor P-cores, the core PMU supports the built-in metrics that were introduced in the Ice Lake microarchitecture PMU. This core PMU extends the PERF_METRICS MSR to feature TMA method level 2 metrics, as shown in Figure 20-38.

![Figure 20-38. PERF_METRICS MSR Definition for 12th Generation Intel® Core™ Processor P-core](image-url)
The lower half of the register is the TMA level 1 metrics (legacy). The upper half is also divided into four 8-bit fields, each of which is an integer fraction of 255. Additionally, each of the new level 2 metrics in the upper half is a subset of the corresponding level 1 metric in the lower half (that is, its parent node per the TMA hierarchy). This enables software to deduce the other four level 2 metrics by subtracting corresponding metrics as shown in Figure 20-39.

![Figure 20-39. Deducing Implied Level 2 Metrics in the Core PMU for 12th Generation Intel® Core™ Processor P-core](image)

The PERF_METRICS MSR and fixed-function performance-monitoring counter 3 of the core PMU feature 12 metrics in total that cover all level 1 and level 2 nodes of the TMA hierarchy.

### 20.3.10.1.2 P-core Counter Restrictions Simplification

The 12th generation Intel Core processor P-core allows identification of performance monitoring events with counter restrictions based on event encodings. The general rule is: Event Codes < 0x90 are restricted to general-purpose performance-monitoring counters 0-3. Event Codes ≥ 0x90 are likely to have no restrictions. Table 20-51 lists the exceptions to this rule.

![Table 20-51. Special Performance Monitoring Events with Counter Restrictions](table)

### 20.3.10.1.3 P-core Off-core Response Facility

For the 12th generation Intel Core processor P-core, the Off-core Response (OCR) Facility is similar to that described in Section 20.3.9.2.

The following enhancements are introduced for the Request_Type of MSR_OFFCORE_RSP_x:

- **WB** (bits 3 and 12): Count writeback (modified or non-modified) transactions by core caches.
- **HWPF_L1D** (bit 10): Counts hardware generated data read prefetches targeting the L1 data cache (only).
- **SWPF_READ** (bit 14): Counts software generated data read prefetches by the PREFETCHNTA and PREFETCHT0/1/2 instructions.

![Light_Operations = Retiring - Heavy_Operations](image)

![Machine_Clears = Bad_Speculation - Branch_Mispredicts](image)

![Fetch_Bandwidth = Frontend_Bound - Fetch_Latency](image)

![Core_Bound = Backend_Bound - Memory_Bound](image)
20.3.10.2  E-core Performance Monitoring Unit

The core PMU capabilities on the 12th generation Intel Core processor E-core are summarized in Table 20-52 below.

<table>
<thead>
<tr>
<th>Box</th>
<th>Gracemont Microarchitecture</th>
<th>Tremont Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of fixed-function performance-monitoring counters per core</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to enumerate number of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>Number of general-purpose counters per core</td>
<td>6</td>
<td>4</td>
<td>Use CPUID to enumerate number of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>Architectural Performance Monitoring version ID</td>
<td>5</td>
<td>5</td>
<td>See Section 20.2.5.</td>
</tr>
<tr>
<td>PEBS record format encoding</td>
<td>0100b</td>
<td>0100b</td>
<td>See Section 20.5.5.</td>
</tr>
<tr>
<td>EPT-friendly PEBS support</td>
<td>Yes</td>
<td>No</td>
<td>See Section 20.9.5.</td>
</tr>
<tr>
<td>Extended PEBS</td>
<td>Yes</td>
<td>Yes</td>
<td>See Section 20.9.1.</td>
</tr>
<tr>
<td>Adaptive PEBS</td>
<td>Yes</td>
<td>Yes</td>
<td>See Section 20.9.2.</td>
</tr>
<tr>
<td>Precise distribution (PDist) PEBS</td>
<td>IA32_PMC0 and IA32_FIXED_CTR0</td>
<td>IA32_PMC0 and IA32_FIXED_CTR0</td>
<td>PDist eliminates skid, see Section 20.9.3, Section 20.9.4, and Section 20.9.6.</td>
</tr>
<tr>
<td>PEBS Latency</td>
<td>Load and Store Latency</td>
<td>No</td>
<td>See Section 20.3.10.2.1, Section 20.3.10.2.2, Section 20.9.7, and Section 20.9.8.</td>
</tr>
<tr>
<td>PEBS Output</td>
<td>DS Save Area or Intel® Processor Trace</td>
<td>DS Save Area or Intel® Processor Trace</td>
<td>See Section 20.5.5.2.1.</td>
</tr>
<tr>
<td>Offcore Response</td>
<td>MSR 01A6H and 01A7H, each core has its own register, extended request and response types.</td>
<td>MSR 1A6H and 1A7H, each core has its own register, extended request and response types.</td>
<td>See Section 20.5.5.4.</td>
</tr>
</tbody>
</table>

20.3.10.2.1  E-core PEBS Load Latency

The 12th generation Intel Core processor E-core includes PEBS Load Latency support similar to that described in Section 20.9.7.

When a programmable counter is configured to count MEM_UOPS_RETIRED.LOAD_LATENCY_ABOVE_THRESHOLD (IA32_PERFEVTSELx[15:0] = 0xD005, with CMASK=0 and INV=0), selected load operations whose latency exceeds the threshold provided in MSR_PEBS_LD_LAT_THRESHOLD (MSR 03F6H) will be counted. If a PEBS record is generated on overflow of this counter, the Memory Access Latency and Memory Auxiliary Info data is reported in the Memory Access Info group (Section 20.9.2.2.2). The formats of these fields are shown in Table 20-53 and Table 20-94.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>Data Source</td>
<td>The source of the data; see Table 20-54.</td>
</tr>
<tr>
<td>4</td>
<td>Lock</td>
<td>0: The operation was not part of a locked transaction. 1: The operation was part of a locked transaction.</td>
</tr>
</tbody>
</table>
PERFORMANCE MONITORING

For details on E-core PEBS memory access latency encoding, see the Access Latency Field in Table 20-94.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>Unknown Data Source (the processor could not retrieve the origin of this request) and MMIO. Memory mapped I/O hit.</td>
</tr>
<tr>
<td>01H</td>
<td>L1 HIT. This request was satisfied by the L1 data cache. (Minimal latency core cache hit.)</td>
</tr>
<tr>
<td>02H</td>
<td>FB HIT. Outstanding core cache miss to same cache-line address was already underway. (Pending core cache hit.)</td>
</tr>
<tr>
<td>03H</td>
<td>L2 HIT. This request was satisfied by the L2 cache.</td>
</tr>
<tr>
<td>04H</td>
<td>L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).</td>
</tr>
<tr>
<td>05H</td>
<td>L3 HITE. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where no modified copies were found (clean).</td>
</tr>
<tr>
<td>06H</td>
<td>L3 HITM. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where a modified copy was found.</td>
</tr>
<tr>
<td>07H</td>
<td>Reserved.</td>
</tr>
<tr>
<td>08H</td>
<td>L3 HITF. Local or Remote home requests that hit the L3 cache and were serviced by another processor core with a cross core snoop where a shared or forwarding copy was found.</td>
</tr>
<tr>
<td>09H</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0AH</td>
<td>L3 MISS. Local home requests that missed the L3 cache and were serviced by local DRAM (go to shared state).</td>
</tr>
<tr>
<td>0BH</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0CH</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0DH</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0EH</td>
<td>I/O. Request of input/output operation.</td>
</tr>
<tr>
<td>0FH</td>
<td>The request was to un-cacheable memory.</td>
</tr>
</tbody>
</table>

**Table 20-53. E-core PEBS Memory Access Info Encoding (Contd.)**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>STLB_MISS</td>
<td>0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.</td>
</tr>
<tr>
<td>6</td>
<td>ST_FWD_BLK</td>
<td>0: Load did not get a store forward block. 1: Load got a store forward block.</td>
</tr>
<tr>
<td>63:7</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

20.3.10.2.2 E-core PEBS Store Latency

The 12th generation Intel Core processor E-core includes PEBS Store Latency support. When a programmable counter is configured to count MEM_UOPS_RETIRED.STORE_LATENCY (IA32_PERFEVTSELx[15:0] = 0xD006, with CMASK=0 and INV=0), all store operations will be counted. If a PEBS record is generated on overflow of this counter, the Memory Access Latency and Memory Auxiliary Info data is reported in the Memory Access Info group (Section 18.9.2.2.2). The formats of these fields are shown in Table 20-53 and Table 20-94.

20.3.10.2.3 E-core Precise Distribution (PDist) Support

The 12th generation Intel Core processor E-core supports PEBS with Precise Distribution (PDist) on IA32_PMC0 and IA32_FIXED_CTR0. All precise events support PDist save for UOPS_RETIRED. See Section 20.9.6 for additional details on PDist.
20.3.10.2.4 E-core Enhanced Off-core Response

Event number 0B7H support off-core response monitoring using an associated configuration MSR, MSR_OFFSETCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFSETCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. There are unique pairs of MSR_OFFSETCORE_RSPx registers per core. The layout of MSR_OFFSETCORE_RSP0 and MSR_OFFSETCORE_RSP1 are organized as follows:

- Bits 15:0 and bits 49:44 specify the request type of a transaction request to the uncore. This is described in Table 20-55.
- Bits 30:16 specify Response Type information or an L2 Hit, and is described in Table 20-75.
- If L2 misses, then bits 37:31 can be used to specify snoop response information and is described in Table 20-76.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 20.5.2.3 for details.

Table 20-55. MSR_OFFSETCORE_RSPx Request Type Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMAND_DATA_RD</td>
<td>0</td>
<td>Counts demand data reads.</td>
</tr>
<tr>
<td>DEMAND_RFO</td>
<td>1</td>
<td>Counts all demand reads for ownership (RFO) requests and software based prefetches for exclusive ownership (prefetchw).</td>
</tr>
<tr>
<td>DEMAND_CODE_RD</td>
<td>2</td>
<td>Counts demand instruction fetches and L1 instruction cache prefetches.</td>
</tr>
<tr>
<td>COREWB_M</td>
<td>3</td>
<td>Counts modified write backs from L1 and L2.</td>
</tr>
<tr>
<td>HWPF_L2_DATA_RD</td>
<td>4</td>
<td>Counts prefetch (that bring data to L2) data reads.</td>
</tr>
<tr>
<td>HWPF_L2_RFO</td>
<td>5</td>
<td>Counts all prefetch (that bring data to L2) RFOs.</td>
</tr>
<tr>
<td>HWPF_L2_CODE_RD</td>
<td>6</td>
<td>Counts all prefetch (that bring data to MLC only) code reads.</td>
</tr>
<tr>
<td>HWPF_L3_DATA_RD</td>
<td>7</td>
<td>Counts L3 cache hardware prefetch data reads (written to the L3 cache only).</td>
</tr>
<tr>
<td>HWPF_L3_RFO</td>
<td>8</td>
<td>Counts L3 cache hardware prefetch RFOs (written to the L3 cache only).</td>
</tr>
<tr>
<td>HWPF_L3_CODE_RD</td>
<td>9</td>
<td>Counts L3 cache hardware prefetch code reads (written to the L3 cache only).</td>
</tr>
<tr>
<td>HWPF_L1D_AND_SwpF</td>
<td>10</td>
<td>Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).</td>
</tr>
<tr>
<td>STREAMING_WR</td>
<td>11</td>
<td>Counts all streaming stores.</td>
</tr>
<tr>
<td>COREWB_NONM</td>
<td>12</td>
<td>Counts non-modified write backs from L2.</td>
</tr>
<tr>
<td>RSVD</td>
<td>14:13</td>
<td>Reserved.</td>
</tr>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Counts miscellaneous requests, such as I/O accesses that have any response type.</td>
</tr>
<tr>
<td>UC_RD</td>
<td>44</td>
<td>Counts uncached memory reads (PRd, UCRdF).</td>
</tr>
<tr>
<td>UC_WR</td>
<td>45</td>
<td>Counts uncached memory writes (WiL).</td>
</tr>
<tr>
<td>PARTIAL_STREAMING_WR</td>
<td>46</td>
<td>Counts partial (less than 64 byte) streaming stores (wCiL).</td>
</tr>
<tr>
<td>FULL_STREAMING_WR</td>
<td>47</td>
<td>Counts full, 64 byte streaming stores (wCiLF).</td>
</tr>
<tr>
<td>L1WB_M</td>
<td>48</td>
<td>Counts modified WriteBacks from L1 that miss the L2.</td>
</tr>
<tr>
<td>L2WB_M</td>
<td>49</td>
<td>Counts modified WriteBacks from L2.</td>
</tr>
</tbody>
</table>

20.3.10.3 Unhalted Reference Cycles

The Unhalted Reference Cycles architectural performance monitoring event is enhanced to count at TSC-rate in the 12th generation Intel Core processor P-core when used on a general-purpose PMC. This enhancement makes it consistent with the fixed-function counter 2 and the E-core. As a result, this event is kept enumerated in CPUID leaf 0AH.EBX (unlike prior hybrid parts).
20.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

 NOTE
This section also applies to the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture.

20.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.


A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel Atom® processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 20.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. The processor supports AnyThread counting in three architectural performance monitoring events.

20.4.1.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ Processor Tile

The Intel® Xeon Phi™ processor tile includes the following enhancements to the Silvermont microarchitecture.

- AnyThread support. This facility is limited to following three architectural events: Instructions Retired, Unhalted Core Cycles, Unhalted Reference Cycles using IA32_FIXED_CTR0-2 and Unhalted Core Cycles, Unhalted Reference Cycles using IA32_PERFEVTSELx.

- PEBS-DLA (Processor Event-Based Sampling-Data Linear Address) fields. The processor provides memory address in addition to the Silvermont PEBS record support on select events. The PEBS recording format as reported by IA32_PERF_CAPABILITIES [11:8] is 2.

- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor tile to subsystems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx. Two cores do not share the off-core response MSRs. Knights Landing expands off-core response capability to match the processor untile changes.

- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests. This facility is updated to match the processor untile changes.

20.4.1.1.1 Processor Event-Based Sampling

The processor supports processor event based sampling (PEBS). PEBS is supported using IA32_PMC0 (see also Section 18.4.9, “BTS and DS Save Area”).
PERFORMANCE MONITORING

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 20.6.2.4).

The list of PEBS events supported in the processor is shown in the following table.

### Table 20-56. PEBS Performance Events for Knights Landing Microarchitecture

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Select</th>
<th>Sub-event</th>
<th>UMask</th>
<th>Data Linear Address Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR_INST_RETIRED</td>
<td>C4H</td>
<td>ALL_BRANCHES</td>
<td>00H</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCC</td>
<td>7EH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAKEN_JCC</td>
<td>FEH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CALL</td>
<td>F9H</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REL_CALL</td>
<td>FDH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IND_CALL</td>
<td>FBH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON_RETURN_IND</td>
<td>EBH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR_BRANCH</td>
<td>BFH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN</td>
<td>F7H</td>
<td>No</td>
</tr>
<tr>
<td>BR_MISP_RETIRED</td>
<td>C5H</td>
<td>ALL_BRANCHES</td>
<td>00H</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCC</td>
<td>7EH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAKEN_JCC</td>
<td>FEH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IND_CALL</td>
<td>FBH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON_RETURN_IND</td>
<td>EBH</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN</td>
<td>F7H</td>
<td>No</td>
</tr>
<tr>
<td>MEM_UOPS_RETIRED</td>
<td>04H</td>
<td>L2_HIT_LOADS</td>
<td>02H</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L2_MISS_LOADS</td>
<td>04H</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DLTB_MISS_LOADS</td>
<td>08H</td>
<td>Yes</td>
</tr>
<tr>
<td>RECYCLEQ</td>
<td>03H</td>
<td>LD_BLOCK_ST_FORWARD</td>
<td>01H</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD_SPLITS</td>
<td>08H</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The PEBS record format 2 supported by processors based on the Knights Landing microarchitecture is shown in Table 20-57, and each field in the PEBS record is 64 bits long.

### Table 20-57. PEBS Record Format for Knights Landing Microarchitecture

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field</th>
<th>Byte Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>R/EFLAGS</td>
<td>60H</td>
<td>R10</td>
</tr>
<tr>
<td>08H</td>
<td>R/EIP</td>
<td>68H</td>
<td>R11</td>
</tr>
<tr>
<td>10H</td>
<td>R/EAX</td>
<td>70H</td>
<td>R12</td>
</tr>
<tr>
<td>18H</td>
<td>R/EBX</td>
<td>78H</td>
<td>R13</td>
</tr>
<tr>
<td>20H</td>
<td>R/ECX</td>
<td>80H</td>
<td>R14</td>
</tr>
<tr>
<td>28H</td>
<td>R/EDX</td>
<td>88H</td>
<td>R15</td>
</tr>
<tr>
<td>30H</td>
<td>R/ESI</td>
<td>90H</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
</tr>
<tr>
<td>38H</td>
<td>R/EDI</td>
<td>98H</td>
<td>PSDLA</td>
</tr>
<tr>
<td>40H</td>
<td>R/EBP</td>
<td>A0H</td>
<td>Reserved</td>
</tr>
<tr>
<td>48H</td>
<td>R/ESP</td>
<td>A8H</td>
<td>Reserved</td>
</tr>
<tr>
<td>50H</td>
<td>R8</td>
<td>80H</td>
<td>EventingRIP</td>
</tr>
</tbody>
</table>
20.4.1.1.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 20-58 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

### Table 20-58. OffCore Response Event Encoding

<table>
<thead>
<tr>
<th>Counter</th>
<th>Event code</th>
<th>UMask</th>
<th>Required Off-core Response MSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMC0-1</td>
<td>B7H</td>
<td>01H</td>
<td>MSR_OFFCORE_RSP0 (address 1A6H)</td>
</tr>
<tr>
<td>PMC0-1</td>
<td>B7H</td>
<td>02H</td>
<td>MSR_OFFCORE_RSP1 (address 1A7H)</td>
</tr>
</tbody>
</table>

Some of the MSR_OFFCORE_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 registers are defined in Table 20-59. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 20.5.2.3 for details.

### Table 20-59. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers

<table>
<thead>
<tr>
<th>Main</th>
<th>Sub-field</th>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Type</td>
<td></td>
<td>0</td>
<td>DEMAND_DATA_RD</td>
<td>Demand cacheable data and L1 prefetch data reads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>DEMAND_RFO</td>
<td>Demand cacheable data writes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>DEMAND_CODE_RD</td>
<td>Demand code reads and prefetch code reads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Reserved</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>Reserved</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>PF_L2_RFO</td>
<td>L2 data RFO prefetches (includes PREFETCHW instruction).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>PF_L2_CODE_RD</td>
<td>L2 code HW prefetches.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>PARTIAL_READS</td>
<td>Partial reads (UC or WC).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>PARTIAL_WRITES</td>
<td>Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>UC_CODE_READS</td>
<td>UC code reads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>BUS_LOCKS</td>
<td>Bus locks and split lock requests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>FULL_STREAMING_STORES</td>
<td>Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>SW_PREFETCH</td>
<td>Software prefetches.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>PF_L1_DATA_RD</td>
<td>L1 data HW prefetches.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>PARTIAL_STREAMING_STORES</td>
<td>Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>ANY_REQUEST</td>
<td>Account for any requests.</td>
</tr>
</tbody>
</table>
20.4.1.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR_OFFCORE_RSP0.[bit 38] with the choice of request type specified in MSR_OFFCORE_RSP0.[bit 15:0].

Refer to Section 20.5.2.3, “Average Offcore Request Latency Measurement,” for typical usage. Note that MSR_OFFCORE_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.
20.5 PERFORMANCE MONITORING (INTEL ATOM® PROCESSORS)

20.5.1 Performance Monitoring (45 nm and 32 nm Intel Atom® Processors)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 20.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

**NOTE**

The number of counters available to software may vary from the number of physical counters present on the hardware, because an agent running at a higher privilege level (e.g., a VMM) may not expose all counters. CPUID.0AH:EAX[15:8] reports the MSRs available to software; see Section 20.2.1.

Non-architectural performance monitoring in Intel Atom processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32_PERFEVTSELx MSR, i.e., if IA32_PERFEVTSELx.AnyThread = 1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32_PERFEVTSELx MSR.

20.5.2 Performance Monitoring for Silvermont Microarchitecture

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 20.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32 FIXED CTR0, IA32 FIXED CTR1, IA32 FIXED CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events can be found at: https://perfmon-events.intel.com/.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32_PERFEVTSELx MSR.

20.5.2.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.
• Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

• Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

20.5.2.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32_PMC0 (see also Section 18.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 20.6.2.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 20-60.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Select</th>
<th>Sub-event</th>
<th>UMask</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR_INST_RETIRED</td>
<td>C4H</td>
<td>ALL_BRANCHES</td>
<td>00H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCC</td>
<td>7EH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAKEN_JCC</td>
<td>FEH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CALL</td>
<td>F9H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REL_CALL</td>
<td>FDH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IND_CALL</td>
<td>FBH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON_RETURN_IND</td>
<td>EBH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR_BRANCH</td>
<td>BFH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN</td>
<td>F7H</td>
</tr>
<tr>
<td>BR_MISP_RETIRED</td>
<td>C5H</td>
<td>ALL_BRANCHES</td>
<td>00H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCC</td>
<td>7EH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAKEN_JCC</td>
<td>FEH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IND_CALL</td>
<td>FBH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON_RETURN_IND</td>
<td>EBH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN</td>
<td>F7H</td>
</tr>
<tr>
<td>MEM_UOPS_RETIRED</td>
<td>04H</td>
<td>L2_HIT_LOADS</td>
<td>02H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L2_MISS_LOADS</td>
<td>04H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DLTB_MISS_LOADS</td>
<td>08H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HITM</td>
<td>20H</td>
</tr>
<tr>
<td>REHABQ</td>
<td>03H</td>
<td>LD_BLOCK_ST_FORWARD</td>
<td>01H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD_SPLITS</td>
<td>08H</td>
</tr>
</tbody>
</table>

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 20-61, and each field in the PEBS record is 64 bits long.
20.5.2.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 20-62 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

In the Silvermont microarchitecture, each MSR_OFFCORE_RSPx is shared by two processor cores.

### Table 20-62. OffCore Response Event Encoding

<table>
<thead>
<tr>
<th>Counter</th>
<th>Event code</th>
<th>UMask</th>
<th>Required Off-core Response MSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMC0-1</td>
<td>B7H</td>
<td>01H</td>
<td>MSR_OFFCORE_RSP0 (address 1A6H)</td>
</tr>
<tr>
<td>PMC0-1</td>
<td>B7H</td>
<td>02H</td>
<td>MSR_OFFCORE_RSP1 (address 1A7H)</td>
</tr>
</tbody>
</table>

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 20-40 and Figure 20-41. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 20.5.2.3 for details.
Figure 20-40. Request_Type Fields for MSR_OFFCORE_RSPx

Table 20-63. MSR_OFFCORE_RSPx Request_Type Field Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMND_DATA_RD</td>
<td>0</td>
<td>Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.</td>
</tr>
<tr>
<td>DMND_RFO</td>
<td>1</td>
<td>Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.</td>
</tr>
<tr>
<td>DMND_IFETCH</td>
<td>2</td>
<td>Counts the number of demand instruction cacheline reads and L1 instruction cacheline prefetches.</td>
</tr>
<tr>
<td>WB</td>
<td>3</td>
<td>Counts the number of writeback (modified to exclusive) transactions.</td>
</tr>
<tr>
<td>PF_DATA_RD</td>
<td>4</td>
<td>Counts the number of data cacheline reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_RFO</td>
<td>5</td>
<td>Counts the number of RFO requests generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PF_IFETCH</td>
<td>6</td>
<td>Counts the number of code reads generated by L2 prefetchers.</td>
</tr>
<tr>
<td>PARTIAL_READ</td>
<td>7</td>
<td>Counts the number of demand reads of partial cache lines (including UC and WC).</td>
</tr>
<tr>
<td>PARTIAL_WRITE</td>
<td>8</td>
<td>Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT, and WP).</td>
</tr>
<tr>
<td>UC_IFETCH</td>
<td>9</td>
<td>Counts the number of UC instruction fetches.</td>
</tr>
<tr>
<td>BUS_LOCKS</td>
<td>10</td>
<td>Bus lock and split lock requests.</td>
</tr>
<tr>
<td>STRM_ST</td>
<td>11</td>
<td>Streaming store requests.</td>
</tr>
<tr>
<td>SW_PREFETCH</td>
<td>12</td>
<td>Counts software prefetch requests.</td>
</tr>
<tr>
<td>PF_DATA_RD</td>
<td>13</td>
<td>Counts DCU hardware prefetcher data read requests.</td>
</tr>
<tr>
<td>PARTIAL_STRM_ST</td>
<td>14</td>
<td>Streaming store requests.</td>
</tr>
<tr>
<td>ANY</td>
<td>15</td>
<td>Any request that crosses IDI, including I/O.</td>
</tr>
</tbody>
</table>
To properly program this extra register, software must set at least one request type bit (Table 20-63) and a valid
response type pattern (Table 20-64, Table 20-65). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

\[ \text{ANY} | (\text{OR of Supplier Info Bits}) \land (\text{OR of Snoop Info Bits}) \]

If "ANY" bit is set, the supplier and snoop info bits are ignored.

---

**Table 20-64. MSR_OFFCORE_RSPx Response Supplier Info Field Definition**

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td>ANY_RESPONSE</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>Supplier Info</td>
<td>Reserved</td>
<td>17</td>
<td>Reserved</td>
</tr>
<tr>
<td>L2_HIT</td>
<td>18</td>
<td>Cache reference hit L2 in either M/E/S states.</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>30:19</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**Table 20-65. MSR_OFFCORE_RSPx Snoop Info Field Definition**

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoop Info</td>
<td>SNP_NONE</td>
<td>31</td>
<td>No details on snoop-related information.</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>32</td>
<td>Reserved</td>
</tr>
<tr>
<td>SNOOP_MISS</td>
<td>SNOOP_MISS</td>
<td>33</td>
<td>Counts the number of snoop misses when L2 misses.</td>
</tr>
<tr>
<td>SNOOP_HIT</td>
<td>SNOOP_HIT</td>
<td>34</td>
<td>Counts the number of snoops hit in the other module where no modified copies were found.</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>35</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
20.5.2.3  Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR_OFFCORE_RSP registers. Using two performance monitoring counters, program the two OFFCORE_RESPONSE event encodings into the corresponding IA32_PERFEVTSELx MSRs. Count the weighted cycles via MSR_OFFCORE_RSP0 by programming a request type in MSR_OFFCORE_RSP0.[15:0] and setting MSR_OFFCORE_RSP0.OUTSTANDING[38] to 1, while setting the remaining bits to 0. Count the number of requests via MSR_OFFCORE_RSP1 by programming the same request type from MSR_OFFCORE_RSP0 into MSR_OFFCORE_RSP1[bit 15:0], and setting MSR_OFFCORE_RSP1.ANY_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be obtained by dividing the value of the IA32_PMCx register that counted weight cycles by the register that counted requests.

20.5.3  Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture report architectural performance monitoring versionID = 4 (see Section 20.2.4) and support non-architectural monitoring capabilities described in this section. Architectural performance monitoring version 4 capabilities are described in Section 20.2.4.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 20-6 and described in Section 20.2.1.1 and Section 20.2.3. The Goldmont microarchitecture does not support Hyper-Threading and thus architectural and non-architectural performance monitoring events ignore the AnyThread qualification regardless of its setting in the IA32_PERFEVTSELx MSR. However, Goldmont does not set the AnyThread deprecation bit (CPUID.0AH:EDX[15]).

The core PMU’s capability is similar to that of the Silvermont microarchitecture described in Section 20.5.2, with some differences and enhancements summarized in Table 20-66.

Table 20-66. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

<table>
<thead>
<tr>
<th>Box</th>
<th>Goldmont Microarchitecture</th>
<th>Silvermont Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Fixed counters per core</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of general-purpose counters per core</td>
<td>4</td>
<td>2</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>Architectural Performance Monitoring version ID</td>
<td>4</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
</tbody>
</table>
20.5.3.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32_PMC0 for all events (see Section 18.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way on Goldmont microarchitecture as on the Silvermont microarchitecture. The record will be generated after an instruction that causes the event when the counter is already overflowed and will capture the architectural state at this point (see Section 20.6.2.4 and Section 18.4.9). The eventingIP in the record will indicate the instruction that caused the event. The list of precise events supported in the Goldmont microarchitecture is shown in Table 20-67.
In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor’s current retirement point. It is likely that the instruction fetch that caused the event to increment was beyond that current retirement point. Other examples of non-precise events are CPU_CLK_UNHALTED.CORE_P and HARDWARE_INTERRUPTS.RECEIVED. CPU_CLK_UNHALTED.CORE_P will increment each cycle that the processor is awake. When the counter over-flows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32_PERF_GLOBAL_STATUS_SET.

### Table 20-67. Precise Events Supported by the Goldmont Microarchitecture

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Select</th>
<th>Sub-event</th>
<th>UMask</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD_BLOCKS</td>
<td>03H</td>
<td>DATA_UNKNOWN 01H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORE_FORWARD 02H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4K_ALIAS 04H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>UTLB_MISS 08H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALL_BLOCK 10H</td>
<td></td>
</tr>
<tr>
<td>MISALIGN_MEM_REF</td>
<td>13H</td>
<td>LOAD_PAGE_SPLIT 02H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STORE_PAGE_SPLIT 04H</td>
<td></td>
</tr>
<tr>
<td>INST RETIRED</td>
<td>00H</td>
<td>ANY 00H</td>
<td></td>
</tr>
<tr>
<td>UOPS RETIRED</td>
<td>00H</td>
<td>ANY 00H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD_SPLITSMS 01H</td>
<td></td>
</tr>
<tr>
<td>BR INST RETIRED</td>
<td>00H</td>
<td>ALL_BRANCHES 00H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCC 7EH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAKEN_JCC FEH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CALL F9H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>REL_CALL FDH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IND_CALL FBH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON_RETURN_IND EBH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAR_BRANCH BFH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN F7H</td>
<td></td>
</tr>
<tr>
<td>BR MISP RETIRED</td>
<td>00H</td>
<td>ALL_BRANCHES 00H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCC 7EH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAKEN_JCC FEH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IND_CALL FBH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON_RETURN_IND EBH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN F7H</td>
<td></td>
</tr>
</tbody>
</table>
The PEBS record format supported by processors based on the Goldmont microarchitecture is shown in Table 20-68, and each field in the PEBS record is 64 bits long.

### Table 20-68. PEBS Record Format for the Goldmont Microarchitecture

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field</th>
<th>Byte Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>R/EFLAGS</td>
<td>68H</td>
<td>R11</td>
</tr>
<tr>
<td>08H</td>
<td>R/EIP</td>
<td>70H</td>
<td>R12</td>
</tr>
<tr>
<td>10H</td>
<td>R/EAX</td>
<td>78H</td>
<td>R13</td>
</tr>
<tr>
<td>18H</td>
<td>R/EBX</td>
<td>80H</td>
<td>R14</td>
</tr>
<tr>
<td>20H</td>
<td>R/ECX</td>
<td>88H</td>
<td>R15</td>
</tr>
<tr>
<td>28H</td>
<td>R/EDX</td>
<td>90H</td>
<td>Applicable Counters</td>
</tr>
<tr>
<td>30H</td>
<td>R/ESI</td>
<td>98H</td>
<td>Data Linear Address</td>
</tr>
<tr>
<td>38H</td>
<td>R/EDI</td>
<td>A0H</td>
<td>Reserved</td>
</tr>
<tr>
<td>40H</td>
<td>R/EBP</td>
<td>A8H</td>
<td>Reserved</td>
</tr>
<tr>
<td>48H</td>
<td>R/ESP</td>
<td>B0H</td>
<td>EventingRIP</td>
</tr>
<tr>
<td>50H</td>
<td>R8</td>
<td>B8H</td>
<td>Reserved</td>
</tr>
<tr>
<td>58H</td>
<td>R9</td>
<td>C0H</td>
<td>TSC</td>
</tr>
<tr>
<td>60H</td>
<td>R10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the “Applicable Counter” field, which indicates which counters actually requested generating a PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event even when multiple counters are configured to record PEBS records and multiple bits are set in the field. Additionally, offset COH captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.
20.5.3.1.1 PEBS Data Linear Address Profiling
Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling; those fields are present in the record but are reserved.

For Goldmont microarchitecture, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g., the one that triggered a precise event that caused the PEBS record to be generated). Goldmont microarchitecture may record a Data Linear Address for the instruction that caused the event even for events not related to memory accesses. This may differ from other microarchitectures.

20.5.3.1.2 Reduced Skid PEBS
Processors based on Goldmont Plus microarchitecture support the Reduced Skid PEBS feature described in Section 20.9.4 on the IA32_PMC0 counter. Although Extended PEBS adds support for generating PEBS records for precise events on additional general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

20.5.3.1.3 Enhancements to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62]
In addition to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62] being set when PEBS_Index reaches the PEBS_Interrupt_Theshold, the bit is also set when PEBS_Index is out of bounds. That is, the bit will be set when PEBS_Index < PEBS_Buffer_Base or PEBS_Index > PEBS_Absolute_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32_PERF_GLOBAL_STATUS will be cleared according to Applicable Counters, however the IA32_PMCx values will not be reloaded with the Reset values stored in the DS_AREA.

20.5.3.2 Offcore Response Event
Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with UMASK value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with UMASK value 02H. Table 20-62 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR_OFFCORE_RSPx registers per core.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:
• Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 20-69.
• Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 20-64.
• If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 20-70.
• For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 20.5.2.3 for details.

### Table 20-69. MSR_OFFCORE_RSPx Request_Type Field Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMAND_DATA_RD</td>
<td>0</td>
<td>Counts cacheline read requests due to demand reads (excludes prefetches).</td>
</tr>
<tr>
<td>DEMAND_RFO</td>
<td>1</td>
<td>Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).</td>
</tr>
<tr>
<td>DEMAND_CODE_RD</td>
<td>2</td>
<td>Counts demand instruction cacheline and l-side prefetch requests that miss the instruction cache.</td>
</tr>
<tr>
<td>COREwb</td>
<td>3</td>
<td>Counts writeback transactions caused by L1 or L2 cache evictions.</td>
</tr>
<tr>
<td>PF_L2_DATA_RD</td>
<td>4</td>
<td>Counts data cacheline reads generated by hardware L2 cache prefetcher.</td>
</tr>
<tr>
<td>PF_L2_RFO</td>
<td>5</td>
<td>Counts reads for ownership (RFO) requests generated by L2 prefetcher.</td>
</tr>
<tr>
<td>Reserved</td>
<td>6</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
To properly program this extra register, software must set at least one request type bit (Table 20-63) and a valid response type pattern (either Table 20-64 or Table 20-70). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

\[ \text{Any Response Bit} \mid \text{L2 Hit} \mid \text{'OR' of Snoop Info Bits} \mid \text{Outstanding Bit} \]

### 20.5.3.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 20.5.2.3.
20.5.4 Performance Monitoring for Goldmont Plus Microarchitecture

Intel Atom processors based on the Goldmont Plus microarchitecture report architectural performance monitoring versionID = 4 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 20.2.4.

Goldmont Plus performance monitoring capabilities are similar to Goldmont capabilities. The differences are in specific events and in which counters support PEBS. Goldmont Plus introduces the ability for fixed performance monitoring counters to generate PEBS records.

Goldmont Plus will set the AnyThread deprecation CPUID bit (CPUID.0AH:EDX[15]) to indicate that the Any-Thread bits in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL have no effect.

The core PMU's capability is similar to that of the Goldmont microarchitecture described in Section 20.6.3, with some differences and enhancements summarized in Table 20-71.

<table>
<thead>
<tr>
<th>Box</th>
<th>Goldmont Plus Microarchitecture</th>
<th>Goldmont Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Fixed counters per core</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of general-purpose counters per core</td>
<td>4</td>
<td>4</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>Architectural Performance Monitoring version ID</td>
<td>4</td>
<td>4</td>
<td>No change.</td>
</tr>
<tr>
<td>Processor Event Based Sampling (PEBS) Events</td>
<td>All General-Purpose and Fixed counters. Each General-Purpose counter supports all events (precise and non-precise).</td>
<td>General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 20-67.</td>
<td>Goldmont Plus supports PEBS on all counters.</td>
</tr>
<tr>
<td>PEBS record format encoding</td>
<td>0011b</td>
<td>0011b</td>
<td>No change.</td>
</tr>
</tbody>
</table>

20.5.4.1 Extended PEBS

The PEBS facility in Goldmont Plus microarchitecture provides a number of enhancements relative to PEBS in processors from previous generations. Enhancement of PEBS facility with the Extended PEBS feature are described in detail in section 18.9.

20.5.5 Performance Monitoring for Tremont Microarchitecture

Intel Atom processors based on the Tremont microarchitecture report architectural performance monitoring versionID = 5 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 5 capabilities are described in Section 20.2.5.

Tremont performance monitoring capabilities are similar to Goldmont Plus capabilities, with the following extensions:

- Support for Adaptive PEBS.
- Support for PEBS output to Intel® Processor Trace.
- Precise Distribution support on Fixed Counter 0.
- Compatibility enhancements to off-core response MSRs, MSR_OFFCORE_RSPx.
The differences and enhancements between Tremont microarchitecture and Goldmont Plus microarchitecture are summarized in Table 20-72.

### Table 20-72. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures

<table>
<thead>
<tr>
<th>Box</th>
<th>Tremont Microarchitecture</th>
<th>Goldmont Plus Microarchitecture</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td># of fixed counters per core</td>
<td>3</td>
<td>3</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td># of general-purpose counters per core</td>
<td>4</td>
<td>4</td>
<td>Use CPUID to determine # of counters. See Section 20.2.1.</td>
</tr>
<tr>
<td>Architectural Performance Monitoring version ID</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PEBS record format encoding</td>
<td>0100b</td>
<td>0011b</td>
<td>See Section 20.6.2.4.2.</td>
</tr>
<tr>
<td>Reduce skid PEBS</td>
<td>IA32_PMC0 and IA32_FIXED_CTR0</td>
<td>IA32_PMC0 only</td>
<td></td>
</tr>
<tr>
<td>Extended PEBS</td>
<td>Yes</td>
<td>Yes</td>
<td>See Section 20.5.4.1.</td>
</tr>
<tr>
<td>Adaptive PEBS</td>
<td>Yes</td>
<td>No</td>
<td>See Section 20.9.2.</td>
</tr>
<tr>
<td>PEBS output</td>
<td>DS Save Area or Intel® Processor Trace</td>
<td>DS Save Area only</td>
<td>See Section 20.5.5.2.1.</td>
</tr>
<tr>
<td>PEBS record layout</td>
<td>See Section 20.9.2.3 for output to DS, Section 20.5.5.2.2 for output to Intel PT.</td>
<td>Table 20-68; enhanced fields at offsets 90H-98H; and TSC record field at COH.</td>
<td></td>
</tr>
<tr>
<td>Off-core Response Event</td>
<td>MSR 1A6H and 1A7H, each core has its own register, extended request and response types.</td>
<td>MSR 1A6H and 1A7H, each core has its own register.</td>
<td></td>
</tr>
</tbody>
</table>

#### 20.5.5.1 Adaptive PEBS

The PEBS record format and configuration interface has changed versus Goldmont Plus, as the Tremont microarchitecture includes support for the configurable Adaptive PEBS records; see Section 20.9.2.

#### 20.5.5.2 PEBS output to Intel® Processor Trace

Intel Atom processors based on the Tremont microarchitecture introduce the following Precise Event-Based Sampling (PEBS) extensions:

- A mechanism to direct PEBS output into the Intel® Processor Trace (Intel® PT) output stream. In this scenario, the PEBS record is written in packetized form, in order to co-exist with other Intel PT trace data.
- New Performance Monitoring counter reload MSRs, which are used by PEBS in place of the counter reload values stored in the DS Management area when PEBS output is directed into the Intel PT output stream.

Processors that indicate support for Intel PT by setting CPUID.07H.0.EBX[25]=1, and set the new IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16] bit, support these extensions.

#### 20.5.5.2.1 PEBS Configuration

PEBS output to Intel Processor Trace includes support for two new fields in IA32 PEBS ENABLE.
When PEBS_OUTPUT is set to 01B, the DS Management Area is not used and need not be configured. Instead, the output mechanism is configured through IA32_RTIT_CTL and other Intel PT MSRs, while counter reload values are configured in the MSR_RELOAD_PMCx MSRs. Details on configuring Intel PT can be found in Section 33.2.7.

**20.5.5.2.2 PEBS Record Format in Intel® Processor Trace**

The format of the PEBS record changes when output to Intel PT, as the PEBS state is packetized. Each PEBS grouping is emitted as a Block Begin (BBP) and following Block Item (BIP) packets. A PEBS grouping ends when either a new PEBS grouping begins (indicated by a BBP packet) or a Block End (BEP) packet is encountered. See Section 33.4.1.1 for details of these Intel PT packets.

Because the packet headers describe the state held in the packet payload, PEBS state ordering is not fixed. PEBS state groupings may be emitted in any order, and the PEBS state elements within those groupings may be emitted in any order. Further, there is no packet that provides indication of “Record Format” or “Record Size”.

If Intel PT tracing is not enabled (IA32_RTIT_STATUS.TriggerEn=0), any PEBS records triggered will be dropped. PEBS packets do not depend on ContextEn or FilterEn in IA32_RTIT_STATUS, any filtering of PEBS must be enabled from within the PerfMon configuration. Counter reload will occur in all scenarios where PEBS is triggered, regardless of TriggerEn.

<table>
<thead>
<tr>
<th>Field Description</th>
<th>PEBS_OUTPUT[62:61]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pend a PerfMon Interrupt (PMI) after each PEBS event.</td>
<td>PEBS_OUTPUT[62:61]</td>
</tr>
<tr>
<td>Specifies PEBS output destination. Encodings: 00B: DS Save Area. Matches legacy PEBS behavior, output location defined by IA32_DS_AREA. 01B: Intel PT trace output. 10B: Reserved. 11B: Reserved.</td>
<td>PEBS_OUTPUT[62:61]</td>
</tr>
</tbody>
</table>

**Table 20-73. New Fields in IA32_PEBS_ENABLE**

When PEBS_OUTPUT is set to 01B, the DS Management Area is not used and need not be configured. Instead, the output mechanism is configured through IA32_RTIT_CTL and other Intel PT MSRs, while counter reload values are configured in the MSR_RELOAD_PMCx MSRs. Details on configuring Intel PT can be found in Section 33.2.7.

**Figure 20-42. IA32_PEBS_ENABLE MSR with PEBS Output to Intel® Processor Trace**
The PEBS threshold mechanism for generating PerfMon Interrupts (PMIs) is not available in this mode. However, there exist other means to generate PMIs based on PEBS output. When the Intel PT ToPA output mechanism is chosen, a PMI can optionally be pended when a ToPA region is filled; see Section 33.2.7.2 for details. Further, software can opt to generate a PMI on each PEBS record by setting the new IA32_PEBS_ENABLE.PMI_AFTER_EACH_RECORD[60] bit.

The IA32_PERF_GLOBAL_STATUS.OvfDSBuffer bit will not be set in this mode.

### 20.5.5.2.3 PEBS Counter Reload

When PEBS output is directed into Intel PT (IA32_PEBS_ENABLE.PEBS_OUTPUT = 01B), new MSR_RELOAD_PMCx MSRs are used by the PEBS routine to reload PerfMon counters. The value from the associated reload MSR will be loaded to the appropriate counter on each PEBS event.

### 20.5.5.3 Precise Distribution Support on Fixed Counter 0

The Tremont microarchitecture supports the PDIR (Precise Distribution of Retired Instructions) facility, as described in Section 20.3.4.4.4, on Fixed Counter 0. Fixed Counter 0 counts the INST_RETIRED.ALL event. PEBS skid for Fixed Counter 0 will be precisely one instruction.

This is in addition to the reduced skid PEBS behavior on IA32_PMC0; see Section 20.5.3.1.2.

### 20.5.5.4 Compatibility Enhancements to Offcore Response MSRs

The Off-core Response facility is similar to that described in Section 20.5.3.2.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as shown below. RequestType bits are defined in Table 20-74, ResponseType bits in Table 20-75, and SnoopInfo bits in Table 20-76.

#### Table 20-74. MSR_OFFCORE_RSPx Request Type Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMAND_DATA_RD</td>
<td>0</td>
<td>Counts demand data reads.</td>
</tr>
<tr>
<td>DEMAND_RFO</td>
<td>1</td>
<td>Counts all demand reads for ownership (RFO) requests and software based prefetches for exclusive ownership (prefetchw).</td>
</tr>
<tr>
<td>DEMAND_CODE_RD</td>
<td>2</td>
<td>Counts demand instruction fetches and L1 instruction cache prefetches.</td>
</tr>
<tr>
<td>COREWB_M</td>
<td>3</td>
<td>Counts modified write backs from L1 and L2.</td>
</tr>
<tr>
<td>HWPF_L2_DATA_RD</td>
<td>4</td>
<td>Counts prefetch (that bring data to L2) data reads.</td>
</tr>
<tr>
<td>HWPF_L2_RFO</td>
<td>5</td>
<td>Counts all prefetch (that bring data to L2) RFOs.</td>
</tr>
<tr>
<td>HWPF_L2_CODE_RD</td>
<td>6</td>
<td>Counts all prefetch (that bring data to L2 only) code reads.</td>
</tr>
<tr>
<td>Reserved</td>
<td>9:7</td>
<td>Reserved.</td>
</tr>
<tr>
<td>HWPF_L1D_AND_SWPF</td>
<td>10</td>
<td>Counts L1 data cache hardware prefetch requests, read for ownership prefetch requests and software prefetch requests (except prefetchw).</td>
</tr>
<tr>
<td>STREAMING_WR</td>
<td>11</td>
<td>Counts all streaming stores.</td>
</tr>
<tr>
<td>COREWB_NONM</td>
<td>12</td>
<td>Counts non-modified write backs from L2.</td>
</tr>
<tr>
<td>Reserved</td>
<td>14:13</td>
<td>Reserved.</td>
</tr>
<tr>
<td>OTHER</td>
<td>15</td>
<td>Counts miscellaneous requests, such as I/O accesses that have any response type.</td>
</tr>
<tr>
<td>UC_RD</td>
<td>44</td>
<td>Counts uncached memory reads (PRd, UCRdF).</td>
</tr>
<tr>
<td>UC_WR</td>
<td>45</td>
<td>Counts uncached memory writes (WcL).</td>
</tr>
<tr>
<td>PARTIAL_STREAMING_WR</td>
<td>46</td>
<td>Counts partial (less than 64 byte) streaming stores (WCiL).</td>
</tr>
<tr>
<td>FULL_STREAMING_WR</td>
<td>47</td>
<td>Counts full, 64 byte streaming stores (WCiLF).</td>
</tr>
</tbody>
</table>
Table 20-74. MSR_OFFCORE_RSPx Request Type Definition (Contd.)

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1WB_M</td>
<td>48</td>
<td>Counts modified WriteBacks from L1 that miss the L2.</td>
</tr>
<tr>
<td>L2WB_M</td>
<td>49</td>
<td>Counts modified WriteBacks from L2.</td>
</tr>
</tbody>
</table>

Table 20-75. MSR_OFFCORE_RSPx Response Type Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_RESPONSE</td>
<td>16</td>
<td>Catch all value for any response types.</td>
</tr>
<tr>
<td>L3_HIT_M</td>
<td>18</td>
<td>LLC/L3 Hit - M-state.</td>
</tr>
<tr>
<td>L3_HIT_E</td>
<td>19</td>
<td>LLC/L3 Hit - E-state.</td>
</tr>
<tr>
<td>L3_HIT_S</td>
<td>20</td>
<td>LLC/L3 Hit - S-state.</td>
</tr>
<tr>
<td>L3_HIT_F</td>
<td>21</td>
<td>LLC/L3 Hit - I-state.</td>
</tr>
<tr>
<td>LOCAL_DRAM</td>
<td>26</td>
<td>LLC/L3 Miss, DRAM Hit.</td>
</tr>
<tr>
<td>OUTSTANDING</td>
<td>63</td>
<td>Average latency of outstanding requests with the other counter counting number of occurrences; can also be used to count occupancy.</td>
</tr>
</tbody>
</table>

Table 20-76. MSR_OFFCORE_RSPx Snoop Info Definition

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNOOP_NONE</td>
<td>31</td>
<td>None of the cores were snooped.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC miss and Dram data returned directly to the core.</td>
</tr>
<tr>
<td>SNOOP_NOT_NEEDED</td>
<td>32</td>
<td>No snoop needed to satisfy the request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC hit and CV bit(s) (core valid) was not set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC miss and Dram data returned directly to the core.</td>
</tr>
<tr>
<td>SNOOP_MISS</td>
<td>33</td>
<td>A snoop was sent but missed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC hit and CV bit(s) was set but snoop missed (silent data drop in core), data returned from LLC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC miss and Dram data returned directly to the core.</td>
</tr>
<tr>
<td>SNOOP_HIT_NO_FWD</td>
<td>34</td>
<td>A snoop was sent but no data forward.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC hit and CV bit(s) was set but no data forward from the core, data returned from LLC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC miss and Dram data returned directly to the core.</td>
</tr>
<tr>
<td>SNOOP_HIT_WITH_FWD</td>
<td>35</td>
<td>A snoop was sent and non-modified data was forward.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC hit and CV bit(s) was set, non-modified data was forward from core.</td>
</tr>
<tr>
<td>SNOOP_HITM</td>
<td>36</td>
<td>A snoop was sent and modified data was forward.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC hit E or M and the CV bit(s) was set, modified data was forward from core.</td>
</tr>
<tr>
<td>NON_DRAM_BIT</td>
<td>37</td>
<td>Target was non-DRAM system address, MMIO access.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LLC miss and Non-Dram data returned.</td>
</tr>
</tbody>
</table>
The Off-core Response capability behaves as follows:

- To specify a complete offcore response filter, software must properly program at least one RequestType and one ResponseType. A valid request type must have at least one bit set in the non-reserved bits of 15:0 or 49:44. A valid response type must be a non-zero value of one the following expressions:
  - **Read requests:**
    - Any_Response Bit | ('OR’ of Supplier Info Bits) ’AND’ ( ‘OR’ of Snoop Info Bits) | Outstanding Bit
  - **Write requests:**
    - Any_Response Bit | ('OR’ of Supplier Info Bits) | Outstanding Bit

- When the ANY_RESPONSE bit in the ResponseType is set, all other response type bits will be ignored.
- True Demand Cacheable Loads include neither L1 Prefetches nor Software Prefetches.
- Bits 15:0 and Bits 49:44 specifies the request type of a transaction request to the uncore. This is described in Table 20-74.
- Bits 30:16 specifies common supplier information.
- “Outstanding Requests” (bit 63) is only available on MSR_OFFCORE_RSP0; a #GP fault will occur if software attempts to write a 1 to this bit in MSR_OFFCORE_RSP1. It is mutually exclusive with any ResponseType. Software must guarantee that all other ResponseType bits are set to 0 when the “Outstanding Requests” bit is set.
- “Outstanding Requests” bit 63 can enable measurement of the average latency of a specific type of off-core transaction; two programmable counters must be used simultaneously and the RequestType programming for MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 must be the same when using this Average Latency feature. See Section 20.5.2.3 for further details.

### 20.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS)

#### 20.6.1 Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 20-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 9.6, “Detecting Hardware Multi-Threading Support and Topology,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). As a result, the unit mask field (for example, IA32_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 20-77. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see: [https://perfmon-events.intel.com/](https://perfmon-events.intel.com/)) and for Intel Core Duo processors. Such events are referred to as core-specific events.

<table>
<thead>
<tr>
<th>IA32_PERFEVTSELx MSRs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 15:14 Encoding</td>
<td></td>
</tr>
<tr>
<td>11B</td>
<td>All cores</td>
</tr>
<tr>
<td>10B</td>
<td>Reserved</td>
</tr>
<tr>
<td>01B</td>
<td>This core</td>
</tr>
<tr>
<td>00B</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 20-78.

**Table 20-78. Agent Specificity Encoding within a Non-Architectural Umask**

<table>
<thead>
<tr>
<th>IA32_PERFEVTSELx MSRs</th>
<th>Bit 13 Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>This agent</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Include all agents</td>
</tr>
</tbody>
</table>

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 20-79.

**Table 20-79. HW Prefetch Qualification Encoding within a Non-Architectural Umask**

<table>
<thead>
<tr>
<th>IA32_PERFEVTSELx MSRs</th>
<th>Bit 13:12 Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11B</td>
<td>All inclusive</td>
</tr>
<tr>
<td></td>
<td>10B</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>01B</td>
<td>Hardware prefetch only</td>
</tr>
<tr>
<td></td>
<td>00B</td>
<td>Exclude hardware prefetch</td>
</tr>
</tbody>
</table>

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 20-79.

**Table 20-80. MESI Qualification Definitions within a Non-Architectural Umask**

<table>
<thead>
<tr>
<th>IA32_PERFEVTSELx MSRs</th>
<th>Bit Position 11:8</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bit 11</td>
<td>Counts modified state</td>
</tr>
<tr>
<td></td>
<td>Bit 10</td>
<td>Counts exclusive state</td>
</tr>
<tr>
<td></td>
<td>Bit 9</td>
<td>Counts shared state</td>
</tr>
<tr>
<td></td>
<td>Bit 8</td>
<td>Counts invalid state</td>
</tr>
</tbody>
</table>

**20.6.2 Performance Monitoring (Processors Based on Intel Core™ Microarchitecture)**

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32_PERFEVTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 20.6.2.1). IA32_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 20-1. Starting with Intel Core 2
processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 20.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits can be found at: https://perfmon-events.intel.com/. The UMASK field may contain sub-fields identical to those listed in Table 20-77, Table 20-78, Table 20-79, and Table 20-80. One or more of these sub-fields may apply to specific events on an event-by-event basis.

In addition, the UMASK field may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 20-81.

**NOTE**
Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

### 20.6.2.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural Perfmon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. See Table 20-2 for details of the PMC addresses and what these events count.

Programming the fixed-function performance counters does not involve any of the IA32_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR IA32_FIXED_CTR_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 20-43. Two sub-fields are defined for each control. See Figure 20-43; bit fields are:

- **Enable field (low 2 bits in each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

<table>
<thead>
<tr>
<th>IA32_PERFEVTSELx MSRs</th>
<th>Bit Position 11:8</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 11</td>
<td>HITM response</td>
<td></td>
</tr>
<tr>
<td>Bit 10</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Bit 9</td>
<td>HIT response</td>
<td></td>
</tr>
<tr>
<td>Bit 8</td>
<td>CLEAN response</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IA32_PERFEVTSELx MSRs</th>
<th>Bit Position 9:8</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 9</td>
<td>CMP2I snoops</td>
<td></td>
</tr>
<tr>
<td>Bit 8</td>
<td>CMP2S snoops</td>
<td></td>
</tr>
</tbody>
</table>

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.
When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

**PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

### 20.6.2.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e., enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSRs:

- **MSR_PERF_GLOBAL_CTRL** enables/disables event counting for all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
- **MSR_PERF_GLOBAL_STATUS** allows software to query counter overflow conditions on any combination of fixed-function PMCs (IA32 FIXED_CTRx) or general-purpose PMCs via a single RDMSR.
- **MSR_PERF_GLOBAL_OVF_CTRL** allows software to clear counter overflow conditions on any combination of fixed-function PMCs (IA32 FIXED_CTRx) or general-purpose PMCs via a single WRMSR.

MSR_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 20-44). Each enable bit in MSR_PERF_GLOBAL_CTRL is AND’ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND’ed results is true; counting is disabled when the result is false.
PERFORMANCE MONITORING

MSR_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 20-45). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 18.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

MSR_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 20-46). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.
20.6.2.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst® microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core microarchitecture do not require special MSR programming control (see Section 20.6.3.6, “At-Retirement Counting”), but is limited to IA32_PMC0. See Table 20-83 for a list of events available to processors based on Intel Core microarchitecture.

### Table 20-83. At-Retirement Performance Events for Intel Core Microarchitecture

<table>
<thead>
<tr>
<th>Event Name</th>
<th>UMask</th>
<th>Event Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITLB_MISS_RETIRED</td>
<td>00H</td>
<td>C9H</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L1D_MISS</td>
<td>01H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L1D_LINE_MISS</td>
<td>02H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L2_MISS</td>
<td>04H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L2_LINE_MISS</td>
<td>08H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.DTLB_MISS</td>
<td>10H</td>
<td>CBH</td>
</tr>
</tbody>
</table>

20.6.2.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (see Section 20.6.2.4.2 and Section 18.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 20-84. The procedure for detecting availability of PEBS is the same as described in Section 20.6.3.8.1.
20.6.2.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 18-8 to set up the precise event records buffer in memory.

2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.

3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 20-84.

20.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITIES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version ID equals 2 or higher. The bit fields of IA32_PERF_CAPABILITIES are defined in Table 2-2 of Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The relevant bit fields that governs PEBS are:

- **PEBSTrap [bit 6]:** When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.

- **PEBSSaveArchRegs [bit 7]:** When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1.

- **PEBSRecordFormat [bits 11:8]:** Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (See Section 20.6.3.8).
  - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (See Section 20.3.1.1.1).
  - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (See Section 20.3.6.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32_PERF_GLOBAL_STATUS at offset 90H. (See Section 20.8.1.1).
  - 0100B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 20.9.2.3). The PEBS Configuration Buffer is defined as shown in Figure 20-64 with Counter Reset fields allocation for 8 general-purpose counters followed by 4 fixed-function counters.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>UMask</th>
<th>Event Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED.ANY_P</td>
<td>00H</td>
<td>C0H</td>
</tr>
<tr>
<td>X87_OPS_RETIRED.ANY</td>
<td>FEH</td>
<td>C1H</td>
</tr>
<tr>
<td>BR_INST_RETIRED.MISPRED</td>
<td>00H</td>
<td>C5H</td>
</tr>
<tr>
<td>SIMD_INST_RETIRED.ANY</td>
<td>01H</td>
<td>C7H</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L1D_MISS</td>
<td>02H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L2D_LINE_MISS</td>
<td>04H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.L2D_MISS</td>
<td>08H</td>
<td>CBH</td>
</tr>
<tr>
<td>MEM_LOAD_RETIRED.DTLB_MISS</td>
<td>10H</td>
<td>CBH</td>
</tr>
</tbody>
</table>

Table 20-84. PEBS Performance Events for Intel Core Microarchitecture
0101B: PEBS record contents are defined by elections in MSR_PEBS_DATA_CFG. (See Section 20.9.2.3). The PEBS Configuration Buffer is defined as shown in Figure 20-64 with Counter Reset fields allocation for 32 general-purpose counters followed by 16 fixed-function counters.

### 20.6.2.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 18.4.9.1, “64 Bit Format of the DS Save Area,” for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 20-85.

### Table 20-85. Requirements to Program PEBS

<table>
<thead>
<tr>
<th></th>
<th>For Processors based on Intel Core microarchitecture</th>
<th>For Processors based on Intel NetBurst microarchitecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verify PEBS support of processor/OS.</td>
<td>• IA32_MISC_ENABLE.EMON_AVAILABLE (bit 7) is set. • IA32_MISC_ENABLE.PEBS_UNAVAILABLE (bit 12) is clear.</td>
<td>Optional</td>
</tr>
<tr>
<td>Ensure counters are in disabled.</td>
<td>On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: • Clear all counters if “Counter Freeze on PMI” is not enabled. • If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.</td>
<td>Optional</td>
</tr>
<tr>
<td>Disable PEBS.</td>
<td>Clear ENABLE PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).</td>
<td>Optional</td>
</tr>
<tr>
<td>Check overflow conditions.</td>
<td>Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.</td>
<td>Check OVF flag of each CCCR for overflow condition</td>
</tr>
<tr>
<td>Clear overflow status.</td>
<td>Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).</td>
<td>Clear OVF flag of each CCCR.</td>
</tr>
<tr>
<td>Write “sample-after” values.</td>
<td>Configure the counter(s) with the sample after value.</td>
<td></td>
</tr>
<tr>
<td>Configure specific counter configuration MSR.</td>
<td>• Set local enable bit 22 - 1. • Do NOT set local counter PMI/INT bit, bit 20 - 0. • Event programmed must be PEBS capable.</td>
<td>• Set appropriate OVF_PMI bits - 1. • Only CCCR for MSR_IQ_COUNTER4 support PEBS.</td>
</tr>
<tr>
<td>Allocate buffer for PEBS states.</td>
<td>Allocate a buffer in memory for the precise information.</td>
<td></td>
</tr>
<tr>
<td>Program the IA32_DS_AREA MSR.</td>
<td>Program the IA32_DS_AREA MSR.</td>
<td></td>
</tr>
<tr>
<td>Configure the PEBS buffer management records.</td>
<td>Configure the PEBS buffer management records in the DS buffer management area.</td>
<td></td>
</tr>
<tr>
<td>Configure/Enable PEBS.</td>
<td>Set Enable PMCO bit in IA32_PEBS_ENABLE MSR (3F1H).</td>
<td>Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_PEBS_MATRIX_HORZ as needed.</td>
</tr>
<tr>
<td>Enable counters.</td>
<td>Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).</td>
<td>Set each CCCR enable bit 12 - 1.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.
20.6.2.4.4 Re-configuring PEBS Facilities

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32_DS_AREA).

20.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32_MISC_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCR sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32_DS_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR_PEBS_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 20-86 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events can be found at: https://perfmonevents.intel.com/.

<table>
<thead>
<tr>
<th>Counter Name</th>
<th>CCCR Name</th>
<th>ESCR Name</th>
<th>No.</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_BPU_COUNTER0</td>
<td>MSR_BPU_CCCR0</td>
<td>MSR_BPU_ESCR0</td>
<td>0</td>
<td>300H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER1</td>
<td>MSR_BPU_CCCR1</td>
<td>MSR_BPU_ESCR1</td>
<td>1</td>
<td>302H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER2</td>
<td>MSR_BPU_CCCR2</td>
<td>MSR_BPU_ESCR2</td>
<td>2</td>
<td>304H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER3</td>
<td>MSR_BPU_CCCR3</td>
<td>MSR_BPU_ESCR3</td>
<td>3</td>
<td>306H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER4</td>
<td>MSR_BPU_CCCR4</td>
<td>MSR_BPU_ESCR4</td>
<td>4</td>
<td>308H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER5</td>
<td>MSR_BPU_CCCR5</td>
<td>MSR_BPU_ESCR5</td>
<td>5</td>
<td>30AH</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER6</td>
<td>MSR_BPU_CCCR6</td>
<td>MSR_BPU_ESCR6</td>
<td>6</td>
<td>30CH</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER7</td>
<td>MSR_BPU_CCCR7</td>
<td>MSR_BPU_ESCR7</td>
<td>7</td>
<td>302H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER8</td>
<td>MSR_BPU_CCCR8</td>
<td>MSR_BPU_ESCR8</td>
<td>8</td>
<td>304H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER9</td>
<td>MSR_BPU_CCCR9</td>
<td>MSR_BPU_ESCR9</td>
<td>9</td>
<td>306H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER10</td>
<td>MSR_BPU_CCCR10</td>
<td>MSR_BPU_ESCR10</td>
<td>10</td>
<td>308H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER11</td>
<td>MSR_BPU_CCCR11</td>
<td>MSR_BPU_ESCR11</td>
<td>11</td>
<td>30AH</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER12</td>
<td>MSR_BPU_CCCR12</td>
<td>MSR_BPU_ESCR12</td>
<td>12</td>
<td>30CH</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER13</td>
<td>MSR_BPU_CCCR13</td>
<td>MSR_BPU_ESCR13</td>
<td>13</td>
<td>302H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER14</td>
<td>MSR_BPU_CCCR14</td>
<td>MSR_BPU_ESCR14</td>
<td>14</td>
<td>304H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER15</td>
<td>MSR_BPU_CCCR15</td>
<td>MSR_BPU_ESCR15</td>
<td>15</td>
<td>306H</td>
</tr>
</tbody>
</table>

Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)
Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs
(Processors Based on Intel NetBurst Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Counter</th>
<th>No.</th>
<th>Addr</th>
<th>Name</th>
<th>No.</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_BPU_COUNTER1</td>
<td>1</td>
<td>301H</td>
<td>MSR_BPU_CCCR1</td>
<td>361H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_BSU_ESCR0</td>
<td>7</td>
<td>3A0H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FSB_ESCR0</td>
<td>6</td>
<td>3A2H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MOB_ESCR0</td>
<td>2</td>
<td>3AAH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_PMH_ESCR0</td>
<td>4</td>
<td>3AC1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_BPU_ESCR0</td>
<td>0</td>
<td>3B2H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_IS_ESCR0</td>
<td>1</td>
<td>3B4H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_ITLB_ESCR0</td>
<td>3</td>
<td>3B6H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_IX_ESCR0</td>
<td>5</td>
<td>3C8H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER2</td>
<td>2</td>
<td>302H</td>
<td>MSR_BPU_CCCR2</td>
<td>362H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_BSU_ESCR1</td>
<td>7</td>
<td>3A1H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FSB_ESCR1</td>
<td>6</td>
<td>3A3H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MOB_ESCR1</td>
<td>2</td>
<td>3A5H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_PMH_ESCR1</td>
<td>4</td>
<td>3AD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_BPU_ESCR1</td>
<td>0</td>
<td>3B5H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_IS_ESCR1</td>
<td>1</td>
<td>3B7H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_ITLB_ESCR1</td>
<td>3</td>
<td>3C9H</td>
</tr>
<tr>
<td>MSR_BPU_COUNTER3</td>
<td>3</td>
<td>303H</td>
<td>MSR_BPU_CCCR3</td>
<td>363H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_BSU_ESCR1</td>
<td>7</td>
<td>3A1H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FSB_ESCR1</td>
<td>6</td>
<td>3A3H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MOB_ESCR1</td>
<td>2</td>
<td>3A5H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_PMH_ESCR1</td>
<td>4</td>
<td>3AD1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_BPU_ESCR1</td>
<td>0</td>
<td>3B5H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_IS_ESCR1</td>
<td>1</td>
<td>3B7H</td>
</tr>
<tr>
<td>MSR_MS_COUNTER0</td>
<td>4</td>
<td>304H</td>
<td>MSR_MS_CCCR0</td>
<td>364H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MS_ESCR0</td>
<td>0</td>
<td>3C0H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TBPU_ESCR0</td>
<td>2</td>
<td>3C2H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TC_ESCR0</td>
<td>1</td>
<td>3C4H</td>
</tr>
<tr>
<td>MSR_MS_COUNTER1</td>
<td>5</td>
<td>305H</td>
<td>MSR_MS_CCCR1</td>
<td>365H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MS_ESCR0</td>
<td>0</td>
<td>3C0H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TBPU_ESCR0</td>
<td>2</td>
<td>3C2H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TC_ESCR0</td>
<td>1</td>
<td>3C4H</td>
</tr>
<tr>
<td>MSR_MS_COUNTER2</td>
<td>6</td>
<td>306H</td>
<td>MSR_MS_CCCR2</td>
<td>366H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MS_ESCR1</td>
<td>0</td>
<td>3C1H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TBPU_ESCR1</td>
<td>1</td>
<td>3C3H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TC_ESCR1</td>
<td>1</td>
<td>3C5H</td>
</tr>
<tr>
<td>MSR_MS_COUNTER3</td>
<td>7</td>
<td>307H</td>
<td>MSR_MS_CCCR3</td>
<td>367H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_MS_ESCR1</td>
<td>0</td>
<td>3C1H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TBPU_ESCR1</td>
<td>1</td>
<td>3C3H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_TC_ESCR1</td>
<td>1</td>
<td>3C5H</td>
</tr>
<tr>
<td>MSR_FLAME_COUNTER0</td>
<td>8</td>
<td>308H</td>
<td>MSR_FLAME_CCCR0</td>
<td>368H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FIRM_ESCR0</td>
<td>1</td>
<td>3A4H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FLAME_ESCR0</td>
<td>0</td>
<td>3A6H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_DAC_ESCR0</td>
<td>5</td>
<td>3A8H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_SAAT_ESCR0</td>
<td>2</td>
<td>3AEH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_U2L_ESCR0</td>
<td>3</td>
<td>3B0H</td>
</tr>
<tr>
<td>MSR_FLAME_COUNTER1</td>
<td>9</td>
<td>309H</td>
<td>MSR_FLAME_CCCR1</td>
<td>369H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FIRM_ESCR0</td>
<td>1</td>
<td>3A4H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FLAME_ESCR0</td>
<td>0</td>
<td>3A6H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_DAC_ESCR0</td>
<td>5</td>
<td>3A8H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_SAAT_ESCR0</td>
<td>2</td>
<td>3AEH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_U2L_ESCR0</td>
<td>3</td>
<td>3B0H</td>
</tr>
<tr>
<td>MSR_FLAME_COUNTER2</td>
<td>10</td>
<td>30AH</td>
<td>MSR_FLAME_CCCR2</td>
<td>36AH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FIRM_ESCR1</td>
<td>1</td>
<td>3A5H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FLAME_ESCR1</td>
<td>0</td>
<td>3A7H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_DAC_ESCR1</td>
<td>5</td>
<td>3A9H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_SAAT_ESCR1</td>
<td>2</td>
<td>3AFH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_U2L_ESCR1</td>
<td>3</td>
<td>3B1H</td>
</tr>
<tr>
<td>MSR_FLAME_COUNTER3</td>
<td>11</td>
<td>30BH</td>
<td>MSR_FLAME_CCCR3</td>
<td>36BH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FIRM_ESCR1</td>
<td>1</td>
<td>3A5H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_FLAME_ESCR1</td>
<td>0</td>
<td>3A7H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_DAC_ESCR1</td>
<td>5</td>
<td>3A9H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_SAAT_ESCR1</td>
<td>2</td>
<td>3AFH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSR_U2L_ESCR1</td>
<td>3</td>
<td>3B1H</td>
</tr>
</tbody>
</table>
The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- **Non-retirement events** are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- **At-retirement events** are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

The at-retirement counting mechanism includes facilities for tagging μops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

### Table 20-86. Performance Counter MSRs and Associated CCCR and ESCR MSRs
(Processors Based on Intel NetBurst Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Counter</th>
<th>No.</th>
<th>Addr</th>
<th>CCCR</th>
<th>Name</th>
<th>No.</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_IQ_COUNTER0</td>
<td>12</td>
<td>30CH</td>
<td>MSR_IQ_CCCR0</td>
<td>36CH</td>
<td>MSR_CRU_ESCR0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_IQ_ESCR0^1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_RAT_ESCR0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_SSU_ESCR0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_ALF_ESCR0</td>
<td>1</td>
</tr>
<tr>
<td>MSR_IQ_COUNTER1</td>
<td>13</td>
<td>30DH</td>
<td>MSR_IQ_CCCR1</td>
<td>36DH</td>
<td>MSR_CRU_ESCR0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_IQ_ESCR0^1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_RAT_ESCR0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_SSU_ESCR0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_ALF_ESCR0</td>
<td>1</td>
</tr>
<tr>
<td>MSR_IQ_COUNTER2</td>
<td>14</td>
<td>30EH</td>
<td>MSR_IQ_CCCR2</td>
<td>36EH</td>
<td>MSR_CRU_ESCR1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_IQ_ESCR1^1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_RAT_ESCR1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_ALF_ESCR1</td>
<td>1</td>
</tr>
<tr>
<td>MSR_IQ_COUNTER3</td>
<td>15</td>
<td>30FH</td>
<td>MSR_IQ_CCCR3</td>
<td>36FH</td>
<td>MSR_CRU_ESCR1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_IQ_ESCR1^1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_RAT_ESCR1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_ALF_ESCR1</td>
<td>1</td>
</tr>
<tr>
<td>MSR_IQ_COUNTER4</td>
<td>16</td>
<td>310H</td>
<td>MSR_IQ_CCCR4</td>
<td>370H</td>
<td>MSR_CRU_ESCR0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_IQ_ESCR0^1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_RAT_ESCR0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_SSU_ESCR0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_ALF_ESCR0</td>
<td>1</td>
</tr>
<tr>
<td>MSR_IQ_COUNTER5</td>
<td>17</td>
<td>311H</td>
<td>MSR_IQ_CCCR5</td>
<td>371H</td>
<td>MSR_CRU_ESCR1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_CRU_ESCR5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_IQ_ESCR1^1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_RAT_ESCR1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MSR_ALF_ESCR1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOTES:**
1. MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- **Non-retirement events** are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- **At-retirement events** are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

The at-retirement counting mechanism includes facilities for tagging μops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).
The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting** — A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.

- **Interrupt-based event sampling** — A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

  When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.

- **Processor event-based sampling (PEBS)** — In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

### 20.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 20-86) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 20-86) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 20-47 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- **USR flag, bit 2** — When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.

- **OS flag, bit 3** — When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)

- **Tag enable, bit 4** — When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 20.6.3.6, “At-Retirement Counting.”

---

**Figure 20-47. Event Selection Control Register (ESCR) for Pentium 4 and Intel® Xeon® Processors without Intel HT Technology Support**
• **Tag value field, bits 5 through 8** — Selects a tag value to associate with a µop to assist in at-retirement event counting.

• **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.

• **Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 20-86 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

### 20.6.3.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR’s (see Table 20-86). The counter pairs are partitioned into four groups:

- **The BPU group**, includes two performance counter pairs:
  - MSR_BPU_COUNTER0 and MSR_BPU_COUNTER1.
  - MSR_BPU_COUNTER2 and MSR_BPU_COUNTER3.

- **The MS group**, includes two performance counter pairs:
  - MSR_MS_COUNTER0 and MSR_MS_COUNTER1.
  - MSR_MS_COUNTER2 and MSR_MS_COUNTER3.

- **The FLAME group**, includes two performance counter pairs:
  - MSR_FLAME_COUNTER0 and MSR_FLAME_COUNTER1.
  - MSR_FLAME_COUNTER2 and MSR_FLAME_COUNTER3.

- **The IQ group**, includes three performance counter pairs:
  - MSR_IQ_COUNTER0 and MSR_IQ_COUNTER1.
  - MSR_IQ_COUNTER2 and MSR_IQ_COUNTER3.
  - MSR_IQ_COUNTER4 and MSR_IQ_COUNTER5.

The MSR_IQ_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR_BPU_COUNTER0 can start MSR_BPU_COUNTER2 and vice versa, and MSR_BPU_COUNTER1 can start MSR_BPU_COUNTER3 and vice versa (see Section 20.6.3.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 20-48). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or, if ECX[31] is set to 1, the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits. In such cases, counter bits 31:0 are written to EAX, while 0 is written to EDX.
The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

20.6.3.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 20-86). The CCCRMs control the filtering and counting of events as well as interrupt generation. Figure 20-49 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 20.6.3.5.2, “Filtering Events”). The complement flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 20.6.3.5.2, “Filtering Events”).
- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
• **FORCE_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.

• **OVF_PMI flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.

• **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 20.6.3.2, “Performance Counters,” for further details); when clear, disables cascading of counters.

• **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.

2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.

3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.

4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.

5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one “stage” forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCRs are discussed in greater detail in Section 20.6.3.5, “Programming the Performance Counters for Non-Retirement Events.”
20.6.3.4  Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 18.4.5, “Branch Trace Store (BTS),” and Section 20.6.3.8, “Processor Event-Based Sampling (PEBS),” for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 18.4.9, “BTS and DS Save Area.”

20.6.3.5  Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event.
3. Match the CCCR Select value and ESCR name to a value listed in Table 20-86; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which they are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

20.6.3.5.1  Selecting Events to Count

There is a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event, setup information is provided. Table 20-87 gives an example of one of the events.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Parameters</th>
<th>Parameter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_retired</td>
<td></td>
<td></td>
<td>Counts the retirement of a branch. Specify one or more mask bits to select</td>
</tr>
<tr>
<td></td>
<td>ESCR restrictions</td>
<td>MSR_CRU_ESCR2, MSR_CRU_ESCR3</td>
<td>any combination of branch taken, not-taken, predicted, and mispredicted.</td>
</tr>
<tr>
<td></td>
<td>Counter numbers</td>
<td>ESCR2: 12, 13, 16, ESCR3:</td>
<td>The counter numbers associated with each ESCR are provided. The performance</td>
</tr>
<tr>
<td></td>
<td>per ESCR</td>
<td>14, 15, 17</td>
<td>counters and corresponding CCCR are obtained from Table 15-3.</td>
</tr>
<tr>
<td></td>
<td>ESCR Event Select</td>
<td>06H</td>
<td>ESCR[31:25]</td>
</tr>
<tr>
<td></td>
<td>ESCR Event Mask</td>
<td>Bit 0: MMNP, 1: MMNM, 2:</td>
<td>Branch Not-taken Predicted, Branch Not-taken Mispredicted, Branch Taken</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: MMTM</td>
<td>Predicted, Branch Taken Mispredicted</td>
</tr>
<tr>
<td></td>
<td>CCCR Select</td>
<td>05H</td>
<td>CCCR[15:13]</td>
</tr>
</tbody>
</table>
Event Parameters are described below.

- **ESCR restrictions** — Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.

- **Counter numbers per ESCR** — Lists which performance counters are associated with each ESCR. Table 20-86 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.

- **ESCR event select** — Gives the value to be placed in the event select field of the ESCR to select the event.

- **ESCR event mask** — Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.

- **CCCR select** — Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 20-86.

- **Event specific notes** — Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.

- **Can support PEBS** — Indicates if PEBS is supported for the event (only supplied for at-retirement events).

- **Requires additional MSRs for tagging** — Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events).

**NOTE**

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

An event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.
3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 20-86.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.
6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.
NOTE

Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 20.6.3.5.2, “Filtering Events.”

20.6.3.5.2 Filtering Events

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 3 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counters threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a “greater than” or a “less than or equal to” comparison of the input value vs. a threshold value can be made. The complement flag selects “less than or equal to” (flag set) or “greater than” (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, than any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a “rising edge” event; that is, a false-to-true transition. Figure 20-50 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 20.6.3.5.1, “Selecting Events to Count.”

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
   — Set the compare flag.
   — Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
   — Enter a value from 0 to 15 in the threshold field.

8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 20.6.3.5.3, “Starting Event Counting.”

![Figure 20-50. Effects of Edge Filtering](image-url)
20.6.3.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter’s CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 20.6.3.5.5, “Halting Event Counting”).

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 20.6.3.5.2, “Filtering Events.”

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 20.6.3.5.4, “Reading a Performance Counter’s Count.”

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 20.6.3.5.6, “Cascading Counters”).

20.6.3.5.4 Reading a Performance Counter’s Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 20.6.3.2, “Performance Counters.” These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 20.6.3.5.3, “Starting Event Counting.”

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 20-86 used as an operand.

This setup procedure is continued in the next section, Section 20.6.3.5.5, “Halting Event Counting.”

20.6.3.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 20.6.3.5.4, “Reading a Performance Counter’s Count.”

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter’s CCCR MSR or clear the OVF flag in the alternate counter’s CCCR MSR.

20.6.3.5.6 Cascading Counters

As described in Section 20.6.3.2, “Performance Counters,” eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 20-86). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR_IQ_CCCR0 through MSR_IQ_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 20-49 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR_MO B_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It’s possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14
cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

Example 20-1. Counting Events

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 20.6.3.5.1, "Selecting Events to Count." Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 20.6.3.5.8, "Generating an Interrupt on Overflow."

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

20.6.3.5.7 EXTENDED CASCADING

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily_DisplayModel 0F_02, 0F_03, 0F_04, 0F_06. This feature uses bit 11 in CCRs associated with the IQ block. See Table 20-88.

Table 20-88. CCR Names and Bit Positions

<table>
<thead>
<tr>
<th>CCCR Name:Bit Position</th>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_IQ_CCCR1:2:11</td>
<td>CASCNT4INTO0</td>
<td>Allow counter 4 to cascade into counter 0</td>
</tr>
<tr>
<td>MSR_IQ_CCCR0:11</td>
<td>CASCNT5INTO3</td>
<td>Allow counter 5 to cascade into counter 3</td>
</tr>
<tr>
<td>MSR_IQ_CCCR3:11</td>
<td>CASCNT5INTO4</td>
<td>Allow counter 5 to cascade into counter 4</td>
</tr>
<tr>
<td>MSR_IQ_CCCR4:11</td>
<td>CASCNT4INTO5</td>
<td>Allow counter 4 to cascade into counter 5</td>
</tr>
</tbody>
</table>

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily_DisplayModel signature of 0F_02. For processors with CPUID DisplayFamily_DisplayModel signature of 0F_00 and 0F_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINTOy bits is used.

Example 20-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:
1. Write the value 0 to counter 12.

2. Write the value 04000603H to MSR_CRU_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGNTAG event masks with qualification restricted to logical processor 1).

3. Write the value 04038800H to MSR_IQ_CCCR0. This enables CASCNT4INTO0 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).

4. Write the value FFFFFFFFH into counter 16.1.

5. Write the value 0400060CH to MSR_CRU_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGNTAG event masks with qualification restricted to logical processor 0).

6. Write the value 00039000H to MSR_IQ_CCCR4 (set ENABLE bit, but not OVF_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

20.6.3.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR_PMI flag in the counter’s associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 20.6.3.5.6, “Cascading Counters.” Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

20.6.3.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no_event, which generally has a select value of 0).
20.6.3.6  At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called “at-retirement counting.”

There are predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire** — In at-retirement event descriptions, the term “bogus” refers to instructions or \(\mu\)ops that must be canceled because they are on a path taken from a mispredicted branch. The terms “retired” and “non-bogus” refer to instructions or \(\mu\)ops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and \(\mu\)ops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors’ performance monitoring events (such as, Instruction_Retired and Uops_Retired) can count instructions or \(\mu\)ops that are retired based on the characterization of bogus” versus non-bogus.

- **Tagging** — Tagging is a means of marking \(\mu\)ops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per \(\mu\)op and a direct count of the event would not provide an indication of how many \(\mu\)ops encountered that event. The tagging mechanisms allow a \(\mu\)op to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per \(\mu\)op, rather than once per event. For example, a \(\mu\)op may encounter a cache miss more than once during its life time, but a “Miss Retired” metric (that counts the number of retired \(\mu\)ops that encountered a cache miss) will increment only once for that \(\mu\)op. A “Miss Retired” metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the Intel® 64 and IA-32 Architectures Optimization Reference Manual (see Section 1.4, “Related Literature”).

- **Replay** — To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules \(\mu\)ops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, \(\mu\)ops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of \(\mu\)ops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.

- **Assist** — When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of \(\mu\)ops before they begin and are costly.

20.6.3.6.1  Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and \(\mu\)ops that encountered a specified event. For a subset of the at-retirement events, a \(\mu\)op may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count \(\mu\)ops tagged with that mechanism:

- **Front-end tagging** — This mechanism pertains to the tagging of \(\mu\)ops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the Front_end_event event.

- **Execution tagging** — This mechanism pertains to the tagging of \(\mu\)ops that encountered execution events (for example, instruction types) and are counted with the Execution_Event event.
• **Replay tagging** — This mechanism pertains to tagging of µops whose retirement is replayed (for example, a cache miss) and are counted with the Replay_event event. Branch mispredictions are also tagged with this mechanism.

• **No tags** — This mechanism does not use tags. It uses the Instr_retired and the Uops_retired events.

Each tagging mechanism is independent from all others; that is, a µop that has been tagged using one mechanism will not be detected with another mechanism’s tagged-µop detector. For example, if µops are tagged using the front-end tagging mechanisms, the Replay_event will not count those as tagged µops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of µops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of µops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

There are performance monitoring events that support at-retirement counting: specifically the Front_end_event, Execution_event, Replay_event, Inst_retired, and Uops_retired events. The following sections describe the tagging mechanisms for using these events to tag µop and count tagged µops.

### 20.6.3.6.2 Tagging Mechanism for Front_end_event

The Front_end_event counts µops that have been tagged as encountering any of the following events:

- **µop decode events** — Tagging µops for µop decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop_type.

- **Trace cache events** — Tagging µops for trace cache events may require specifying certain bits in the MSR_TC_PRECISE_EVENT MSR.

The MSRs that are supported by the front-end tagging mechanism must be set and one or both of the NBOGUS and BOGUS bits in the Front_end_event event mask must be set to count events. None of the events currently supported requires the use of the MSR_TC_PRECISE_EVENT MSR.

### 20.6.3.6.3 Tagging Mechanism For Execution_event

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect µops that have been tagged with that tag value identifier using Execution_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular µop. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged µops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. The tag enable and tag value bits are irrelevant for the downstream ESCR used to select the Execution_event.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 20.6.3.8.3, “Setting Up the PEBS Buffer.”) It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream ESCR or multiple mask bits in the downstream ESCR. For example, use a tag value of 3H in the upstream ESCR and use NBOGUS0/NBOGUS1 in the downstream ESCR event mask.

### 20.6.3.7 Tagging Mechanism for Replay_event

The replay mechanism enables tagging of µops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of µop that may experience the replay in the MSR_PEBs_MATRIX_VERT MSR and selecting the type of event in the MSR_PEBs_ENABLE MSR. Replay tagging must also be enabled with the UOP_Tag flag (bit 24) in the MSR_PEBs_ENABLE MSR.
The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 18.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA_32_PEBS_ENABLE MSR. Each of these metrics requires that the Replay_Event be used to count the tagged μops.

20.6.3.8 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 18.4.5, “Branch Trace Store (BTS),” for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 18.4.9, “BTS and DS Save Area”). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can only be carried out using the one performance counter, the MSR_IQ_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32_PMC0 and IA32_PERFEVTSEL0 MSRs (See Section 20.6.2.4).

20.6.3.8.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS_UNAVAILABLE flag in the IA32_MISC_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR_PEBS_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR_PEBS_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32_DS_AREA MSR can be programmed to point to the DS save area.

20.6.3.8.2 Setting Up the DS Save Area

Section 18.4.9.2, “Setting Up the DS Save Area,” describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

20.6.3.8.3 Setting Up the PEBS Buffer

Only the MSR_IQ_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 18-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR_PEBS_ENABLE MSR.
3. Set up the MSR_IQ_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS.
20.6.3.8.4  Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 18.4.9.5, "Writing the DS Interrupt Service Routine," for guidelines for writing the DS ISR.

20.6.3.8.5  Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT. The DS mechanism is available in real address mode.

20.6.3.9  Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

20.6.4  Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 20.6.3. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_TC_PRECISE_EVENT.

20.6.4.1  ESCR MSRs

Figure 20-51 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1_USR flag, bit 0** — When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
PERFORMANCE MONITORING

• **T1_OS flag, bit 1** — When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1_OS and T1_USR flags are set, thread 1 events are counted at all privilege levels.)

• **T0_USR flag, bit 2** — When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.

• **T0_OS flag, bit 3** — When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0_OS and T0_USR flags are set, thread 0 events are counted at all privilege levels.)

• **Tag enable, bit 4** — When set, enables tagging of \( \mu \)ops to assist in at-retirement event counting; when clear, disables tagging. See Section 20.6.3.6, “At-Retirement Counting.”

• **Tag value field, bits 5 through 8** — Selects a tag value to associate with a \( \mu \)op to assist in at-retirement event counting.

• **Event mask field, bits 9 through 24** — Selects events to be counted from the event class selected with the event select field.

• **Event select field, bits 25 through 30** — Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0_OS and T0_USR flags and the T1_OS and T1_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 9.4.5, “Identifying Logical Processors in an MP System,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 20.6.4.4, “Performance Monitoring Events”). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

### 20.6.4.2 CCCR MSRs

Figure 20-52 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

• **Enable flag, bit 12** — When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset

• **ESCR select field, bits 13 through 15** — Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.

• **Active thread field, bits 16 and 17** — Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:

  **00** — None. Count only when neither logical processor is active.
01 — Single. Count only when one logical processor is active (either 0 or 1).
10 — Both. Count only when both logical processors are active.
11 — Any. Count when either logical processor is active.

A halted logical processor or a logical processor in the “wait for SIPI” state is considered inactive.

- **Compare flag, bit 18** — When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.

![Diagram of Counter Configuration Control Register (CCCR)](image_url)

- **Complement flag, bit 19** — Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 20.6.3.5.2, "Filtering Events"). The compare flag is not active unless the compare flag is set.

- **Threshold field, bits 20 through 23** — Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 20.6.3.5.2, “Filtering Events”).

- **Edge flag, bit 24** — When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.

- **FORCE_OVF flag, bit 25** — When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.

- **OVF_PMI_T0 flag, bit 26** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.

- **OVF_PMI_T1 flag, bit 27** — When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.

- **Cascade flag, bit 30** — When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 20.6.3.2, “Performance Counters,” for further details); when clear, disables cascading of counters.
• **OVF flag, bit 31** — Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

### 20.6.4.3 IA32_PEB_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR_PEB_ENABLE MSR: bit 25 (ENABLE_PEB_MY_THR) and 26 (ENABLE_PEB_OTH_THR) respectively. These bits do not explicitly identify a specific logical processor by logical processor ID (T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running ("my thread") or for the other logical processor in the physical package on which the agent is not running ("other thread").

PEBS is supported for only a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can be carried out only with two performance counters: MSR_IQ_CCCR4 (MSR address 370H) for logical processor 0 and MSR_IQ_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE_PEB_MY_THR and ENABLE_PEB_OTH_THR bits in the MSR_PEB_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

### 20.6.4.4 Performance Monitoring Events

When Intel Hyper-Threading Technology is active, many performance monitoring events can be can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS)** — The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI)** — The event cannot be qualified as being associated with a specific logical processor.

If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 20-89) depends only on the setting of the T0_USR and T0_OS flags in the ESCR being used to set up the event counter. The T1_USR and T1_OS flags have no effect on the count.

<table>
<thead>
<tr>
<th>T0_OS/T0_USR = 00</th>
<th>T0_OS/T0_USR = 01</th>
<th>T0_OS/T0_USR = 11</th>
<th>T0_OS/T0_USR = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0_OS/T0_USR = 00</td>
<td>Zero count</td>
<td>Counts while T1 in OS or USR</td>
<td>Counts while T1 in OS</td>
</tr>
<tr>
<td>T0_OS/T0_USR = 01</td>
<td>Counts while T0 in USR or T1 in USR</td>
<td>Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR</td>
<td>Counts while (a) T0 in OS or (b) T1 in OS</td>
</tr>
<tr>
<td>T0_OS/T0_USR = 11</td>
<td>Counts while T0 in OS or USR</td>
<td>Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR</td>
<td>Counts irrespective of CPL, T0, T1</td>
</tr>
<tr>
<td>T0_OS/T0_USR = 10</td>
<td>Counts T0 in OS</td>
<td>Counts T0 in OS or T1 in USR</td>
<td>Counts while (a) T0 in OS or (b) T1 in OS or (c) T1 in USR</td>
</tr>
</tbody>
</table>

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI, the effect of selectively specifying T0_USR, T0_OS, T1_USR, T1_OS bits is shown in Table 20-90.
20.6.4.5 Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

20.6.4.5.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCR to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the global_power_events and specify the RUNNING sub-event mask and the desired T0_OS/T0_USR/T1_OS/T1_USR bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

20.6.4.5.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than “no_event”; the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to “1”.
4. Set the threshold to “15” and the complement to “1” in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g., by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an HLT instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical processors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

### Table 20-90: Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (T1) Events

<table>
<thead>
<tr>
<th>T0_OS/T0_USR = 00</th>
<th>T0_OS/T0_USR = 01</th>
<th>T0_OS/T0_USR = 11</th>
<th>T0_OS/T0_USR = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1_OS/T1_USR = 00</td>
<td>Zero count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1_OS/T1_USR = 01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1_OS/T1_USR = 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1_OS/T1_USR = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20-124 Vol. 3B
The Non-sleep Clockticks mechanism uses a filtering mechanism in CCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

### 20.6.5 Performance Monitoring and Dual-Core Technology

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but two logical processors in the same core share performance monitoring resources (see Section 20.6.4, "Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture").

### 20.6.6 Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 20.1 and Section 20.6.4) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 20-53.

**Figure 20-53. Block Diagram of 64-bit Intel® Xeon® Processor MP with 8-MByte L3 Cache**

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:
**IBUSQ event** — This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR_IFSB_IBUSQ0 and MSR_IFSB_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 20-54.

![Figure 20-54. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH](image)

**ISNPQ event** — This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR_IFSB_ISNPQ0 and MSR_IFSB_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 20-55.

![Figure 20-55. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH](image)

**EFSB event** — This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR_EFSB_DRDY0 and MSR_EFSB_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 20-56.
• **IBUSQ Latency event** — This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR_IFSB_CTRL6[bit 26] to 1; the count freezes after software sets MSR_IFSB_CTRL6[bit 26] to 0. MSR_IFSB_CNTR7 acts as a 64-bit event counter for this event. See Figure 20-57.

**Figure 20-56. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H**

**Figure 20-57. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H**

### 20.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems

The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily_DisplayModel value of 06_1DH (see the CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through additional control logic. See Figure 20-58 for the block configuration of six processor cores and the L3/Caching bus.
controller sub-system in Intel Xeon processor 7400 series. Figure 20-58 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.

![Figure 20-58. Block Diagram of the Intel® Xeon® Processor 7400 Series](image)

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 20.1 and Section 20.6.4) apply to Intel Xeon processor 7100 series. The MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily_DisplayModel signature 06_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ_allocation and IOQ_active_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.
20.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR_EMON_L3_GL_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3) are dedicated to counting GSNPQ events. Up to two GSNPQ events can be programmed and counted simultaneously.
- Four MSRs (MSR_EMON_L3_CTR_CTL4, MSR_EMON_L3_CTR_CTL5, MSR_EMON_L3_CTR_CTL6, and MSR_EMON_L3_CTR_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.
20.6.7.2 GBSQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1 is given in Figure 20-60. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- **Agent_Select (bits 35:32)**: The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
  
  For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits correspond to two logical processors in the first processor core, the upper two bits correspond to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.
  

- **Data_Flow (bits 37:36)**: Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.

- **Type_Match (bits 43:38)**: Specifies transaction types. If all six bits are set, event count will include all transaction types.

- **Snoop_Match: (bits 46:44)**: The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.

- **L3_State (bits 53:47)**: Each bit specifies an L2 coherency state.

- **Core_Module_Select (bits 55:54)**: The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.
  
  For Intel Xeon processor 7100 series,
  
  - 00B: Match transactions from any core in the physical package
  - 01B: Match transactions from this core only
  - 10B: Match transactions from the other core in the physical package
  - 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series,
  
  - 00B: Match transactions from any dual-core module in the physical package
  - 01B: Match transactions from this dual-core module only
  - 10B: Match transactions from either one of the other two dual-core modules in the physical package
— 11B: Match transaction from more than one dual-core modules in the physical package

- Fill_Eviction (bits 57:56): The valid encodings are
  — 00B: Match any transactions
  — 01B: Match transactions that fill L3
  — 10B: Match transactions that fill L3 without an eviction
  — 11B: Match transaction fill L3 with an eviction

- Cross_Snoop (bit 58): The encodings are
  — 0B: Match any transactions
  — 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

20.6.7.3 GSNPQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3 is given in Figure 20-61. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent_Selec (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
- For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.
- For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.

- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.

- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.


- Core_Module_Select (bits 56:54): Bit 56 enables Core_Module_Select matching. If bit 56 is clear, Core_Module_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.
  - For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
    — 00B: Match transactions from only one core (irrespective which core) in the physical package
    — 01B: Match transactions from this core and not the other core
    — 10B: Match transactions from the other core in the physical package, but not this core
    — 11B: Match transaction from both cores in the physical package
  - For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
    — 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
    — 01B: Match transactions from one or more dual-core modules.
    — 10B: Match transactions from two or more dual-core modules.
    — 11B: Match transaction from all three dual-core modules in the physical package.

- Block_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.
The layout of MSR_EMON_L3_CTR_CTL4 through MSR_EMON_L3_CTR_CTL7 is given in Figure 20-62. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:
- Bit 58: must set to 1.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

**20.6.7.4 FSB Event Interface**

**20.6.7.4.1 FSB Sub-Event Mask Interface**
- FSB_type (bit 37:32): Specifies different FSB transaction types originated from this physical package.
- FSB_L_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package.
- FSB_L_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package.
• FSB_L_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package.
• FSB_L_defer (bit 41): Count DEFER responses to this processor’s transactions.
• FSB_L_retry (bit 42): Count RETRY responses to this processor’s transactions.
• FSB_L_snoop_stall (bit 43): Count snoop stalls to this processor’s transactions.
• FSB_DBYS (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY).
• FSB_DRDY (bit 45): Count DRDY assertions by this processor.
• FSB_BNR (bit 46): Count BNR assertions by this processor.
• FSB_IOQ_empty (bit 47): Counts each bus clocks when the IOQ is empty.
• FSB_IOQ_full (bit 48): Counts each bus clocks when the IOQ is full.
• FSB_IOQ_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ.
• FSB_WW_data (bit 50): Counts back-to-back write transaction’s data phase.
• FSB_WW_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
• FSB_RW_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
• FSB_other_DBYS (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY).
• FSB_other_DRDY (bit 55): Count DRDY assertions by another agent.
• FSB_other_snoop_stall (bit 56): Count snoop stalls on the FSB due to another agent.
• FSB_other_BNR (bit 57): Count BNR assertions from another agent.

20.6.7.5  Common Event Control Interface

The MSR_EMON_L3_GL_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:
• GL_freeze_cmd (bit 0): Freeze the event counters specified by the GL_event_select field.
• GL_unfreeze_cmd (bit 1): Unfreeze the event counters specified by the GL_event_select field.
• GL_reset_cmd (bit 2): Clear the event count field of the event counters specified by the GL_event_select field. The event select field is not affected.
• GL_event_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR_EMON_L3_CTR_CTL0, bit 23 corresponds to MSR_EMON_L3_CTR_CTL7.
• GL_event_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR_EMON_L3_CTR_CTL0, bit 55 corresponds to MSR_EMON_L3_CTR_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 20-60 for example) is set, the event logic forces the value FFFF_FFFFH into the event count field instead of incrementing it.

20.6.8  Performance Monitoring (P6 Family Processor)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.
PERFORMANCE MONITORING

NOTE

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSRs: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

NOTE

The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed for P6 family processors are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

20.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 20-63 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7)** — Selects the event logic unit to detect certain microarchitectural conditions.
- **Unit mask (UMASK) field (bits 8 through 15)** — Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states.

![Figure 20-63. PerfEvtSel0 and PerfEvtSel1 MSRs](image)

- **USR (user mode) flag (bit 16)** — Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17)** — Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with theUSR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
• **PC (pin control) flag (bit 19)** — When set, the processor toggles the PMI pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PMI pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.

• **INT (APIC interrupt enable) flag (bit 20)** — When set, the processor generates an exception through its local APIC on counter overflow.

• **EN (Enable Counters) Flag (bit 22)** — This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.

• **INV (invert) flag (bit 23)** — When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.

• **Counter mask (CMASK) field (bits 24 through 31)** — When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

### 20.6.8.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

### 20.6.8.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

### 20.6.8.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking.
- Initialize and start counters.
- Stop counters.
- Read the event counters.
- Read the time-stamp counter.
The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 20.6.8.3, “Starting and Stopping the Performance-Monitoring Counters”).

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

20.6.8.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

20.6.9 Performance Monitoring (Pentium Processors)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed for Pentium processors are model-specific for the Pentium processor.

The performance-monitoring events found at https://perfmon-events.intel.com/ are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.
20.6.9.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 20-64). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21)** — Selects (by entering an event code in the field) up to two events to be monitored.

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24)** — Controls the operation of the counter. Control codes are as follows:
  000 — Count nothing (counter disabled).
  001 — Count the selected event while CPL is 0, 1, or 2.
  010 — Count the selected event while CPL is 3.
  011 — Count the selected event regardless of CPL.
  100 — Count nothing (counter disabled).
  101 — Count clocks (duration) while CPL is 0, 1, or 2.
  110 — Count clocks (duration) while CPL is 3.
  111 — Count clocks (duration) regardless of CPL.

The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25)** — Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the overflow or incremented condition. The external signaling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

20.6.9.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an “occurrence event” is being counted, the associated pin is asserted (high) each time the event occurs. When a “duration event” is being counted, the associated PM pin is asserted for the...
entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has
overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be
noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event
occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single
external clock may correspond to multiple internal clocks.

A “count up to” function may be provided when the event pin is programmed to signal an overflow of the counter.
Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific
value less than $2^{40} - 1$. After the counter has been enabled and the prescribed number of events has transpired,
the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an inter-
rupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which
time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and
PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these
pins to indicate breakpoint matches.

### 20.6.9.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in
two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins
  are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an
  event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).

- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to
  indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

### 20.7 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute.
Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop
ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save
  power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both
  logical processors must be halted for performance-monitoring counters to be powered down.

- The processor is asleep as a result of being halted or because of a power-management scheme. There are
different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor’s bus clock
frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions.
- Enhanced Intel SpeedStep Technology transitions (P-state transitions).

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the
Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See
Section 20.7.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock** — The frequency of this clock is the frequency of the processor when the processor is not in turbo
  mode, and not being throttled via Intel SpeedStep.

- **Maximum Clock** — This is the maximum frequency of the processor when turbo mode is at the highest point.

- **Bus Clock** — These clockticks increment at a fixed frequency and help coordinate the bus on some systems.
• **Core Crystal Clock** — This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.

• **Non-halted Clockticks** — Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.

• **Non-sleep Clockticks** — Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.

• **Time-stamp Counter** — See Section 18.17, “Time-Stamp Counter.”

• **Reference Clockticks** — TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 18.17, “Time-Stamp Counter,” for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

• **Non-halted CPI** — Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.

• **Nominal CPI** — Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

### 20.7.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and UMASK 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

### 20.7.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 15, "Power and Thermal Management"), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

• The time stamp counter operates at a fixed-rate frequency of the processor.

• The IA32_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.

• The IA32_FIXED_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.

• The Local APIC timer operation is unaffected by opportunistic processor operation.

• The TSC, IA32_MPERF, and IA32_FIXED_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.
20.7.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-8 “Information Returned by CPUID Instruction”), the nominal TSC frequency can be determined by using the following equation:

\[
\text{Nominal TSC frequency} = \left( \frac{\text{CPUID.15H.ECX}[31:0] \times \text{CPUID.15H.EBX}[31:0]}{\text{CPUID.15H.EAX}[31:0]} \right)
\]

For Intel processors in which CPUID.15H.EBX[31:0] ÷ CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 20-91 can be used to look up the nominal core crystal clock frequency.

### Table 20-91. Nominal Core Crystal Clock Frequency

<table>
<thead>
<tr>
<th>Processor Families/Processor Number Series</th>
<th>Nominal Core Crystal Clock Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon® Scalable Processor Family with CPUID signature 06_55H.</td>
<td>25 MHz</td>
</tr>
<tr>
<td>6th and 7th generation Intel® Core™ processors and Intel® Xeon® W Processor Family.</td>
<td>24 MHz</td>
</tr>
<tr>
<td>Next Generation Intel Atom® processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).</td>
<td>19.2 MHz</td>
</tr>
</tbody>
</table>

**NOTES:**
1. For any processor in which CPUID.15H is enumerated and MSR_PLATFORM_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

20.7.3.1 For Intel® Processors Based on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell Microarchitectures

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

20.7.3.2 For Intel® Processors Based on Nehalem Microarchitecture

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

20.7.3.3 For Intel Atom® Processors Based on Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR_FSB_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

20.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] at (0CDH), see Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR_PLATFORM_ID[12:8]. It corresponds to the Processor Base frequency.
• If XE operation is enabled, the maximum resolved bus ratio is given in MSR_PERF_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR_PERF_STATUS[31] is set, XE operation is enabled. The MSR_PERF_STATUS[31] field is read-only.

20.8 IA32_PERF_CAPABILITIES MSR ENUMERATION

The layout of IA32_PERF_CAPABILITIES MSR is shown in Figure 20-65; it provides enumeration of a variety of interfaces:

• IA32_PERF_CAPABILITIES.LBR_FMT[bits 5:0]: encodes the LBR format, details are described in Section 18.4.8.1.

• IA32_PERF_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist; see Section 20.6.2.4.2.

• IA32_PERF_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers; see Section 20.6.2.4.2.

• IA32_PERF_CAPABILITIES.PEBS_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records; see Section 20.6.2.4.2.

• IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[12]: Indicates IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if 1, see Section 20.8.1.

• IA32_PERF_CAPABILITIES.FULL_WRITE[13]: Indicates the processor supports IA32_A_PMCx interface for updating bits 32 and above of IA32_PMCx; see Section 20.2.6.

• IA32_PERF_CAPABILITIES.PEBS_BASELINE [bit 14]: If set, the following is true:
  — The IA32_PEBS_ENABLE MSR (address 3F1H) exists and all architecturally enumerated fixed and general-purpose counters have corresponding bits in IA32_PEBS_ENABLE that enable generation of PEBS records. The general-purpose counter bits start at bit IA32_PEBS_ENABLE[0], and the fixed counter bits start at bit IA32_PEBS_ENABLE[32].
  — The format of the PEBS record is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 20.6.2.4.2.
  — Extended PEBS is supported. All counters support the PEBS facility, and all events (both precise and non-precise) can generate PEBS records when PEBS is enabled for that counter. Note that not all events may be available on all counters.
  — Adaptive PEBS is supported. The PEBS_DATA_CFG MSR (address 3F2H) and adaptive record enable bits (IA32_PERFEVTSELx.Adaptive_Record and IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record) are supported. The definition of the PEBS_DATA_CFG MSR, including which bits are supported and how they affect the record, is enumerated by IA32_PERF_CAPABILITIES.PEBS_FMT; see Section 20.9.2.3.

• IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE[15]: If set, indicates that the architecture provides built in support for TMA L1 metrics through the PERF_METRICS MSR, see Section 20.3.9.3.

• IA32_PERF_CAPABILITIES.PEBS_OUTPUT_PT_AVAIL[16]: If set on parts that enumerate support for Intel PT (CPUID.0x7.EBX[25]=1), setting IA32_PEBS_ENABLE.PEBS_OUTPUT to 01B will result in PEBS output being written into the Intel PT trace stream. See Section 20.5.5.2.
20.8.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 18.5, “Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel Atom® Processors)”) are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32_PERF_CAPABILITIES MSR. If IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE_WHILE_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTB, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32_DEBUGCTL.FREEZE_WHILE_SMM[bit 14] to 1 only supported as indicated by IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] reporting 1.

### 20.9 PEBS FACILITY

#### 20.9.1 Extended PEBS

- The Extended PEBS feature supports Processor Event Based Sampling (PEBS) on all counters, both fixed function and general purpose; and all performance monitoring events, both precise and non-precise. PEBS can be enabled for the general purpose counters using PEBS_EN_PMCi bits of IA32_PBS_ENABLE (i = 0, 1,..m). PEBS can be enabled for 'i' fixed function counters using the PEBS_EN_FIXEDi bits of IA32_PBS_ENABLE (i = 0, 1, ...n).
A PEBS record due to a precise event will be generated after an instruction that causes the event when the counter has already overflowed. A PEBS record due to a non-precise event will occur at the next opportunity after the counter has overflowed, including immediately after an overflow is set by an MSR write.

Currently, IA32_FIXED_CTR0 counts instructions retired and is a precise event. IA32_FIXED_CTR1, IA32_FIXED_CTR2 … IA32_FIXED_CTRm count as non-precise events.

The Applicable Counter field in the Basic Info Group of the PEBS record indicates which counters caused the PEBS record to be generated. It is in the same format as the enable bits for each counter in IA32_PEBS_ENABLE. As an example, an Applicable Counter field with bits 2 and 32 set would indicate that both general purpose counter 2 and fixed function counter 0 generated the PEBS record.

- To properly use PEBS for the additional counters, software will need to set up the counter reset values in PEBS portion of the DS_BUFFER_MANAGEMENT_AREA data structure that is indicated by the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 20-67. When a counter generates a PEBS records, the appropriate counter reset values will be loaded into that counter. In the above example where general purpose counter 2 and fixed function counter 0 generated the PEBS record, general purpose counter 2 would be reloaded with the value contained in PEBS GP Counter 2 Reset (offset 50H) and fixed function counter 0 would be reloaded with the value contained in PEBS Fixed Counter 0 Reset (offset 80H).

---

**Figure 20-66. Layout of IA32_PEBS_ENABLE MSR**

- PEBS_EN_PMCm (R/W)
- PEBS_EN_FIXED1 (R/W)
- PEBS_EN_FIXED0 (R/W)
- PEBS_EN_PMC1 (R/W)
- PEBS_EN_PMC0 (R/W)
- PEBS_EN_FIXEDn (R/W)

**Reserved**

RESET Value – 00000000 _00000000 H
Extended PEBS support debuts on Intel Atom® processors based on the Goldmont Plus microarchitecture and future Intel® Core™ processors based on the Ice Lake microarchitecture.

### 20.9.2 Adaptive PEBS

The PEBS facility has been enhanced to collect the following CPU state in addition to GPRs, EventingIP, TSC, and memory access related information collected by legacy PEBS:

- XMM registers
- LBR records (TO/FROM/INFO)

The PEBS record is restructured where fields are grouped into Basic group, Memory group, GPR group, XMM group, and LBR group. A new register MSR_PEBS_DATA_CFG provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record's size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.

By default, the PEBS record will only contain the Basic group. Optionally, each counter can be configured to generate a PEBS records with the groups specified in MSR_PEBS_DATA_CFG.
Details and examples for the Adaptive PEBS capability follow below.

### 20.9.2.1 Adaptive_Record Counter Control

- **IA32_PERFEVTSELx.Adaptive_Record[34]**: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_PMCx is set for the corresponding GP counter, an overflow of PMCx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.

- **IA32_FIXED_CTR_CTRL.FCx_Adaptive_Record**: If this bit is set and IA32_PEBS_ENABLE.PEBS_EN_FIXEDx is set for the corresponding Fixed counter, an overflow of FixedCtrx results in generation of an adaptive PEBS record with state information based on the selections made in MSR_PEBS_DATA_CFG. If this bit is not set, a basic record is generated.
20.9.2.2 PEBS Record Format

The data fields in the PEBS record are aggregated into five groups which are described in the sub-sections below. Processors that support Adaptive PEBS implement a new MSR called MSR_PEBS_DATA_CFG which allows software to select the data groups to be captured. The data groups are not placed at fixed locations in the PEBS record, but are positioned immediately after one another, thus making the record format/size variable based on the groups selected.

20.9.2.2.1 Basic Info

The Basic group contains essential information for software to parse a record along with several critical fields. It is always collected.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record Format</td>
<td>[47:0]</td>
<td>This field indicates which data groups are included in the record. The field is zero if none of the counters that triggered the current PEBS record have their Adaptive_Record bit set. Otherwise it contains the value of MSR_PEBS_DATA_CFG.</td>
</tr>
<tr>
<td></td>
<td>[63:48]</td>
<td>This field provides the size of the current record in bytes. Selected groups are packed back-to-back in the record without gaps or padding for unselected groups.</td>
</tr>
</tbody>
</table>
### Table 20-93. Memory Access Info Group

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Access Address</td>
<td>[63:0]</td>
<td>This field contains the linear address of the source of the load, or linear address of the destination (target) of the store. This value is written as a 64-bit address in canonical form.</td>
</tr>
<tr>
<td>Memory Auxiliary Info</td>
<td>[63:0]</td>
<td>When a MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains an encoded value indicating the memory hierarchy source which satisfied the load. These encodings are detailed in Table 20-4 and Table 20-13. If the PEBS assist was triggered for a store uop, this field will contain information indicating the status of the store, as detailed in Table 20-14.</td>
</tr>
<tr>
<td>Memory Access Latency(^1)</td>
<td>[63:0]</td>
<td>When a MEM_TRANS_RETIRED.* event is configured in a General Purpose counter, this field contains the latency to service the load in core clock cycles.</td>
</tr>
<tr>
<td>TSX Auxiliary Info</td>
<td>[31:0]</td>
<td>This field contains the number of cycles in the last TSX region, regardless of whether that region had aborted or committed.</td>
</tr>
<tr>
<td></td>
<td>[63:32]</td>
<td>This field contains the abort details. Refer to Section 20.3.6.5.1.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. In certain conditions, high latencies in fields under “Memory Access Latency” may be observed even when the Data Src of the “Memory Auxiliary Info” field indicates a close source.

Beginning with 12th generation Intel Core processors, the memory access information group has been updated. New fields added are shaded gray in Table 20-94.

---

**20.9.2.2.2 Memory Access Info**

This group contains the legacy PEBS memory-related fields; see Section 20.3.1.1.2.
To determine which fields are supported for certain performance monitoring events, consult the Memory Info attribute in the event lists at https://download.01.org/perfmon/.

**NOTE**

There may be additional block reasons, even if Data-Blk and Address-Blk are both clear, e.g., non-optimal instruction latency.

On P-core, the new Data-Blk and Address-Blk bits require the event LD_BLOCKS.STORE_FORWARD (r8203) to be configured in a programmable counter.

**20.9.2.2.3 GPRs**

This group is captured when the GPR bit is enabled in MSR_PEBS_DATA_CFG. GPRs are always 64 bits wide. If they are selected for non 64-bit mode, the upper 32-bit of the legacy RAX - RDI and all contents of R8-15 GPRs will be filled with 0s. In 64bit mode, the full 64 bit value of each register is written.
The order differs from legacy. The table below shows the order of the GPRs in Ice Lake microarchitecture.

### Table 20-95. GPRs in Ice Lake Microarchitecture

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFLAGS</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RIP</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RAX</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RCX*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RDX*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RBX*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RSP*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RBP*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RSI*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>RDI*</td>
<td>[63:0]</td>
</tr>
<tr>
<td>R8</td>
<td>[63:0]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>R15</td>
<td>[63:0]</td>
</tr>
</tbody>
</table>

The machine state reported in the PEBS record is the committed machine state immediately after the instruction that triggers PEBS completes.

For instance, consider the following instruction sequence:

```
MOV eax, [eax]; triggers PEBS record generation
NOP
```

If the mov instruction triggers PEBS record generation, the EventingIP field in the PEBS record will report the address of the mov, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation. And the value of RIP will contain the linear address of the nop.

#### 20.9.2.2.4 XMMs

This group is captured when the XMM bit is enabled in MSR_PEBS_DATA_CFG and SSE is enabled. If SSE is not enabled, the fields will contain zeroes. XMM8-XMM15 will also contain zeroes if not in 64-bit mode.

### Table 20-96. XMMs

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMM0</td>
<td>[127:0]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>XMM15</td>
<td>[127:0]</td>
</tr>
</tbody>
</table>
20.9.2.2.5 LBRs

To capture LBR data in the PEBS record, the LBR bit in MSR_PEBS_DATA_CFG must be enabled. The number of LBR entries included in the record can be configured in the LBR_entries field of MSR_PEBS_DATA_CFG.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBR[].FROM</td>
<td>[63:0]</td>
<td>Branch from address.</td>
</tr>
<tr>
<td>LBR[].TO</td>
<td>[63:0]</td>
<td>Branch to address.</td>
</tr>
<tr>
<td>LBR[].INFO</td>
<td>[63:0]</td>
<td>Other LBR information, like timing. This field is described in more detail in Section 18.12.1, “MSR_LBR_INFO_x MSR.”</td>
</tr>
</tbody>
</table>

LBR entries are recorded into the record starting at LBR[TOS] and proceeding to LBR[TOS-1] and following. Note that LBR index is modulo the number of LBRs supporting on the processor.

20.9.2.3 MSR_PEBS_DATA_CFG

Bits in MSR_PEBS_DATA_CFG can be set to include data field blocks/groups into adaptive records. The Basic Info group is always included in the record. Additionally, the number of LBR entries included in the record is configurable.
20.9.2.4 PEBS Record Examples

The following example shows the layout of the PEBS record when all data groups are selected (all valid bits in MSR_PEBS_DATA_CFG are set) and maximum number of LBRs are selected. There are no gaps in the PEBS record when a subset of the groups are selected, thus keeping the layout compact. Implementations that do not support some features will have to pad zeroes in the corresponding fields.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Group Name</th>
<th>Field Name</th>
<th>Legacy Name (If Different)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Basic Info</td>
<td>Record Format</td>
<td>New</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Record Size</td>
<td>New</td>
</tr>
<tr>
<td>0x8</td>
<td></td>
<td>Instruction Pointer</td>
<td>EventingRIP</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td>Applicable Counters</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td>TSC</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>Memory Info</td>
<td>Memory Access Address</td>
<td>DLA</td>
</tr>
<tr>
<td>0x28</td>
<td></td>
<td>Memory Auxiliary Info</td>
<td>DATA_SRC</td>
</tr>
<tr>
<td>0x30</td>
<td></td>
<td>Memory Access Latency</td>
<td>Load Latency</td>
</tr>
<tr>
<td>0x38</td>
<td></td>
<td>TSX Auxiliary Info</td>
<td>HLE Information</td>
</tr>
</tbody>
</table>
The following example shows the layout of the PEBS record when Basic, GPR, and LBR group with 3 LBR entries are selected.

**Table 20-99. PEBS Record Example 1**

<table>
<thead>
<tr>
<th>Offset</th>
<th>GPRs</th>
<th>Field Name</th>
<th>Legacy Name (If Different)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>GPRs</td>
<td>RFLAGS</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
<td>RIP</td>
<td></td>
</tr>
<tr>
<td>0x50</td>
<td></td>
<td>RAX</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x88</td>
<td></td>
<td>RDI</td>
<td></td>
</tr>
<tr>
<td>0x90</td>
<td></td>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xC8</td>
<td></td>
<td>R15</td>
<td></td>
</tr>
<tr>
<td>0xD0</td>
<td>XMMs</td>
<td>XMM0</td>
<td>New</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x1C0</td>
<td></td>
<td>XMM15</td>
<td></td>
</tr>
<tr>
<td>0x1D0</td>
<td>LBRs</td>
<td>LBR[TOS].FROM</td>
<td>New</td>
</tr>
<tr>
<td>0x1D8</td>
<td></td>
<td>LBR[TOS].TO</td>
<td></td>
</tr>
<tr>
<td>0x1E0</td>
<td></td>
<td>LBR[TOS].INFO</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x4B8</td>
<td></td>
<td>LBR[TOS +1].FROM</td>
<td></td>
</tr>
<tr>
<td>0x4C0</td>
<td></td>
<td>LBR[TOS +1].TO</td>
<td></td>
</tr>
<tr>
<td>0x4C8</td>
<td></td>
<td>LBR[TOS +1].INFO</td>
<td></td>
</tr>
</tbody>
</table>

The following example shows the layout of the PEBS record when Basic, GPR, and LBR group with 3 LBR entries are selected.

**Table 20-100. PEBS Record Example 2**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Group Name</th>
<th>Field Name</th>
<th>Legacy Name (If Different)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Basic Info</td>
<td>Record Format</td>
<td>New</td>
</tr>
<tr>
<td>0x8</td>
<td></td>
<td>Record Size</td>
<td>New</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td>Instruction Pointer</td>
<td>EventingRIP</td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td>Applicable Counters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TSC</td>
<td></td>
</tr>
</tbody>
</table>
20.9.3 Precise Distribution of Instructions Retired (PDIR) Facility

Precise Distribution of Instructions Retired Facility is available via PEBS on some microarchitectures. Refer to Section 20.3.4.4.4. Counters that support PDIR also vary. See the processor specific sections for availability.

20.9.4 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the “skid” problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 20.3.4.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST_RETIRED, except for UOPS_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

With Reduced Skid PEBS, the skid is precisely one event occurrence. Hence if counting INST_RETIRED, PEBS will indicate the instruction that follows which caused the counter to overflow.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g., via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g., what event is specified in IA32_PERFVTSELx or whether PEBS is enabled for that counter via IA32_PEBSEnable) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

---

Table 20-100. PEBS Record Example 2

<table>
<thead>
<tr>
<th>0x20</th>
<th>GPRs</th>
<th>RFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x28</td>
<td></td>
<td>RIP</td>
</tr>
<tr>
<td>0x30</td>
<td></td>
<td>RAX</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0x68</td>
<td></td>
<td>RDI</td>
</tr>
<tr>
<td>0x70</td>
<td></td>
<td>R8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0xA8</td>
<td></td>
<td>R15</td>
</tr>
</tbody>
</table>

| 0x80 | LBRs           | LBR[TOS].FROM | New |
|------|----------------|---------------|
| 0x88 | LBR[TOS].TO    |               |
| 0xC0 | LBR[TOS].INFO  |               |
| ...  | ...            |               |
| 0xE0 | LBR[TOS +1].FROM |           |
| 0xE8 | LBR[TOS +1].TO  |               |
| 0xF0 | LBR[TOS +1].INFO |            |
20.9.5 EPT-Friendly PEBS

The 3rd generation Intel Xeon Scalable Family of processors based on Ice Lake microarchitecture (and later processors) and the 12th generation Intel Core processor (and later processors) support VMX guest use of PEBS when the DS Area (including the PEBS Buffer and DS Management Area) is allocated from a paged pool of EPT pages. In such a configuration PEBS DS Area accesses may result in VM exits (e.g., EPT violations due to “lazy” EPT page-table entry propagation), and in such cases the PEBS record will not be lost but instead will “skid” to after the subsequent VM Entry back to the guest. For precise events the guest will observe that the record skid by one event occurrence, while for non-precise events the record will skid by one instruction.

20.9.6 PDist: Precise Distribution

PDist eliminates any skid or shadowing effects from PEBS. With PDist, the PEBS record will be generated precisely upon completion of the instruction or operation that causes the counter to overflow (there is no “wait for next occurrence” by default).

PDist is supported by selected counters, and is only supported when those counters are programmed to count select precise events. The legacy PEBS behavior applies to counters that do not support PDist, unless specified otherwise. PDist requires that the INV, ANY, E, and CMASK fields are cleared. Which counters support PDist, and which events are supported for PDist, is model-specific. Further, the counter reload value must not be lesser than 127 for PDist to operate.

For the PDist mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g., via IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g., what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBSENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

20.9.7 Load Latency Facility

The load latency facility provides software a means to characterize the latencies of memory load operations to different levels of cache/memory hierarchy. This facility requires a processor supporting the enhanced PEBS record format in the PEBS buffer.

Beginning with 12th generation Intel Core processors, the load latency facility supports all fields in Table 20-94, “Updated Memory Access Info Group,” in addition to the Memory Access Address field:

- The **Instruction Latency** field measures the load latency from the load’s first dispatch until final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches and data dependencies).
- The **Cache Latency** field measures the subset of cache access latency in core cycles. It starts from the actual cache access until the data is returned by the memory subsystem. The latency is reported for retired demand load operations in core cycles (it does not account for memory ordering blocks).
- The **Data Source** field is an encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 20-101. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory refers to system memory or cache physically attached to another processor package (in a server product).
- Through the **Access Info** field, load latency features binary indications on certain blocks that the load operation may have encountered. Refer to STLB-miss, Is-Lock, Data-Blk and Address-Blk fields in Table 20-94.

**NOTE**

For loads triggered by software prefetch instructions, the cache related fields including Data Source and Cache Latency, report values as if the load was an L1 cache hit (the prefetch completes without waiting for data return, for performance reasons).

---

5. To determine whether an event is precise or supports PDist, consult the relevant attribute in the event lists at [https://download.01.org/perfmon/](https://download.01.org/perfmon/).
To use this feature, software must complete the following steps:

- Complete the PEBS configuration steps.
- Set the Memory Info bit in the PEBS_DATA_CFG MSR.
- One of the relevant IA32_PERFEVTSELx MSRs is programmed to specify the event unit MEM_TRANS_RETIRED.LOAD_LATENCY (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter, IA32_PMCx, will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in an MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with instruction latency greater than this value are eligible for counting and PEBS data reporting. The minimum value that may be programmed in this register is 1.
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register.

Refer to Section 20.3.4.4.2 for further implementation details of Load Latency.

### 20.9.8 Store Latency Facility

Store latency support is available on the 12th generation Intel Core processor. Store latency is a PEBS extension that provides a means to profile store memory accesses in the system. It complements the load latency facility.

Store latency leverages the PEBS facility where it can provide additional information about sampled stores. The additional information includes the data address, memory auxiliary information, and the cache latency of the store access. Normal stores (those preceded with a read-for-ownership) as well as streaming stores are supported by the store latency facility.
Memory store operations typically do not limit performance since they update the memory with no operation that directly depends on them. Thus, data out of this facility should be carefully used once stores are suspected as a performance limiter; for example, once the TMA node of Backend_Bound.Memory_Bound.Store_Bound is flagged\(^6\).

To enable the store latency facility, software must complete the following steps:

- Complete the PEBS configuration steps.
- Set the Memory Info bit in the PEBS_DATA_CFG MSR.
- Program the MEM_TRANS_RETIRED.STORE_SAMPLE event on general-purpose performance-monitoring counter 0 (IA32_PERFEVTSEL0[15:0] = 2CDH).
- Setup the PEBS buffer to hold at least two records, setting both 'PEBS Absolute Maximum' and 'PEBS Interrupt Threshold', should any other counter be used by PEBS (that is whenever IA32_PEBS_ENABLE[x] \(\neq 0 \) for \( x \neq 0 \)).
- Set IA32_PEBS_ENABLE[0].

The store latency information is written into a PEBS record as shown in Table 20-48.

The store latency relies on the PEBS facility, so the PEBS configuration must be completed first. Unlike load latency, there is no option to filter on a subset of stores that exceed a certain threshold.

---

\(^6\) For more details about the method, refer to Section B.1, "Top-Down Analysis Method" of the Intel® 64 and IA-32 Architectures Optimization Reference Manual.
IA-32 processors (beginning with the Intel386 processor) provide two ways to execute new or legacy programs that are assembled and/or compiled to run on an Intel 8086 processor:

- Real-address mode.
- Virtual-8086 mode.

Figure 2-3 shows the relationship of these operating modes to protected mode and system management mode (SMM).

When the processor is powered up or reset, it is placed in the real-address mode. This operating mode almost exactly duplicates the execution environment of the Intel 8086 processor, with some extensions. Virtually any program assembled and/or compiled to run on an Intel 8086 processor will run on an IA-32 processor in this mode.

When running in protected mode, the processor can be switched to virtual-8086 mode to run 8086 programs. This mode also duplicates the execution environment of the Intel 8086 processor, with extensions. In virtual-8086 mode, an 8086 program runs as a separate protected-mode task. Legacy 8086 programs are thus able to run under an operating system (such as Microsoft Windows*) that takes advantage of protected mode and to use protected-mode facilities, such as the protected-mode interrupt- and exception-handling facilities. Protected-mode multitasking permits multiple virtual-8086 mode tasks (with each task running a separate 8086 program) to be run on the processor along with other non-virtual-8086 mode tasks.

This section describes both the basic real-address mode execution environment and the virtual-8086-mode execution environment, available on the IA-32 processors beginning with the Intel386 processor.

## 21.1 REAL-ADDRESS MODE

The IA-32 architecture’s real-address mode runs programs written for the Intel 8086, Intel 8088, Intel 80186, and Intel 80188 processors, or for the real-address mode of the Intel 286, Intel386, Intel486, Pentium, P6 family, Pentium 4, and Intel Xeon processors.

The execution environment of the processor in real-address mode is designed to duplicate the execution environment of the Intel 8086 processor. To an 8086 program, a processor operating in real-address mode behaves like a high-speed 8086 processor. The principal features of this architecture are defined in Chapter 3, "Basic Execution Environment," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

The following is a summary of the core features of the real-address mode execution environment as would be seen by a program written for the 8086:

- The processor supports a nominal 1-MByte physical address space (see Section 21.1.1, “Address Translation in Real-Address Mode,” for specific details). This address space is divided into segments, each of which can be up to 64 KBytes in length. The base of a segment is specified with a 16-bit segment selector, which is shifted left by 4 bits to form a 20-bit offset from address 0 in the address space. An operand within a segment is addressed with a 16-bit offset from the base of the segment. A physical address is thus formed by adding the offset to the 20-bit segment base (see Section 21.1.1, "Address Translation in Real-Address Mode").
- All operands in “native 8086 code” are 8-bit or 16-bit values. (Operand size override prefixes can be used to access 32-bit operands.)
- Eight 16-bit general-purpose registers are provided: AX, BX, CX, DX, SP, BP, SI, and DI. The extended 32 bit registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI) are accessible to programs that explicitly perform a size override operation.
- Four segment registers are provided: CS, DS, SS, and ES. (The FS and GS registers are accessible to programs that explicitly access them.) The CS register contains the segment selector for the code segment; the DS and ES registers contain segment selectors for data segments; and the SS register contains the segment selector for the stack segment.
- The 8086 16-bit instruction pointer (IP) is mapped to the lower 16-bits of the EIP register. Note this register is a 32-bit register and unintentional address wrapping may occur.
The 16-bit FLAGS register contains status and control flags. (This register is mapped to the 16 least significant bits of the 32-bit EFLAGS register.)

All of the Intel 8086 instructions are supported (see Section 21.1.3, "Instructions Supported in Real-Address Mode").

A single, 16-bit-wide stack is provided for handling procedure calls and invocations of interrupt and exception handlers. This stack is contained in the stack segment identified with the SS register. The SP (stack pointer) register contains an offset into the stack segment. The stack grows down (toward lower segment offsets) from the stack pointer. The BP (base pointer) register also contains an offset into the stack segment that can be used as a pointer to a parameter list. When a CALL instruction is executed, the processor pushes the current instruction pointer (the 16 least-significant bits of the EIP register and, on far calls, the current value of the CS register) onto the stack. On a return, initiated with a RET instruction, the processor pops the saved instruction pointer from the stack into the EIP register (and CS register on far returns). When an implicit call to an interrupt or exception handler is executed, the processor pushes the EIP, CS, and EFLAGS (low-order 16-bits only) registers onto the stack. On a return from an interrupt or exception handler, initiated with an IRET instruction, the processor pops the saved instruction pointer and EFLAGS image from the stack into the EIP, CS, and EFLAGS registers.

A single interrupt table, called the “interrupt vector table” or “interrupt table,” is provided for handling interrupts and exceptions (see Figure 21-2). The interrupt table (which has 4-byte entries) takes the place of the interrupt descriptor table (IDT, with 8-byte entries) used when handling protected-mode interrupts and exceptions. Interrupt and exception vector numbers provide an index to entries in the interrupt table. Each entry provides a pointer (called a “vector”) to an interrupt- or exception-handling procedure. See Section 21.1.4, “Interrupt and Exception Handling,” for more details. It is possible for software to relocate the IDT by means of the LIDT instruction on IA-32 processors beginning with the Intel386 processor.

The x87 FPU is active and available to execute x87 FPU instructions in real-address mode. Programs written to run on the Intel 8087 and Intel 287 math coprocessors can be run in real-address mode without modification.

The following extensions to the Intel 8086 execution environment are available in the IA-32 architecture’s real-address mode. If backwards compatibility to Intel 286 and Intel 8086 processors is required, these features should not be used in new programs written to run in real-address mode.

- Two additional segment registers (FS and GS) are available.
- Many of the integer and system instructions that have been added to later IA-32 processors can be executed in real-address mode (see Section 21.1.3, “Instructions Supported in Real-Address Mode”).
- The 32-bit operand prefix can be used in real-address mode programs to execute the 32-bit forms of instructions. This prefix also allows real-address mode programs to use the processor’s 32-bit general-purpose registers.
- The 32-bit address prefix can be used in real-address mode programs, allowing 32-bit offsets.

The following sections describe address formation, registers, available instructions, and interrupt and exception handling in real-address mode. For information on I/O in real-address mode, see Chapter 19, “Input/Output,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

### 21.1.1 Address Translation in Real-Address Mode

In real-address mode, the processor does not interpret segment selectors as indexes into a descriptor table; instead, it uses them directly to form linear addresses as the 8086 processor does. It shifts the segment selector left by 4 bits to form a 20-bit base address (see Figure 21-1). The offset into a segment is added to the base address to create a linear address that maps directly to the physical address space.

When using 8086-style address translation, it is possible to specify addresses larger than 1 MByte. For example, with a segment selector value of FFFFH and an offset of FFFFH, the linear (and physical) address would be 10FFEFH (1 megabyte plus 64 KBytes). The 8086 processor, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby “wrapping” this address to FFEFH. When operating in real-address mode, however, the processor does not truncate such an address and uses it as a physical address. (Note, however, that for IA-32 processors beginning with the Intel486 processor, the A20M# signal can be used in real-address mode to mask address line A20, thereby mimicking the 20-bit wrap-around behavior of the 8086 processor.) Care should be taken to ensure that A20M# based address wrapping is handled correctly in multiprocessor based system.
The IA-32 processors beginning with the Intel386 processor can generate 32-bit offsets using an address override prefix; however, in real-address mode, the value of a 32-bit offset may not exceed FFFFH without causing an exception.

For full compatibility with Intel 286 real-address mode, pseudo-protection faults (interrupt 12 or 13) occur if a 32-bit offset is generated outside the range 0 through FFFFH.

21.1.2 Registers Supported in Real-Address Mode

The register set available in real-address mode includes all the registers defined for the 8086 processor plus the new registers introduced in later IA-32 processors, such as the FS and GS segment registers, the debug registers, the control registers, and the floating-point unit registers. The 32-bit operand prefix allows a real-address mode program to use the 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI).

21.1.3 Instructions Supported in Real-Address Mode

The following instructions make up the core instruction set for the 8086 processor. If backwards compatibility to the Intel 286 and Intel 8086 processors is required, only these instructions should be used in a new program written to run in real-address mode.

- Move (MOV) instructions that move operands between general-purpose registers, segment registers, and between memory and general-purpose registers.
- The exchange (XCHG) instruction.
- Load segment register instructions LDS and LES.
- Arithmetic instructions ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC, CMP, and NEG.
- Logical instructions AND, OR, XOR, and NOT.
- Decimal instructions DAA, DAS, AAA, AAS, AAM, and AAD.
- Stack instructions PUSH and POP (to general-purpose registers and segment registers).
- Type conversion instructions CWD, CDQ, CBW, and CWDE.
- Shift and rotate instructions SAL, SHL, SHR, SAR, ROL, ROR, RCL, and RCR.
- TEST instruction.
- Control instructions JMP, Jcc, CALL, RET, LOOP, LOOPE, and LOOPNE.
- Interrupt instructions INT n, INTO, and IRET.
- EFLAGS control instructions STC, CLC, CMC, CLD, STD, LAHF, SAHF, PUSHF, and POPF.
- I/O instructions IN, INS, OUT, and OUTS.
- Load effective address (LEA) instruction, and translate (XLATB) instruction.
21.4 Interrupt and Exception Handling

When operating in real-address mode, software must provide interrupt and exception-handling facilities that are separate from those provided in protected mode. Even during the early stages of processor initialization when the processor is still in real-address mode, elementary real-address mode interrupt and exception-handling facilities must be provided to ensure reliable operation of the processor, or the initialization code must ensure that no interrupts or exceptions will occur.

The IA-32 processors handle interrupts and exceptions in real-address mode similar to the way they handle them in protected mode. When a processor receives an interrupt or generates an exception, it uses the vector number of the interrupt or exception as an index into the interrupt table. (In protected mode, the interrupt table is called the interrupt descriptor table (IDT), but in real-address mode, the table is usually called the interrupt vector table, or simply the interrupt table.) The entry in the interrupt vector table provides a pointer to an interrupt- or exception-handler procedure. (The pointer consists of a segment selector for a code segment and a 16-bit offset into the segment.) The processor performs the following actions to make an implicit call to the selected handler:

1. Pushes the current values of the CS and EIP registers onto the stack. (Only the 16 least-significant bits of the EIP register are pushed.)
2. Pushes the low-order 16 bits of the EFLAGS register onto the stack.
3. Clears the IF flag in the EFLAGS register to disable interrupts.
4. Clears the TF, RF, and AC flags, in the EFLAGS register.
5. Transfers program control to the location specified in the interrupt vector table.

An IRET instruction at the end of the handler procedure reverses these steps to return program control to the interrupted program. Exceptions do not return error codes in real-address mode.

The interrupt vector table is an array of 4-byte entries (see Figure 21-2). Each entry consists of a far pointer to a handler procedure, made up of a segment selector and an offset. The processor scales the interrupt or exception vector by 4 to obtain an offset into the interrupt table. Following reset, the base of the interrupt vector table is located at physical address 0 and its limit is set to 3FFH. In the Intel 8086 processor, the base address and limit of the interrupt vector table cannot be changed. In the later IA-32 processors, the base address and limit of the interrupt vector table are contained in the IDTR register and can be changed using the LIDT instruction.

(For backward compatibility to Intel 8086 processors, the default base address and limit of the interrupt vector table should not be changed.)

![Figure 21-2. Interrupt Vector Table in Real-Address Mode](image)

Table 21-1 shows the interrupt and exception vectors that can be generated in real-address mode and virtual-8086 mode, and in the Intel 8086 processor. See Chapter 6, “Interrupt and Exception Handling,” for a description of the exception conditions.

### 21.2 VIRTUAL-8086 MODE

Virtual-8086 mode is actually a special type of a task that runs in protected mode. When the operating-system or executive switches to a virtual-8086-mode task, the processor emulates an Intel 8086 processor. The execution environment of the processor while in the 8086-emulation state is the same as is described in Section 21.1, “Real-Address Mode,” for real-address mode, including the extensions. The major difference between the two modes is that in virtual-8086 mode the 8086 emulator uses some protected-mode services (such as the protected-mode interrupt and exception-handling and paging facilities).
As in real-address mode, any new or legacy program that has been assembled and/or compiled to run on an Intel 8086 processor will run in a virtual-8086-mode task. And several 8086 programs can be run as virtual-8086-mode tasks concurrently with normal protected-mode tasks, using the processor’s multitasking facilities.

### Table 21-1. Real-Address Mode Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Description</th>
<th>Real-Address Mode</th>
<th>Virtual-8086 Mode</th>
<th>Intel 8086 Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide Error (#DE)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>Debug Exception (#DB)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>NMI Interrupt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint (#BP)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Overflow (#OF)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>BOUND Range Exceeded (#BR)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Invalid Opcode (#UD)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Device Not Available (#NM)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Double Fault (#DF)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>(Intel reserved. Do not use.)</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Invalid TSS (#TS)</td>
<td>Reserved</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Segment Not Present (#NP)</td>
<td>Reserved</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>Stack Fault (#SS)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>General Protection (#GP)*</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>Page Fault (#PF)</td>
<td>Reserved</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td>(Intel reserved. Do not use.)</td>
<td>Reserved</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>Floating-Point Error (#MF)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>17</td>
<td>Alignment Check (#AC)</td>
<td>Reserved</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>18</td>
<td>Machine Check (#MC)</td>
<td>Yes</td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>19-31</td>
<td>(Intel reserved. Do not use.)</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>32-255</td>
<td>User Defined Interrupts</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**NOTE:**
* In the real-address mode, vector 13 is the segment overrun exception. In protected and virtual-8086 modes, this exception covers all general-protection error conditions, including traps to the virtual-8086 monitor from virtual-8086 mode.

### 21.2.1 Enabling Virtual-8086 Mode

The processor runs in virtual-8086 mode when the VM (virtual machine) flag in the EFLAGS register is set. This flag can only be set when the processor switches to a new protected-mode task or resumes virtual-8086 mode via an IRET instruction.

System software cannot change the state of the VM flag directly in the EFLAGS register (for example, by using the POPFD instruction). Instead it changes the flag in the image of the EFLAGS register stored in the TSS or on the stack following a call to an interrupt- or exception-handler procedure. For example, software sets the VM flag in the EFLAGS image in the TSS when first creating a virtual-8086 task.

The processor tests the VM flag under three general conditions:
- When loading segment registers, to determine whether to use 8086-style address translation.
- When decoding instructions, to determine which instructions are not supported in virtual-8086 mode and which instructions are sensitive to IOPL.
When checking privileged instructions, on page accesses, or when performing other permission checks.
(Virtual-8086 mode always executes at CPL 3.)

### 21.2.2 Structure of a Virtual-8086 Task

A virtual-8086-mode task consists of the following items:

- A 32-bit TSS for the task.
- The 8086 program.
- A virtual-8086 monitor.
- 8086 operating-system services.

The TSS of the new task must be a 32-bit TSS, not a 16-bit TSS, because the 16-bit TSS does not load the most-significant word of the EFLAGS register, which contains the VM flag. All TSS’s, stacks, data, and code used to handle exceptions when in virtual-8086 mode must also be 32-bit segments.

The processor enters virtual-8086 mode to run the 8086 program and returns to protected mode to run the virtual-8086 monitor.

The virtual-8086 monitor is a 32-bit protected-mode code module that runs at a CPL of 0. The monitor consists of initialization, interrupt- and exception-handling, and I/O emulation procedures that emulate a personal computer or other 8086-based platform. Typically, the monitor is either part of or closely associated with the protected-mode general-protection (#GP) exception handler, which also runs at a CPL of 0. As with any protected-mode code module, code-segment descriptors for the virtual-8086 monitor must exist in the GDT or in the task’s LDT. The virtual-8086 monitor also may need data-segment descriptors so it can examine the IDT or other parts of the 8086 program in the first 1 MByte of the address space. The linear addresses above 10FFEFH are available for the monitor, the operating system, and other system software.

The 8086 operating-system services consists of a kernel and/or operating-system procedures that the 8086 program makes calls to. These services can be implemented in either of the following two ways:

- They can be included in the 8086 program. This approach is desirable for either of the following reasons:
  - The 8086 program code modifies the 8086 operating-system services.
  - There is not sufficient development time to merge the 8086 operating-system services into main operating system or executive.

- They can be implemented or emulated in the virtual-8086 monitor. This approach is desirable for any of the following reasons:
  - The 8086 operating-system procedures can be more easily coordinated among several virtual-8086 tasks.
  - Memory can be saved by not duplicating 8086 operating-system procedure code for several virtual-8086 tasks.
  - The 8086 operating-system procedures can be easily emulated by calls to the main operating system or executive.

The approach chosen for implementing the 8086 operating-system services may result in different virtual-8086-mode tasks using different 8086 operating-system services.

### 21.2.3 Paging of Virtual-8086 Tasks

Even though a program running in virtual-8086 mode can use only 20-bit linear addresses, the processor converts these addresses into 32-bit linear addresses before mapping them to the physical address space. If paging is being used, the 8086 address space for a program running in virtual-8086 mode can be paged and located in a set of pages in physical address space. If paging is used, it is transparent to the program running in virtual-8086 mode just as it is for any task running on the processor.

Paging is not necessary for a single virtual-8086-mode task, but paging is useful or necessary in the following situations:
When running multiple virtual-8086-mode tasks. Here, paging allows the lower 1 MByte of the linear address space for each virtual-8086-mode task to be mapped to a different physical address location.

When emulating the 8086 address-wraparound that occurs at 1 MByte. When using 8086-style address translation, it is possible to specify addresses larger than 1 MByte. These addresses automatically wraparound in the Intel 8086 processor (see Section 21.1.1, “Address Translation in Real-Address Mode”). If any 8086 programs depend on address wraparound, the same effect can be achieved in a virtual-8086-mode task by mapping the linear addresses between 100000H and 110000H and linear addresses between 0 and 10000H to the same physical addresses.

When sharing the 8086 operating-system services or ROM code that is common to several 8086 programs running as different 8086-mode tasks.

When redirecting or trapping references to memory-mapped I/O devices.

### 21.2.4 Protection within a Virtual-8086 Task

Protection is not enforced between the segments of an 8086 program. Either of the following techniques can be used to protect the system software running in a virtual-8086-mode task from the 8086 program:

- Reserve the first 1 MByte plus 64 KBytes of each task’s linear address space for the 8086 program. An 8086 processor task cannot generate addresses outside this range.
- Use the U/S flag of page-table entries to protect the virtual-8086 monitor and other system software in the virtual-8086 mode task space. When the processor is in virtual-8086 mode, the CPL is 3. Therefore, an 8086 processor program has only user privileges. If the pages of the virtual-8086 monitor have supervisor privilege, they cannot be accessed by the 8086 program.

### 21.2.5 Entering Virtual-8086 Mode

Figure 21-3 summarizes the methods of entering and leaving virtual-8086 mode. The processor switches to virtual-8086 mode in either of the following situations:

- Task switch when the VM flag is set to 1 in the EFLAGS register image stored in the TSS for the task. Here the task switch can be initiated in either of two ways:
  - A CALL or JMP instruction.
  - An IRET instruction, where the NT flag in the EFLAGS image is set to 1.

- Return from a protected-mode interrupt or exception handler when the VM flag is set to 1 in the EFLAGS register image on the stack.

When a task switch is used to enter virtual-8086 mode, the TSS for the virtual-8086-mode task must be a 32-bit TSS. (If the new TSS is a 16-bit TSS, the upper word of the EFLAGS register is not in the TSS, causing the processor to clear the VM flag when it loads the EFLAGS register.) The processor updates the VM flag prior to loading the segment registers from their images in the new TSS. The new setting of the VM flag determines whether the processor interprets the contents of the segment registers as 8086-style segment selectors or protected-mode segment selectors. When the VM flag is set, the segment registers are loaded from the TSS, using 8086-style address translation to form base addresses.

See Section 21.3, “Interrupt and Exception Handling in Virtual-8086 Mode,” for information on entering virtual-8086 mode on a return from an interrupt or exception handler.
21.2.6 Leaving Virtual-8086 Mode

The processor can leave the virtual-8086 mode only through an interrupt or exception. The following are situations where an interrupt or exception will lead to the processor leaving virtual-8086 mode (see Figure 21-3):

- The processor services a hardware interrupt generated to signal the suspension of execution of the virtual-8086 application. This hardware interrupt may be generated by a timer or other external mechanism. Upon receiving the hardware interrupt, the processor enters protected mode and switches to a protected-mode (or another virtual-8086 mode) task either through a task gate in the protected-mode IDT or through a trap or interrupt gate that points to a handler that initiates a task switch. A task switch from a virtual-8086 task to another task loads the EFLAGS register from the TSS of the new task. The value of the VM flag in the new EFLAGS determines if the new task executes in virtual-8086 mode or not.

- The processor services an exception caused by code executing the virtual-8086 task or services a hardware interrupt that “belongs to” the virtual-8086 task. Here, the processor enters protected mode and services the
exception or hardware interrupt through the protected-mode IDT (normally through an interrupt or trap gate) and the protected-mode exception- and interrupt-handlers. The processor may handle the exception or interrupt within the context of the virtual 8086 task and return to virtual-8086 mode on a return from the handler procedure. The processor may also execute a task switch and handle the exception or interrupt in the context of another task.

- The processor services a software interrupt generated by code executing in the virtual-8086 task (such as a software interrupt to call a MS-DOS* operating system routine). The processor provides several methods of handling these software interrupts, which are discussed in detail in Section 21.3.3, "Class 3—Software Interrupt Handling in Virtual-8086 Mode." Most of them involve the processor entering protected mode, often by means of a general-protection (#GP) exception. In protected mode, the processor can send the interrupt to the virtual-8086 monitor for handling and/or redirect the interrupt back to the application program running in virtual-8086 mode task for handling.

IA-32 processors that incorporate the virtual mode extension (enabled with the VME flag in control register CR4) are capable of redirecting software-generated interrupts back to the program's interrupt handlers without leaving virtual-8086 mode. See Section 21.3.3.4, "Method 5: Software Interrupt Handling," for more information on this mechanism.

- A hardware reset initiated by asserting the RESET or INIT pin is a special kind of interrupt. When a RESET or INIT is signaled while the processor is in virtual-8086 mode, the processor leaves virtual-8086 mode and enters real-address mode.

- Execution of the HLT instruction in virtual-8086 mode will cause a general-protection (GP#) fault, which the protected-mode handler generally sends to the virtual-8086 monitor. The virtual-8086 monitor then determines the correct execution sequence after verifying that it was entered as a result of a HLT execution. See Section 21.3, "Interrupt and Exception Handling in Virtual-8086 Mode," for information on leaving virtual-8086 mode to handle an interrupt or exception generated in virtual-8086 mode.

21.2.7 Sensitive Instructions

When an IA-32 processor is running in virtual-8086 mode, the CLI, STI, PUSHF, POPF, INT n, and IRET instructions are sensitive to IOPL. The IN, INS, OUT, and OUTS instructions, which are sensitive to IOPL in protected mode, are not sensitive in virtual-8086 mode.

The CPL is always 3 while running in virtual-8086 mode; if the IOPL is less than 3, an attempt to use the IOPL-sensitive instructions listed above triggers a general-protection exception (#GP). These instructions are sensitive to IOPL to give the virtual-8086 monitor a chance to emulate the facilities they affect.

21.2.8 Virtual-8086 Mode I/O

Many 8086 programs written for non-multitasking systems directly access I/O ports. This practice may cause problems in a multitasking environment. If more than one program accesses the same port, they may interfere with each other. Most multitasking systems require application programs to access I/O ports through the operating system. This results in simplified, centralized control.

The processor provides I/O protection for creating I/O that is compatible with the environment and transparent to 8086 programs. Designers may take any of several possible approaches to protecting I/O ports:

- Protect the I/O address space and generate exceptions for all attempts to perform I/O directly.
- Let the 8086 program perform I/O directly.
- Generate exceptions on attempts to access specific I/O ports.
- Generate exceptions on attempts to access specific memory-mapped I/O ports.

The method of controlling access to I/O ports depends upon whether they are I/O-port mapped or memory mapped.
21.2.8.1 I/O-Port-Mapped I/O

The I/O permission bit map in the TSS can be used to generate exceptions on attempts to access specific I/O port addresses. The I/O permission bit map of each virtual-8086-mode task determines which I/O addresses generate exceptions for that task. Because each task may have a different I/O permission bit map, the addresses that generate exceptions for one task may be different from the addresses for another task. This differs from protected mode in which, if the CPL is less than or equal to the IOPL, I/O access is allowed without checking the I/O permission bit map. See Chapter 19, “Input/Output,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information about the I/O permission bit map.

21.2.8.2 Memory-Mapped I/O

In systems which use memory-mapped I/O, the paging facilities of the processor can be used to generate exceptions for attempts to access I/O ports. The virtual-8086 monitor may use paging to control memory-mapped I/O in these ways:

- Map part of the linear address space of each task that needs to perform I/O to the physical address space where I/O ports are placed. By putting the I/O ports at different addresses (in different pages), the paging mechanism can enforce isolation between tasks.
- Map part of the linear address space to pages that are not-present. This generates an exception whenever a task attempts to perform I/O to those pages. System software then can interpret the I/O operation being attempted.

Software emulation of the I/O space may require too much operating system intervention under some conditions. In these cases, it may be possible to generate an exception for only the first attempt to access I/O. The system software then may determine whether a program can be given exclusive control of I/O temporarily, the protection of the I/O space may be lifted, and the program allowed to run at full speed.

21.2.8.3 Special I/O Buffers

Buffers of intelligent controllers (for example, a bit-mapped frame buffer) also can be emulated using page mapping. The linear space for the buffer can be mapped to a different physical space for each virtual-8086-mode task. The virtual-8086 monitor then can control which virtual buffer to copy onto the real buffer in the physical address space.

21.3 INTERRUPT AND EXCEPTION HANDLING IN VIRTUAL-8086 MODE

When the processor receives an interrupt or detects an exception condition while in virtual-8086 mode, it invokes an interrupt or exception handler, just as it does in protected or real-address mode. The interrupt or exception handler that is invoked and the mechanism used to invoke it depends on the class of interrupt or exception that has been detected or generated and the state of various system flags and fields.

In virtual-8086 mode, the interrupts and exceptions are divided into three classes for the purposes of handling:

- **Class 1** — All processor-generated exceptions and all hardware interrupts, including the NMI interrupt and the hardware interrupts sent to the processor’s external interrupt delivery pins. All class 1 exceptions and interrupts are handled by the protected-mode exception and interrupt handlers.
- **Class 2** — Special case for maskable hardware interrupts (Section 6.3.2, “Maskable Hardware Interrupts”) when the virtual mode extensions are enabled.
- **Class 3** — All software-generated interrupts, that is interrupts generated with the INT \( n \) instruction\(^1\).

The method the processor uses to handle class 2 and 3 interrupts depends on the setting of the following flags and fields:

- **IOPL field (bits 12 and 13 in the EFLAGS register)** — Controls how class 3 software interrupts are handled when the processor is in virtual-8086 mode (see Section 2.3, “System Flags and Fields in the EFLAGS

---

\(^1\) The INT 3 instruction is a special case (see the description of the INT \( n \) instruction in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).
Register”). This field also controls the enabling of the VIF and VIP flags in the EFLAGS register when the VME flag is set. The VIF and VIP flags are provided to assist in the handling of class 2 maskable hardware interrupts.

- **VME flag (bit 0 in control register CR4)** — Enables the virtual mode extension for the processor when set (see Section 2.5, “Control Registers”).

- **Software interrupt redirection bit map (32 bytes in the TSS, see Figure 21-5)** — Contains 256 flags that indicate how class 3 software interrupts should be handled when they occur in virtual-8086 mode. A software interrupt can be directed either to the interrupt and exception handlers in the currently running 8086 program or to the protected-mode interrupt and exception handlers.

- **The virtual interrupt flag (VIF) and virtual interrupt pending flag (VIP) in the EFLAGS register** — Provides virtual interrupt support for the handling of class 2 maskable hardware interrupts (see Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism”).

**NOTE**

The VME flag, software interrupt redirection bit map, and VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor.

The following sections describe the actions that processor takes and the possible actions of interrupt and exception handlers for the two classes of interrupts described in the previous paragraphs. These sections describe three possible types of interrupt and exception handlers:

- **Protected-mode interrupt and exceptions handlers** — These are the standard handlers that the processor calls through the protected-mode IDT.

- **Virtual-8086 monitor interrupt and exception handlers** — These handlers are resident in the virtual-8086 monitor, and they are commonly accessed through a general-protection exception (#GP, interrupt 13) that is directed to the protected-mode general-protection exception handler.

- **8086 program interrupt and exception handlers** — These handlers are part of the 8086 program that is running in virtual-8086 mode.

The following sections describe how these handlers are used, depending on the selected class and method of interrupt and exception handling.

### 21.3.1 Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode

In virtual-8086 mode, the Pentium, P6 family, Pentium 4, and Intel Xeon processors handle hardware interrupts and exceptions in the same manner as they are handled by the Intel486 and Intel386 processors. They invoke the protected-mode interrupt or exception handler that the interrupt or exception vector points to in the IDT. Here, the IDT entry must contain either a 32-bit trap or interrupt gate or a task gate. The following sections describe various ways that a virtual-8086 mode interrupt or exception can be handled after the protected-mode handler has been invoked.

See Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism,” for a description of the virtual interrupt mechanism that is available for handling maskable hardware interrupts while in virtual-8086 mode. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled in the same manner as exceptions, as described in the following sections.

#### 21.3.1.1 Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate

When an interrupt or exception vector points to a 32-bit trap or interrupt gate in the IDT, the gate must in turn point to a nonconforming, privilege-level 0, code segment. When accessing this code segment, processor performs the following steps.

1. Switches to 32-bit protected mode and privilege level 0.
2. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 21-4).
3. Clears the segment registers. Saving the DS, ES, FS, and GS registers on the stack and then clearing the registers lets the interrupt or exception handler safely save and restore these registers regardless of the type segment selectors they contain (protected-mode or 8086-style). The interrupt and exception handlers, which may be called in the context of either a protected-mode task or a virtual-8086-mode task, can use the same code sequences for saving and restoring the registers for any task. Clearing these registers before execution of the IRET instruction does not cause a trap in the interrupt handler. Interrupt procedures that expect values in the segment registers or that return values in the segment registers must use the register images saved on the stack for privilege level 0.

4. Clears VM, NT, RF, and TF flags (in the EFLAGS register). If the gate is an interrupt gate, clears the IF flag.

5. Begins executing the selected interrupt or exception handler.

If the trap or interrupt gate references a procedure in a conforming segment or in a segment at a privilege level other than 0, the processor generates a general-protection exception (#GP). Here, the error code is the segment selector of the code segment to which a call was attempted.

Interrupt and exception handlers can examine the VM flag on the stack to determine if the interrupted procedure was running in virtual-8086 mode. If so, the interrupt or exception can be handled in one of three ways:

- The protected-mode interrupt or exception handler that was called can handle the interrupt or exception.
- The protected-mode interrupt or exception handler can call the virtual-8086 monitor to handle the interrupt or exception.
- The virtual-8086 monitor (if called) can in turn pass control back to the 8086 program’s interrupt and exception handler.

If the interrupt or exception is handled with a protected-mode handler, the handler can return to the interrupted program in virtual-8086 mode by executing an IRET instruction. This instruction loads the EFLAGS and segment registers from the images saved in the privilege level 0 stack (see Figure 21-4). A set VM flag in the EFLAGS image causes the processor to switch back to virtual-8086 mode. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

The virtual-8086 monitor runs at privilege level 0, like the protected-mode interrupt and exception handlers. It is commonly closely tied to the protected-mode general-protection exception (#GP, vector 13) handler. If the
protected-mode interrupt or exception handler calls the virtual-8086 monitor to handle the interrupt or exception, the return from the virtual-8086 monitor to the interrupted virtual-8086 mode program requires two return instructions: a RET instruction to return to the protected-mode handler and an IRET instruction to return to the interrupted program.

The virtual-8086 monitor has the option of directing the interrupt and exception back to an interrupt or exception handler that is part of the interrupted 8086 program, as described in Section 21.3.1.2, “Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler.”

21.3.1.2 Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler

Because it was designed to run on an 8086 processor, an 8086 program running in a virtual-8086-mode task contains an 8086-style interrupt vector table, which starts at linear address 0. If the virtual-8086 monitor correctly directs an interrupt or exception vector back to the virtual-8086-mode task it came from, the handlers in the 8086 program can handle the interrupt or exception. The virtual-8086 monitor must carry out the following steps to send an interrupt or exception back to the 8086 program:

1. Use the 8086 interrupt vector to locate the appropriate handler procedure in the 8086 program interrupt table.
2. Store the EFLAGS (low-order 16 bits only), CS and EIP values of the 8086 program on the privilege-level 3 stack. This is the stack that the virtual-8086-mode task is using. (The 8086 handler may use or modify this information.)
3. Change the return link on the privilege-level 0 stack to point to the privilege-level 3 handler procedure.
4. Execute an IRET instruction to pass control to the 8086 program handler.
5. When the IRET instruction from the privilege-level 3 handler triggers a general-protection exception (#GP) and thus effectively again calls the virtual-8086 monitor, restore the return link on the privilege-level 0 stack to point to the original, interrupted, privilege-level 3 procedure.
6. Copy the low order 16 bits of the EFLAGS image from the privilege-level 3 stack to the privilege-level 0 stack (because some 8086 handlers modify these flags to return information to the code that caused the interrupt).
7. Execute an IRET instruction to pass control back to the interrupted 8086 program.

Note that if an operating system intends to support all 8086 MS-DOS-based programs, it is necessary to use the actual 8086 interrupt and exception handlers supplied with the program. The reason for this is that some programs modify their own interrupt vector table to substitute (or hook in series) their own specialized interrupt and exception handlers.

21.3.1.3 Handling an Interrupt or Exception Through a Task Gate

When an interrupt or exception vector points to a task gate in the IDT, the processor performs a task switch to the selected interrupt- or exception-handling task. The following actions are carried out as part of this task switch:

1. The EFLAGS register with the VM flag set is saved in the current TSS.
2. The link field in the TSS of the called task is loaded with the segment selector of the TSS for the interrupted virtual-8086-mode task.
3. The EFLAGS register is loaded from the image in the new TSS, which clears the VM flag and causes the processor to switch to protected mode.
4. The NT flag in the EFLAGS register is set.
5. The processor begins executing the selected interrupt- or exception-handler task.

When an IRET instruction is executed in the handler task and the NT flag in the EFLAGS register is set, the processors switches from a protected-mode interrupt- or exception-handler task back to a virtual-8086-mode task. Here, the EFLAGS and segment registers are loaded from images saved in the TSS for the virtual-8086-mode task. If the VM flag is set in the EFLAGS image, the processor switches back to virtual-8086 mode on the task switch. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.
21.3.2 Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism

Maskable hardware interrupts are those interrupts that are delivered through the INTR# pin or through an interrupt request to the local APIC (see Section 6.3.2, “Maskable Hardware Interrupts”). These interrupts can be inhibited (masked) from interrupting an executing program or task by clearing the IF flag in the EFLAGS register.

When the VME flag in control register CR4 is set and the IOPL field in the EFLAGS register is less than 3, two additional flags are activated in the EFLAGS register:

- VIF (virtual interrupt) flag, bit 19 of the EFLAGS register.
- VIP (virtual interrupt pending) flag, bit 20 of the EFLAGS register.

These flags provide the virtual-8086 monitor with more efficient control over handling maskable hardware interrupts that occur during virtual-8086 mode tasks. They also reduce interrupt-handling overhead, by eliminating the need for all IF related operations (such as PUSHF, POPF, CLI, and STI instructions) to trap to the virtual-8086 monitor. The purpose and use of these flags are as follows.

**NOTE**

The VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled as class 1 interrupts. Here, if VIF and VIP flags are needed, the virtual-8086 monitor can implement them in software.

Existing 8086 programs commonly set and clear the IF flag in the EFLAGS register to enable and disable maskable hardware interrupts, respectively; for example, to disable interrupts while handling another interrupt or an exception. This practice works well in single task environments, but can cause problems in multitasking and multiple-processor environments, where it is often desirable to prevent an application program from having direct control over the handling of hardware interrupts. When using earlier IA-32 processors, this problem was often solved by creating a virtual IF flag in software. The IA-32 processors (beginning with the Pentium processor) provide hardware support for this virtual IF flag through the VIF and VIP flags.

The VIF flag is a virtualized version of the IF flag, which an application program running from within a virtual-8086 task can use to control the handling of maskable hardware interrupts. When the VIF flag is enabled, the CLI and STI instructions operate on the VIF flag instead of the IF flag. When an 8086 program executes the CLI instruction, the processor clears the VIF flag to request that the virtual-8086 monitor inhibit maskable hardware interrupts from interrupting program execution; when it executes the STI instruction, the processor sets the VIF flag requesting that the virtual-8086 monitor enable maskable hardware interrupts for the 8086 program. But actually the IF flag, managed by the operating system, always controls whether maskable hardware interrupts are enabled. Also, if under these circumstances an 8086 program tries to read or change the IF flag using the PUSHF or POPF instructions, the processor will change the VIF flag instead, leaving IF unchanged.

The VIP flag provides software a means of recording the existence of a deferred (or pending) maskable hardware interrupt. This flag is read by the processor but never explicitly written by the processor; it can only be written by software.

If the IF flag is set and the VIF and VIP flags are enabled, and the processor receives a maskable hardware interrupt (interrupt vector 0 through 255), the processor performs and the interrupt handler software should perform the following operations:

1. The processor invokes the protected-mode interrupt handler for the interrupt received, as described in the following steps. These steps are almost identical to those described for method 1 interrupt and exception handling in Section 21.3.1.1, “Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate”:
   a. Switches to 32-bit protected mode and privilege level 0.
   b. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 21-4).
   c. Clears the segment registers.
d. Clears the VM flag in the EFLAGS register.
e. Begins executing the selected protected-mode interrupt handler.

2. The recommended action of the protected-mode interrupt handler is to read the VM flag from the EFLAGS image on the stack. If this flag is set, the handler makes a call to the virtual-8086 monitor.

3. The virtual-8086 monitor should read the VIF flag in the EFLAGS register.
   — If the VIF flag is clear, the virtual-8086 monitor sets the VIP flag in the EFLAGS image on the stack to indicate that there is a deferred interrupt pending and returns to the protected-mode handler.
   — If the VIF flag is set, the virtual-8086 monitor can handle the interrupt if it "belongs" to the 8086 program running in the interrupted virtual-8086 task; otherwise, it can call the protected-mode interrupt handler to handle the interrupt.

4. The protected-mode handler executes a return to the program executing in virtual-8086 mode.
5. Upon returning to virtual-8086 mode, the processor continues execution of the 8086 program.

When the 8086 program is ready to receive maskable hardware interrupts, it executes the STI instruction to set the VIF flag (enabling maskable hardware interrupts). Prior to setting the VIF flag, the processor automatically checks the VIP flag and does one of the following, depending on the state of the flag:
• If the VIP flag is clear (indicating no pending interrupts), the processor sets the VIF flag.
• If the VIP flag is set (indicating a pending interrupt), the processor generates a general-protection exception (#GP).

The recommended action of the protected-mode general-protection exception handler is to then call the virtual-8086 monitor and let it handle the pending interrupt. After handling the pending interrupt, the typical action of the virtual-8086 monitor is to clear the VIF flag and set the VIP flag in the EFLAGS image on the stack, and then execute a return to the virtual-8086 mode. The next time the processor receives a maskable hardware interrupt, it will then handle it as described in steps 1 through 5 earlier in this section.

If the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, it generates a general-protection exception. This action allows the virtual-8086 monitor to handle the pending interrupt for the virtual-8086 mode task for which the VIF flag is enabled. Note that this situation can only occur immediately following execution of a POPF or IRET instruction or upon entering a virtual-8086 mode task through a task switch.

Note that the states of the VIF and VIP flags are not modified in real-address mode or during transitions between real-address and protected modes.

NOTE
The virtual interrupt mechanism described in this section is also available for use in protected mode, see Section 21.4, "Protected-Mode Virtual Interrupts."

21.3.3 Class 3—Software Interrupt Handling in Virtual-8086 Mode

When the processor receives a software interrupt (an interrupt generated with the INT n instruction) while in virtual-8086 mode, it can use any of six different methods to handle the interrupt. The method selected depends on the settings of the VME flag in control register CR4, the IOPL field in the EFLAGS register, and the software interrupt redirection bit map in the TSS. Table 21-2 lists the six methods of handling software interrupts in virtual-8086 mode and the respective settings of the VME flag, IOPL field, and the bits in the interrupt redirection bit map for each method. The table also summarizes the various actions the processor takes for each method.

The VME flag enables the virtual mode extensions for the Pentium and later IA-32 processors. When this flag is clear, the processor responds to interrupts and exceptions in virtual-8086 mode in the same manner as an Intel386 or Intel486 processor does. When this flag is set, the virtual mode extension provides the following enhancements to virtual-8086 mode:
• Speeds up the handling of software-generated interrupts in virtual-8086 mode by allowing the processor to bypass the virtual-8086 monitor and redirect software interrupts back to the interrupt handlers that are part of the currently running 8086 program.
• Supports virtual interrupts for software written to run on the 8086 processor.
The IOPL value interacts with the VME flag and the bits in the interrupt redirection bit map to determine how specific software interrupts should be handled.

The software interrupt redirection bit map (see Figure 21-5) is a 32-byte field in the TSS. This map is located directly below the I/O permission bit map in the TSS. Each bit in the interrupt redirection bit map is mapped to an interrupt vector. Bit 0 in the interrupt redirection bit map (which maps to vector zero in the interrupt table) is located at the I/O base map address in the TSS minus 32 bytes. When a bit in this bit map is set, it indicates that the associated software interrupt (interrupt generated with an INT \( n \) instruction) should be handled through the protected-mode IDT and interrupt and exception handlers. When a bit in this bit map is clear, the processor redirects the associated software interrupt back to the interrupt table in the 8086 program (located at linear address 0 in the program’s address space).

**NOTE**

The software interrupt redirection bit map does not affect hardware generated interrupts and exceptions. Hardware generated interrupts and exceptions are always handled by the protected-mode interrupt and exception handlers.

### Table 21-2. Software Interrupt Handling Methods While in Virtual-8086 Mode

<table>
<thead>
<tr>
<th>Method</th>
<th>VME</th>
<th>IOPL</th>
<th>Bit in Redir. Bitmap*</th>
<th>Processor Action</th>
</tr>
</thead>
</table>
| 1      | 0   | 3    | X                    | Interrupt directed to a protected-mode interrupt handler:  
|        |     |      |                      | • Switches to privilege-level 0 stack.  
|        |     |      |                      | • Pushes GS, FS, DS, and ES onto privilege-level 0 stack.  
|        |     |      |                      | • Pushes SS, ESP, EFLAGS, CS, and EIP of interrupted task onto privilege-level 0 stack.  
|        |     |      |                      | • Clears VM, RF, NT, and TF flags.  
|        |     |      |                      | • If serviced through interrupt gate, clears IF flag.  
|        |     |      |                      | • Clears GS, FS, DS, and ES to 0.  
|        |     |      |                      | • Sets CS and EIP from interrupt gate.  
| 2      | 0   | < 3  | X                    | Interrupt directed to protected-mode general-protection exception (#GP) handler.  
| 3      | 1   | < 3  | 1                    | Interrupt directed to a protected-mode general-protection exception (#GP) handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts.  
| 4      | 1   | 3    | 1                    | Interrupt redirected to protected-mode interrupt handler: (see method 1 processor action).  
| 5      | 1   | 3    | 0                    | Interrupt redirected to 8086 program interrupt handler:  
|        |     |      |                      | • Pushes EFLAGS.  
|        |     |      |                      | • Pushes CS and EIP (lower 16 bits only).  
|        |     |      |                      | • Clears IF flag.  
|        |     |      |                      | • Clears TF flag.  
|        |     |      |                      | • Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task.  
| 6      | 1   | < 3  | 0                    | Interrupt redirected to 8086 program interrupt handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts:  
|        |     |      |                      | • Pushes EFLAGS with IOPL set to 3 and VIF copied to IF.  
|        |     |      |                      | • Pushes CS and EIP (lower 16 bits only).  
|        |     |      |                      | • Clears the VIF flag.  
|        |     |      |                      | • Clears TF flag.  
|        |     |      |                      | • Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task.  

**NOTE:**

* When set to 0, software interrupt is redirected back to the 8086 program interrupt handler; when set to 1, interrupt is directed to protected-mode handler.
Redirecting software interrupts back to the 8086 program potentially speeds up interrupt handling because a switch back and forth between virtual-8086 mode and protected mode is not required. This latter interrupt-handling technique is particularly useful for 8086 operating systems (such as MS-DOS) that use the INT $n$ instruction to call operating system procedures.

The CPUID instruction can be used to verify that the virtual mode extension is implemented on the processor. Bit 1 of the feature flags register (EDX) indicates the availability of the virtual mode extension (see “CPUID—CPU Identification” in Chapter 3, “Instruction Set Reference, A-L,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A).

The following sections describe the six methods (or mechanisms) for handling software interrupts in virtual-8086 mode. See Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism,” for a description of the use of the VIF and VIP flags in the EFLAGS register for handling maskable hardware interrupts.

**21.3.3.1 Method 1: Software Interrupt Handling**

When the VME flag in control register CR4 is clear and the IOPL field is 3, a Pentium or later IA-32 processor handles software interrupts in the same manner as they are handled by an Intel386 or Intel486 processor. It executes an implicit call to the interrupt handler in the protected-mode IDT pointed to by the interrupt vector. See Section 21.3.1, “Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode,” for a complete description of this mechanism and its possible uses.

**21.3.3.2 Methods 2 and 3: Software Interrupt Handling**

When a software interrupt occurs in virtual-8086 mode and the method 2 or 3 conditions are present, the processor generates a general-tection exception (#GP). Method 2 is enabled when the VME flag is set to 0 and the IOPL value is less than 3. Here the IOPL value is used to bypass the protected-mode interrupt handlers and cause any software interrupt that occurs in virtual-8086 mode to be treated as a protected-mode general-tection exception (#GP). The general-protection exception handler calls the virtual-8086 monitor, which can then emulate an 8086-program interrupt handler or pass control back to the 8086 program’s handler, as described in Section 21.3.1.2, “Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler.”

Method 3 is enabled when the VME flag is set to 1, the IOPL value is less than 3, and the corresponding bit for the software interrupt in the software interrupt redirection bit map is set to 1. Here, the processor performs the same
operation as it does for method 2 software interrupt handling. If the corresponding bit for the software interrupt in the software interrupt redirection bit map is set to 0, the interrupt is handled using method 6 (see Section 21.3.3.5, "Method 6: Software Interrupt Handling").

21.3.3.3 Method 4: Software Interrupt Handling

Method 4 handling is enabled when the VME flag is set to 1, the IOPL value is 3, and the bit for the interrupt vector in the redirection bit map is set to 1. Method 4 software interrupt handling allows method 1 style handling when the virtual mode extension is enabled; that is, the interrupt is directed to a protected-mode handler (see Section 21.3.3.1, "Method 1: Software Interrupt Handling").

21.3.3.4 Method 5: Software Interrupt Handling

Method 5 software interrupt handling provides a streamlined method of redirecting software interrupts (invoked with the INT n instruction) that occur in virtual 8086 mode back to the 8086 program's interrupt vector table and its interrupt handlers. Method 5 handling is enabled when the VME flag is set to 1, the IOPL value is 3, and the bit for the interrupt vector in the redirection bit map is set to 0. The processor performs the following actions to make an implicit call to the selected 8086 program interrupt handler:

1. Pushes the low-order 16 bits of the EFLAGS register onto the stack.
2. Pushes the current values of the CS and EIP registers onto the current stack. (Only the 16 least-significant bits of the EIP register are pushed and no stack switch occurs.)
3. Clears the IF flag in the EFLAGS register to disable interrupts.
4. Clears the TF flag in the EFLAGS register.
5. Locates the 8086 program interrupt vector table at linear address 0 for the 8086-mode task.
6. Loads the CS and EIP registers with values from the interrupt vector table entry pointed to by the interrupt vector number. Only the 16 low-order bits of the EIP are loaded and the 16 high-order bits are set to 0. The interrupt vector table is assumed to be at linear address 0 of the current virtual-8086 task.
7. Begins executing the selected interrupt handler.

An IRET instruction at the end of the handler procedure reverses these steps to return program control to the interrupted 8086 program.

Note that with method 5 handling, a mode switch from virtual-8086 mode to protected mode does not occur. The processor remains in virtual-8086 mode throughout the interrupt-handling operation.

The method 5 handling actions are virtually identical to the actions the processor takes when handling software interrupts in real-address mode. The benefit of using method 5 handling to access the 8086 program handlers is that it avoids the overhead of methods 2 and 3 handling, which requires first going to the virtual-8086 monitor, then to the 8086 program handler, then back again to the virtual-8086 monitor, before returning to the interrupted 8086 program (see Section 21.3.1.2, "Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler").

NOTE

Methods 1 and 4 handling can handle a software interrupt in a virtual-8086 task with a regular protected-mode handler, but this approach requires all virtual-8086 tasks to use the same software interrupt handlers, which generally does not give sufficient latitude to the programs running in the virtual-8086 tasks, particularly MS-DOS programs.

21.3.3.5 Method 6: Software Interrupt Handling

Method 6 handling is enabled when the VME flag is set to 1, the IOPL value is less than 3, and the bit for the interrupt or exception vector in the redirection bit map is set to 0. With method 6 interrupt handling, software interrupts are handled in the same manner as was described for method 5 handling (see Section 21.3.3.4, "Method 5: Software Interrupt Handling").
Method 6 differs from method 5 in that with the IOPL value set to less than 3, the VIF and VIP flags in the EFLAGS register are enabled, providing virtual interrupt support for handling class 2 maskable hardware interrupts (see Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism”). These flags provide the virtual-8086 monitor with an efficient means of handling maskable hardware interrupts that occur during a virtual-8086 mode task. Also, because the IOPL value is less than 3 and the VIF flag is enabled, the information pushed on the stack by the processor when invoking the interrupt handler is slightly different between methods 5 and 6 (see Table 21-2).

21.4 PROTECTED-MODE VIRTUAL INTERRUPTS

The IA-32 processors (beginning with the Pentium processor) also support the VIF and VIP flags in the EFLAGS register in protected mode by setting the PVI (protected-mode virtual interrupt) flag in the CR4 register. Setting the PVI flag allows applications running at privilege level 3 to execute the CLI and STI instructions without causing a general-protection exception (#GP) or affecting hardware interrupts.

When the PVI flag is set to 1, the CPL is 3, and the IOPL is less than 3, the STI and CLI instructions set and clear the VIF flag in the EFLAGS register, leaving IF unaffected. In this mode of operation, an application running in protected mode and at a CPL of 3 can inhibit interrupts in the same manner as is described in Section 21.3.2, “Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism,” for a virtual-8086 mode task. When the application executes the CLI instruction, the processor clears the VIF flag. If the processor receives a maskable hardware interrupt, the processor invokes the protected-mode interrupt handler. This handler checks the state of the VIF flag in the EFLAGS register. If the VIF flag is clear (indicating that the active task does not want to have interrupts handled now), the handler sets the VIP flag in the EFLAGS image on the stack and returns to the privilege-level 3 application, which continues program execution. When the application executes a STI instruction to set the VIF flag, the processor automatically invokes the general-protection exception handler, which can then handle the pending interrupt. After handing the pending interrupt, the handler typically sets the VIF flag and clears the VIP flag in the EFLAGS image on the stack and executes a return to the application program. The next time the processor receives a maskable hardware interrupt, the processor will handle it in the normal manner for interrupts received while the processor is operating at a CPL of 3.

If the protected-mode virtual interrupt extension is enabled, CPL = 3, and the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, a general-protection exception is generated.

Because the protected-mode virtual interrupt extension changes only the treatment of EFLAGS.IF (by having CLI and STI update EFLAGS.VIF instead), it affects only the masking of maskable hardware interrupts (interrupt vectors 32 through 255). NMI interrupts and exceptions are handled in the normal manner.

(When protected-mode virtual interrupts are disabled (that is, when the PVI flag in control register CR4 is set to 0, the CPL is less than 3, or the IOPL value is 3), then the CLI and STI instructions execute in a manner compatible with the Intel486 processor. That is, if the CPL is greater (less privileged) than the I/O privilege level (IOPL), a general-protection exception occurs. If the IOPL value is 3, CLI and STI clear or set the IF flag, respectively.)

PUSHF, POPF, IRET, and INT are executed like in the Intel486 processor, regardless of whether protected-mode virtual interrupts are enabled.

It is only possible to enter virtual-8086 mode through a task switch or the execution of an IRET instruction, and it is only possible to leave virtual-8086 mode by faulting to a protected-mode interrupt handler (typically the general-protection exception handler, which in turn calls the virtual 8086-mode monitor). In both cases, the EFLAGS register is saved and restored. This is not true, however, in protected mode when the PVI flag is set and the processor is not in virtual-8086 mode. Here, it is possible to call a procedure at a different privilege level, in which case the EFLAGS register is not saved or modified. However, the states of VIF and VIP flags are never examined by the processor when the CPL is not 3.
Program modules written to run on IA-32 processors can be either 16-bit modules or 32-bit modules. Table 22-1 shows the characteristic of 16-bit and 32-bit modules.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>16-Bit Program Modules</th>
<th>32-Bit Program Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Size</td>
<td>0 to 64 KBytes</td>
<td>0 to 4 GBytes</td>
</tr>
<tr>
<td>Operand Sizes</td>
<td>8 bits and 16 bits</td>
<td>8 bits and 32 bits</td>
</tr>
<tr>
<td>Pointer Offset Size (Address Size)</td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Stack Pointer Size</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
<tr>
<td>Control Transfers Allowed to Code Segments of This Size</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
</tbody>
</table>

The IA-32 processors function most efficiently when executing 32-bit program modules. They can, however, also execute 16-bit program modules, in any of the following ways:

- In real-address mode.
- In virtual-8086 mode.
- System management mode (SMM).
- As a protected-mode task, when the code, data, and stack segments for the task are all configured as a 16-bit segments.
- By integrating 16-bit and 32-bit segments into a single protected-mode task.
- By integrating 16-bit operations into 32-bit code segments.

Real-address mode, virtual-8086 mode, and SMM are native 16-bit modes. A legacy program assembled and/or compiled to run on an Intel 8086 or Intel 286 processor should run in real-address mode or virtual-8086 mode without modification. Sixteen-bit program modules can also be written to run in real-address mode for handling system initialization or to run in SMM for handling system management functions. See Chapter 21, "8086 Emulation," for detailed information on real-address mode and virtual-8086 mode; see Chapter 32, "System Management Mode," for information on SMM.

This chapter describes how to integrate 16-bit program modules with 32-bit program modules when operating in protected mode and how to mix 16-bit and 32-bit code within 32-bit code segments.

## 22.1 DEFINING 16-BIT AND 32-BIT PROGRAM MODULES

The following IA-32 architecture mechanisms are used to distinguish between and support 16-bit and 32-bit segments and operations:

- The D (default operand and address size) flag in code-segment descriptors.
- The B (default stack size) flag in stack-segment descriptors.
- 16-bit and 32-bit call gates, interrupt gates, and trap gates.
- Operand-size and address-size instruction prefixes.
- 16-bit and 32-bit general-purpose registers.

The D flag in a code-segment descriptor determines the default operand-size and address-size for the instructions of a code segment. (In real-address mode and virtual-8086 mode, which do not use segment descriptors, the default is 16 bits.) A code segment with its D flag set is a 32-bit segment; a code segment with its D flag clear is a 16-bit segment.
The B flag in the stack-segment descriptor specifies the size of stack pointer (the 32-bit ESP register or the 16-bit SP register) used by the processor for implicit stack references. The B flag for all data descriptors also controls upper address range for expand down segments.

When transferring program control to another code segment through a call gate, interrupt gate, or trap gate, the operand size used during the transfer is determined by the type of gate used (16-bit or 32-bit), (not by the D-flag or prefix of the transfer instruction). The gate type determines how return information is saved on the stack (or stacks).

For most efficient and trouble-free operation of the processor, 32-bit programs or tasks should have the D flag in the code-segment descriptor and the B flag in the stack-segment descriptor set, and 16-bit programs or tasks should have these flags clear. Program control transfers from 16-bit segments to 32-bit segments (and vice versa) are handled most efficiently through call, interrupt, or trap gates.

Instruction prefixes can be used to override the default operand size and address size of a code segment. These prefixes can be used in real-address mode as well as in protected mode and virtual-8086 mode. An operand-size or address-size prefix only changes the size for the duration of the instruction.

### 22.2 MIXING 16-BIT AND 32-BIT OPERATIONS WITHIN A CODE SEGMENT

The following two instruction prefixes allow mixing of 32-bit and 16-bit operations within one segment:
- The operand-size prefix (66H)
- The address-size prefix (67H)

These prefixes reverse the default size selected by the D flag in the code-segment descriptor. For example, the processor can interpret the (MOV mem, reg) instruction in any of four ways:
- In a 32-bit code segment:
  - Moves 32 bits from a 32-bit register to memory using a 32-bit effective address.
  - If preceded by an operand-size prefix, moves 16 bits from a 16-bit register to memory using a 32-bit effective address.
  - If preceded by an address-size prefix, moves 32 bits from a 32-bit register to memory using a 16-bit effective address.
  - If preceded by both an address-size prefix and an operand-size prefix, moves 16 bits from a 16-bit register to memory using a 16-bit effective address.
- In a 16-bit code segment:
  - Moves 16 bits from a 16-bit register to memory using a 16-bit effective address.
  - If preceded by an operand-size prefix, moves 32 bits from a 32-bit register to memory using a 16-bit effective address.
  - If preceded by an address-size prefix, moves 16 bits from a 16-bit register to memory using a 32-bit effective address.
  - If preceded by both an address-size prefix and an operand-size prefix, moves 32 bits from a 32-bit register to memory using a 32-bit effective address.

The previous examples show that any instruction can generate any combination of operand size and address size regardless of whether the instruction is in a 16- or 32-bit segment. The choice of the 16- or 32-bit default for a code segment is normally based on the following criteria:
- **Performance** — Always use 32-bit code segments when possible. They run much faster than 16-bit code segments on P6 family processors, and somewhat faster on earlier IA-32 processors.
- **The operating system the code segment will be running on** — If the operating system is a 16-bit operating system, it may not support 32-bit program modules.
- **Mode of operation** — If the code segment is being designed to run in real-address mode, virtual-8086 mode, or SMM, it must be a 16-bit code segment.
• **Backward compatibility to earlier IA-32 processors** — If a code segment must be able to run on an Intel 8086 or Intel 286 processor, it must be a 16-bit code segment.

## 22.3 SHARING DATA AMONG MIXED-SIZE CODE SEGMENTS

Data segments can be accessed from both 16-bit and 32-bit code segments. When a data segment that is larger than 64 KBytes is to be shared among 16- and 32-bit code segments, the data that is to be accessed from the 16-bit code segments must be located within the first 64 KBytes of the data segment. The reason for this is that 16-bit pointers by definition can only point to the first 64 KBytes of a segment.

A stack that spans less than 64 KBytes can be shared by both 16- and 32-bit code segments. This class of stacks includes:

- Stacks in expand-up segments with the G (granularity) and B (big) flags in the stack-segment descriptor clear.
- Stacks in expand-down segments with the G and B flags clear.
- Stacks in expand-up segments with the G flag set and the B flag clear and where the stack is contained completely within the lower 64 KBytes. (Offsets greater than FFFFH can be used for data, other than the stack, which is not shared.)

See Section 3.4.5, “Segment Descriptors,” for a description of the G and B flags and the expand-down stack type.

The B flag cannot, in general, be used to change the size of stack used by a 16-bit code segment. This flag controls the size of the stack pointer only for implicit stack references such as those caused by interrupts, exceptions, and the PUSH, POP, CALL, and RET instructions. It does not control explicit stack references, such as accesses to parameters or local variables. A 16-bit code segment can use a 32-bit stack only if the code is modified so that all explicit references to the stack are preceded by the 32-bit address-size prefix, causing those references to use 32-bit addressing and explicit writes to the stack pointer are preceded by a 32-bit operand-size prefix.

In 32-bit, expand-down segments, all offsets may be greater than 64 KBytes; therefore, 16-bit code cannot use this kind of stack segment unless the code segment is modified to use 32-bit addressing.

## 22.4 TRANSFERRING CONTROL AMONG MIXED-SIZE CODE SEGMENTS

There are three ways for a procedure in a 16-bit code segment to safely make a call to a 32-bit code segment:

- Make the call through a 32-bit call gate.
- Make a 16-bit call to a 32-bit interface procedure. The interface procedure then makes a 32-bit call to the intended destination.
- Modify the 16-bit procedure, inserting an operand-size prefix before the call, to change it to a 32-bit call.

Likewise, there are three ways for procedure in a 32-bit code segment to safely make a call to a 16-bit code segment:

- Make the call through a 16-bit call gate. Here, the EIP value at the CALL instruction cannot exceed FFFFH.
- Make a 32-bit call to a 16-bit interface procedure. The interface procedure then makes a 16-bit call to the intended destination.
- Modify the 32-bit procedure, inserting an operand-size prefix before the call, changing it to a 16-bit call. Be certain that the return offset does not exceed FFFFH.

These methods of transferring program control overcome the following architectural limitations imposed on calls between 16-bit and 32-bit code segments:

- Pointers from 16-bit code segments (which by default can only be 16 bits) cannot be used to address data or code located beyond FFFFH in a 32-bit segment.
- The operand-size attributes for a CALL and its companion RETURN instruction must be the same to maintain stack coherency. This is also true for implicit calls to interrupt and exception handlers and their companion IRET instructions.
- A 32-bit parameters (particularly a pointer parameter) greater than FFFFH cannot be squeezed into a 16-bit parameter location on a stack.
The size of the stack pointer (SP or ESP) changes when switching between 16-bit and 32-bit code segments. These limitations are discussed in greater detail in the following sections.

### 22.4.1 Code-Segment Pointer Size

For control-transfer instructions that use a pointer to identify the next instruction (that is, those that do not use gates), the operand-size attribute determines the size of the offset portion of the pointer. The implications of this rule are as follows:

- A JMP, CALL, or RET instruction from a 32-bit segment to a 16-bit segment is always possible using a 32-bit operand size, providing the 32-bit pointer does not exceed FFFFH.
- A JMP, CALL, or RET instruction from a 16-bit segment to a 32-bit segment cannot address a destination greater than FFFFH, unless the instruction is given an operand-size prefix.

See Section 22.4.5, “Writing Interface Procedures,” for an interface procedure that can transfer program control from 16-bit segments to destinations in 32-bit segments beyond FFFFH.

### 22.4.2 Stack Management for Control Transfer

Because the stack is managed differently for 16-bit procedure calls than for 32-bit calls, the operand-size attribute of the RET instruction must match that of the CALL instruction (see Figure 22-1). On a 16-bit call, the processor pushes the contents of the 16-bit IP register and (for calls between privilege levels) the 16-bit SP register. The matching RET instruction must also use a 16-bit operand size to pop these 16-bit values from the stack into the 16-bit registers.

A 32-bit CALL instruction pushes the contents of the 32-bit EIP register and (for inter-privilege-level calls) the 32-bit ESP register. Here, the matching RET instruction must use a 32-bit operand size to pop these 32-bit values from the stack into the 32-bit registers. If the two parts of a CALL/RET instruction pair do not have matching operand sizes, the stack will not be managed correctly and the values of the instruction pointer and stack pointer will not be restored to correct values.
While executing 32-bit code, if a call is made to a 16-bit code segment which is at the same or a more privileged level (that is, the DPL of the called code segment is less than or equal to the CPL of the calling code segment) through a 16-bit call gate, then the upper 16-bits of the ESP register may be unreliable upon returning to the 32-bit code segment (that is, after executing a RET in the 16-bit code segment).

When the CALL instruction and its matching RET instruction are in code segments that have D flags with the same values (that is, both are 32-bit code segments or both are 16-bit code segments), the default settings may be used. When the CALL instruction and its matching RET instruction are in segments which have different D-flag settings, an operand-size prefix must be used.

### 22.4.2.1 Controlling the Operand-Size Attribute For a Call

Three things can determine the operand-size of a call:

- The D flag in the segment descriptor for the calling code segment.
- An operand-size instruction prefix.
- The type of call gate (16-bit or 32-bit), if a call is made through a call gate.

When a call is made with a pointer (rather than a call gate), the D flag for the calling code segment determines the operand-size for the CALL instruction. This operand-size attribute can be overridden by prepending an operand-size prefix to the CALL instruction. So, for example, if the D flag for a code segment is set for 16 bits and the operand-size prefix is used with a CALL instruction, the processor will cause the information stored on the stack to
be stored in 32-bit format. If the call is to a 32-bit code segment, the instructions in that code segment will be able to read the stack coherently. Also, a RET instruction from the 32-bit code segment without an operand-size prefix will maintain stack coherency with the 16-bit code segment being returned to.

When a CALL instruction references a call-gate descriptor, the type of call is determined by the type of call gate (16-bit or 32-bit). The offset to the destination in the code segment being called is taken from the gate descriptor; therefore, if a 32-bit call gate is used, a procedure in a 16-bit code segment can call a procedure located more than 64 KBytes from the base of a 32-bit code segment, because a 32-bit call gate uses a 32-bit offset.

Note that regardless of the operand size of the call and how it is determined, the size of the stack pointer used (SP or ESP) is always controlled by the B flag in the stack-segment descriptor currently in use (that is, when B is clear, SP is used, and when B is set, ESP is used).

An unmodified 16-bit code segment that has run successfully on an 8086 processor or in real-mode on a later IA-32 architecture processor will have its D flag clear and will not use operand-size override prefixes. As a result, all CALL instructions in this code segment will use the 16-bit operand-size attribute. Procedures in these code segments can be modified to safely call procedures to 32-bit code segments in either of two ways:

- Relink the CALL instruction to point to 32-bit call gates (see Section 22.4.2.2, "Passing Parameters With a Gate").
- Add a 32-bit operand-size prefix to each CALL instruction.

### 22.4.2.2 Passing Parameters With a Gate

When referencing 32-bit gates with 16-bit procedures, it is important to consider the number of parameters passed in each procedure call. The count field of the gate descriptor specifies the size of the parameter string to copy from the current stack to the stack of a more privileged (numerically lower privilege level) procedure. The count field of a 16-bit gate specifies the number of 16-bit words to be copied, whereas the count field of a 32-bit gate specifies the number of 32-bit doublewords to be copied. The count field for a 32-bit gate must thus be half the size of the number of words being placed on the stack by a 16-bit procedure. Also, the 16-bit procedure must use an even number of words as parameters.

### 22.4.3 Interrupt Control Transfers

A program-control transfer caused by an exception or interrupt is always carried out through an interrupt or trap gate (located in the IDT). Here, the type of the gate (16-bit or 32-bit) determines the operand-size attribute used in the implicit call to the exception or interrupt handler procedure in another code segment.

A 32-bit interrupt or trap gate provides a safe interface to a 32-bit exception or interrupt handler when the exception or interrupt occurs in either a 32-bit or a 16-bit code segment. It is sometimes impractical, however, to place exception or interrupt handlers in 16-bit code segments, because only 16-bit return addresses are saved on the stack. If an exception or interrupt occurs in a 32-bit code segment when the EIP was greater than FFFFH, the 16-bit handler procedure cannot provide the correct return address.

### 22.4.4 Parameter Translation

When segment offsets or pointers (which contain segment offsets) are passed as parameters between 16-bit and 32-bit procedures, some translation is required. If a 32-bit procedure passes a pointer to data located beyond 64 KBytes to a 16-bit procedure, the 16-bit procedure cannot use it. Except for this limitation, interface code can perform any format conversion between 32-bit and 16-bit pointers that may be needed.

Parameters passed by value between 32-bit and 16-bit code also may require translation between 32-bit and 16-bit formats. The form of the translation is application-dependent.

### 22.4.5 Writing Interface Procedures

Placing interface code between 32-bit and 16-bit procedures can be the solution to the following interface problems:
• Allowing procedures in 16-bit code segments to call procedures with offsets greater than FFFFH in 32-bit code segments.
• Matching operand-size attributes between companion CALL and RET instructions.
• Translating parameters (data), including managing parameter strings with a variable count or an odd number of 16-bit words.
• The possible invalidation of the upper bits of the ESP register.

The interface procedure is simplified where these rules are followed.

1. The interface procedure must reside in a 32-bit code segment (the D flag for the code-segment descriptor is set).
2. All procedures that may be called by 16-bit procedures must have offsets not greater than FFFFH.
3. All return addresses saved by 16-bit procedures must have offsets not greater than FFFFH.

The interface procedure becomes more complex if any of these rules are violated. For example, if a 16-bit procedure calls a 32-bit procedure with an entry point beyond FFFFH, the interface procedure will need to provide the offset to the entry point. The mapping between 16- and 32-bit addresses is only performed automatically when a call gate is used, because the gate descriptor for a call gate contains a 32-bit address. When a call gate is not used, the interface code must provide the 32-bit address.

The structure of the interface procedure depends on the types of calls it is going to support, as follows:

• **Calls from 16-bit procedures to 32-bit procedures** — Calls to the interface procedure from a 16-bit code segment are made with 16-bit CALL instructions (by default, because the D flag for the calling code-segment descriptor is clear), and 16-bit operand-size prefixes are used with RET instructions to return from the interface procedure to the calling procedure. Calls from the interface procedure to 32-bit procedures are performed with 32-bit CALL instructions (by default, because the D flag for the interface procedure’s code segment is set), and returns from the called procedures to the interface procedure are performed with 32-bit RET instructions (also by default).

• **Calls from 32-bit procedures to 16-bit procedures** — Calls to the interface procedure from a 32-bit code segment are made with 32-bit CALL instructions (by default), and returns to the calling procedure from the interface procedure are made with 32-bit RET instructions (also by default). Calls from the interface procedure to 16-bit procedures require the CALL instructions to have the operand-size prefixes, and returns from the called procedures to the interface procedure are performed with 16-bit RET instructions (by default).
MIXING 16-BIT AND 32-BIT CODE
Intel 64 and IA-32 processors are binary compatible. Compatibility means that, within limited constraints, programs that execute on previous generations of processors will produce identical results when executed on later processors. The compatibility constraints and any implementation differences between the Intel 64 and IA-32 processors are described in this chapter.

Each new processor has enhanced the software visible architecture from that found in earlier Intel 64 and IA-32 processors. Those enhancements have been defined with consideration for compatibility with previous and future processors. This chapter also summarizes the compatibility considerations for those extensions.

23.1 PROCESSOR FAMILIES AND CATEGORIES

IA-32 processors are referred to in several different ways in this chapter, depending on the type of compatibility information being related, as described in the following:

- **32-bit Processors** — All the IA-32 processors that use a 32-bit architecture, which include the Intel386, Intel486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
- **16-bit Processors** — All the IA-32 processors that use a 16-bit architecture, which include the 8086/88 and Intel 286 processors.
- **P6 Family Processors** — All the IA-32 processors that are based on the P6 microarchitecture, which include the Pentium Pro, Pentium II, and Pentium III processors.
- **Pentium® 4 Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst® microarchitecture.
- **Intel® Pentium® M Processors** — A family of IA-32 processors that are based on the Intel Pentium M processor microarchitecture.
- **Intel® Core™ Duo and Solo Processors** — Families of IA-32 processors that are based on an improved Intel Pentium M processor microarchitecture.
- **Intel® Xeon® Processors** — A family of IA-32 and Intel 64 processors that are based on the Intel NetBurst microarchitecture. This family includes the Intel Xeon processor and the Intel Xeon processor MP based on the Intel NetBurst microarchitecture. Intel Xeon processors 3000, 3100, 3200, 3300, 3200, 5100, 5200, 5300, 5400, 7200, 7300 series are based on Intel Core microarchitectures and support Intel 64 architecture.
- **Pentium® D Processors** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture.
- **Pentium® Processor Extreme Editions** — A family of dual-core Intel 64 processors that provides two processor cores in a physical package. Each core is based on the Intel NetBurst microarchitecture and supports Intel Hyper-Threading Technology.
- **Intel® Core™ 2 Processor family** — A family of Intel 64 processors that are based on the Intel Core microarchitecture. Intel Pentium Dual-Core processors are also based on the Intel Core microarchitecture.
- **Intel Atom® Processors** — A family of IA-32 and Intel 64 processors. 45 nm Intel Atom processors are based on the Intel Atom microarchitecture. 32 nm Intel Atom processors are based on newer microarchitectures including the Silvermont microarchitecture and the Airmont microarchitecture. Each generation of Intel Atom processors can be identified by the CPUID’s DisplayFamily_DisplayModel signature; see Table 2-1 "CPUID Signature Values of DisplayFamily_DisplayModel” in Chapter 2, “Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
23.2 RESERVED BITS

Throughout this manual, certain bits are marked as reserved in many register and memory layout descriptions. When bits are marked as undefined or reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown effect. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers or memory locations that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing them to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

Software written for existing IA-32 processor that handles reserved bits correctly will port to future IA-32 processors without generating protection exceptions.

23.3 ENABLING NEW FUNCTIONS AND MODES

Most of the new control functions defined for the P6 family and Pentium processors are enabled by new mode flags in the control registers (primarily register CR4). This register is undefined for IA-32 processors earlier than the Pentium processor. Attempting to access this register with an Intel486 or earlier IA-32 processor results in an invalid-opcode exception (#UD). Consequently, programs that execute correctly on the Intel486 or earlier IA-32 processor cannot erroneously enable these functions. Attempting to set a reserved bit in register CR4 to a value other than its original value results in a general-protection exception (#GP). So, programs that execute on the P6 family and Pentium processors cannot erroneously enable functions that may be implemented in future IA-32 processors.

The P6 family and Pentium processors do not check for attempts to set reserved bits in model-specific registers; however these bits may be checked on more recent processors. It is the obligation of the software writer to enforce this discipline. These reserved bits may be used in future Intel processors.

23.4 DETECTING THE PRESENCE OF NEW FEATURES THROUGH SOFTWARE

Software can check for the presence of new architectural features and extensions in either of two ways:

1. Test for the presence of the feature or extension. Software can test for the presence of new flags in the EFLAGS register and control registers. If these flags are reserved (meaning not present in the processor executing the test), an exception is generated. Likewise, software can attempt to execute a new instruction, which results in an invalid-opcode exception (#UD) being generated if it is not supported.

2. Execute the CPUID instruction. The CPUID instruction (added to the IA-32 in the Pentium processor) indicates the presence of new features directly.

See Chapter 20, "Processor Identification and Feature Determination," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for detailed information on detecting new processor features and extensions.

23.5 INTEL MMX TECHNOLOGY

The Pentium processor with MMX technology introduced the MMX technology and a set of MMX instructions to the IA-32. The MMX instructions are described in Chapter 9, "Programming with Intel® MMX™ Technology," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D. The MMX technology and MMX instructions are also included in the Pentium II, Pentium III, Pentium 4, and Intel Xeon processors.
23.6 STREAMING SIMD EXTENSIONS (SSE)

The Streaming SIMD Extensions (SSE) were introduced in the Pentium III processor. The SSE extensions consist of a new set of instructions and a new set of registers. The new registers include the eight 128-bit XMM registers and the 32-bit MXCSR control and status register. These instructions and registers are designed to allow SIMD computations to be made on single precision floating-point numbers. Several of these new instructions also operate in the MMX registers. SSE instructions and registers are described in Section 10, “Programming with Intel® Streaming SIMD Extensions (Intel® SSE),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

23.7 STREAMING SIMD EXTENSIONS 2 (SSE2)

The Streaming SIMD Extensions 2 (SSE2) were introduced in the Pentium 4 and Intel Xeon processors. They consist of a new set of instructions that operate on the XMM and MXCSR registers and perform SIMD operations on double precision floating-point values and on integer values. Several of these new instructions also operate in the MMX registers. SSE2 instructions and registers are described in Chapter 11, “Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

23.8 STREAMING SIMD EXTENSIONS 3 (SSE3)

The Streaming SIMD Extensions 3 (SSE3) were introduced in Pentium 4 processors supporting Intel Hyper-Threading Technology and Intel Xeon processors. SSE3 extensions include 13 instructions. Ten of these 13 instructions support the single instruction multiple data (SIMD) execution model used with SSE/SSE2 extensions. One SSE3 instruction accelerates x87 style programming for conversion to integer. The remaining two instructions (MONITOR and MWAIT) accelerate synchronization of threads. SSE3 instructions are described in Chapter 12, “Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

23.9 ADDITIONAL STREAMING SIMD EXTENSIONS

The Supplemental Streaming SIMD Extensions 3 (SSSE3) were introduced in the Intel Core 2 processor and Intel Xeon processor 5100 series. Streaming SIMD Extensions 4 provided 54 new instructions introduced in 45 nm Intel Xeon processors and Intel Core 2 processors. SSSE3, SSE4.1 and SSE4.2 instructions are described in Chapter 12, “Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, and in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

23.10 INTEL HYPER-THREADING TECHNOLOGY

Intel Hyper-Threading Technology provides two logical processors that can execute two separate code streams (called threads) concurrently by using shared resources in a single processor core or in a physical package.

This feature was introduced in the Intel Xeon processor MP and later steppings of the Intel Xeon processor, and Pentium 4 processors supporting Intel Hyper-Threading Technology. The feature is also found in the Pentium processor Extreme Edition. See also: Section 9.7, “Intel® Hyper-Threading Technology Architecture.”

45 nm and 32 nm Intel Atom processors support Intel Hyper-Threading Technology.

Intel Atom processors based on Silvermont and Airmont microarchitectures do not support Intel Hyper-Threading Technology.
23.11 MULTI-CORE TECHNOLOGY

The Pentium D processor and Pentium processor Extreme Edition provide two processor cores in each physical processor package. See also: Section 9.5, “Intel® Hyper-Threading Technology and Intel® Multi-Core Technology,” and Section 9.8, “Multi-Core Architecture.” Intel Core 2 Duo, Intel Pentium Dual-Core processors, Intel Xeon processors 3000, 3100, 5100, 5200 series provide two processor cores in each physical processor package. Intel Core 2 Extreme, Intel Core 2 Quad processors, Intel Xeon processors 3200, 3300, 5300, 5400, 7300 series provide two processor cores in each physical processor package.

23.12 SPECIFIC FEATURES OF DUAL-CORE PROCESSOR

Dual-core processors may have some processor-specific features. Use CPUID feature flags to detect the availability features. Note the following:

- **CPUID Brand String** — On Pentium processor Extreme Edition, the process will report the correct brand string only after the correct microcode updates are loaded.
- **Enhanced Intel SpeedStep Technology** — This feature is supported in Pentium D processor but not in Pentium processor Extreme Edition.

23.13 NEW INSTRUCTIONS IN THE PENTIUM AND LATER IA-32 PROCESSORS

Table 23-1 identifies the instructions introduced into the IA-32 in the Pentium processor and later IA-32 processors.

### 23.13.1 Instructions Added Prior to the Pentium Processor

The following instructions were added in the Intel486 processor:

- BSWAP (byte swap) instruction.
- XADD (exchange and add) instruction.
- CMPXCHG (compare and exchange) instruction.
- INVD (invalidate cache) instruction.
- WBINVD (write-back and invalidate cache) instruction.
- INVLPD (invalidate TLB entry) instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CPUID Identification Bits</th>
<th>Introduced In</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOVcc (conditional move)</td>
<td>EDX, Bit 15</td>
<td>Pentium Pro processor</td>
</tr>
<tr>
<td>FCMOVcc (floating-point conditional move)</td>
<td>EDX, Bits 0 and 15</td>
<td></td>
</tr>
<tr>
<td>FCOMI (floating-point compare and set EFLAGS)</td>
<td>EDX, Bits 0 and 15</td>
<td></td>
</tr>
<tr>
<td>RDPMC (read performance monitoring counters)</td>
<td>EAX, Bits 8-11, set to 6H; see Note 1</td>
<td></td>
</tr>
<tr>
<td>UD2 (undefined)</td>
<td>EAX, Bits 8-11, set to 6H</td>
<td></td>
</tr>
</tbody>
</table>
The following instructions were added in the Intel386 processor:

- LSS, LFS, and LGS (load SS, FS, and GS registers).
- Long-displacement conditional jumps.
- Single-bit instructions.
- Bit scan instructions.
- Double-shift instructions.
- Byte set on condition instruction.
- Move with sign/zero extension.
- Generalized multiply instruction.
- MOV to and from control registers.
- MOV to and from test registers (now obsolete).
- MOV to and from debug registers.
- RSM (resume from SMM). This instruction was introduced in the Intel386 SL and Intel486 SL processors.

The following instructions were added in the Intel 387 math coprocessor:

- FPREM1.
- FUCOM, FUCOMP, and FUCOMPP.

### OBSOLETE INSTRUCTIONS

The MOV to and from test registers instructions were removed from the Pentium processor and future IA-32 processors. Execution of these instructions generates an invalid-opcode exception (#UD).

### UNDEFINED OPCODES

All new instructions defined for Intel 64 and IA-32 processors use binary encodings that were reserved on earlier-generation processors. Generally, attempting to execute a reserved opcode results in an invalid-opcode (#UD) exception being generated. Consequently, programs that execute correctly on earlier-generation processors cannot erroneously execute these instructions and thereby produce unexpected results when executed on later Intel 64 processors.

For compatibility with prior generations, there are a few reserved opcodes which do not result in a #UD but rather result in the same behavior as certain defined instructions. In the interest of standardization, it is recommended...
that software not use the opcodes given below but instead use those defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

The following items enumerate those reserved opcodes (referring in some cases to opcode groups as defined in Appendix A, "Opcode Map," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D).

- **Immediate Group 1** - When not in 64-bit mode, instructions encoded with opcode 82H result in the behavior of the corresponding instructions encoded with opcode 80H. Depending on the Op/Reg field of the ModR/M Byte, these opcodes are the byte forms of ADD, OR, ADC, SBB, AND, SUB, XOR, CMP. (In 64-bit mode, these opcodes cause a #UD.)

- **Shift Group 2 /6** - Instructions encoded with opcodes C0H, C1H, D0H, D1H, D2H, and D3H with value 110B in the Op/Reg field (/6) of the ModR/M Byte result in the behavior of the corresponding instructions with value 100B in the Op/Reg field (/4). These are various forms of the SAL/SHL instruction.

- **Unary Group 3 /1** - Instructions encoded with opcodes F6H and F7H with value 001B in the Op/Reg field (/01) of the ModR/M Byte result in the behavior of the corresponding instructions with value 000B in the Op/Reg field (/0). These are various forms of the TEST instruction.

- **Reserved NOP** - Instructions encoded with the opcode 0F0DH or with the opcodes 0F18H through 0F1FH result in the behavior of the NOP (No Operation) instruction, except for those opcodes defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D. The opcodes not so defined are considered “Reserved NOP” and may be used for future instructions which have no defined impact on existing architectural state. These reserved NOP opcodes are decoded with a ModR/M byte and typical instruction prefix options but still result in the behavior of the NOP instruction.

- **x87 Opcodes** - There are several groups of x87 opcodes which provide the same behavior as other x87 instructions. See Section 23.18.9 for the complete list.

There are a few reserved opcodes that provide unique behavior but do not provide capabilities that are not already available in the main instructions defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

- **D6H** - When not in 64-bit mode SALC - Set AL to Carry flag. IF (CF=1), AL=FF, ELSE, AL=0 (#UD in 64-bit mode)

- **x87 Opcodes** - There are a few x87 opcodes with subtly different behavior from existing x87 instructions. See Section 23.18.9 for details.

### 23.16 NEW FLAGS IN THE EFLAGS REGISTER

The section titled “EFLAGS Register” in Chapter 3, “Basic Execution Environment,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, shows the configuration of flags in the EFLAGS register for the P6 family processors. No new flags have been added to this register in the P6 family processors. The flags added to this register in the Pentium and Intel486 processors are described in the following sections.

The following flags were added to the EFLAGS register in the Pentium processor:

- VIF (virtual interrupt flag), bit 19.
- VIP (virtual interrupt pending), bit 20.
- ID (identification flag), bit 21.

The AC flag (bit 18) was added to the EFLAGS register in the Intel486 processor.

#### 23.16.1 Using EFLAGS Flags to Distinguish Between 32-Bit IA-32 Processors

The following bits in the EFLAGS register that can be used to differentiate between the 32-bit IA-32 processors:

- Bit 18 (the AC flag) can be used to distinguish an Intel386 processor from the P6 family, Pentium, and Intel486 processors. Since it is not implemented on the Intel386 processor, it will always be clear.
- Bit 21 (the ID flag) indicates whether an application can execute the CPUID instruction. The ability to set and clear this bit indicates that the processor is a P6 family or Pentium processor. The CPUID instruction can then be used to determine which processor.
• Bits 19 (the VIF flag) and 20 (the VIP flag) will always be zero on processors that do not support virtual mode extensions, which includes all 32-bit processors prior to the Pentium processor.

See Chapter 20, “Processor Identification and Feature Determination,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on identifying processors.

23.17 STACK OPERATIONS AND USER SOFTWARE

This section identifies the differences in stack implementation between the various IA-32 processors.

23.17.1 PUSH SP

The P6 family, Pentium, Intel486, Intel386, and Intel 286 processors push a different value on the stack for a PUSH SP instruction than the 8086 processor. The 32-bit processors push the value of the SP register before it is decremented as part of the push operation; the 8086 processor pushes the value of the SP register after it is decremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV BP, SP
XCHG BP, [BP]
```

This code functions as the 8086 processor PUSH SP instruction on the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors.

23.17.2 EFLAGS Pushed on the Stack

The setting of the stored values of bits 12 through 15 (which includes the IOPL field and the NT flag) in the EFLAGS register by the PUSHF instruction, by interrupts, and by exceptions is different with the 32-bit IA-32 processors than with the 8086 and Intel 286 processors. The differences are as follows:

• 8086 processor—bits 12 through 15 are always set.
• Intel 286 processor—bits 12 through 15 are always cleared in real-address mode.
• 32-bit processors in real-address mode—bit 15 (reserved) is always cleared, and bits 12 through 14 have the last value loaded into them.

23.18 X87 FPU

This section addresses the issues that must be faced when porting floating-point software designed to run on earlier IA-32 processors and math coprocessors to a Pentium 4, Intel Xeon, P6 family, or Pentium processor with integrated x87 FPU. To software, a Pentium 4, Intel Xeon, or P6 family processor looks very much like a Pentium processor. Floating-point software which runs on a Pentium or Intel486 DX processor, or on an Intel486 SX processor/Intel 487 SX math coprocessor system or an Intel386 processor/Intel 387 math coprocessor system, will run with at most minor modifications on a Pentium 4, Intel Xeon, or P6 family processor. To port code directly from an Intel 286 processor/Intel 287 math coprocessor system or an Intel 8086 processor/8087 math coprocessor system to a Pentium 4, Intel Xeon, P6 family, or Pentium processor, certain additional issues must be addressed.

In the following sections, the term “32-bit x87 FPUs” refers to the P6 family, Pentium, and Intel486 DX processors, and to the Intel 487 SX and Intel 387 math coprocessors; the term “16-bit IA-32 math coprocessors” refers to the Intel 287 and 8087 math coprocessors.
23.18.1 Control Register CR0 Flags

The ET, NE, and MP flags in control register CR0 control the interface between the integer unit of an IA-32 processor and either its internal x87 FPU or an external math coprocessor. The effect of these flags in the various IA-32 processors is described in the following paragraphs.

The ET (extension type) flag (bit 4 of the CR0 register) is used in the Intel386 processor to indicate whether the math coprocessor in the system is an Intel 287 math coprocessor (flag is clear) or an Intel 387 DX math coprocessor (flag is set). This bit is hardwired to 1 in the P6 family, Pentium, and Intel486 processors.

The NE (Numeric Exception) flag (bit 5 of the CR0 register) is used in the P6 family, Pentium, and Intel486 processors to determine whether unmasked floating-point exceptions are reported internally through interrupt vector 16 (flag is set) or externally through an external interrupt (flag is clear). On a hardware reset, the NE flag is initialized to 0, so software using the automatic internal error-reporting mechanism must set this flag to 1. This flag is nonexistent on the Intel386 processor.

As on the Intel 286 and Intel386 processors, the MP (monitor coprocessor) flag (bit 1 of register CR0) determines whether the WAIT/FWAIT instructions or waiting-type floating-point instructions trap when the context of the x87 FPU is different from that of the currently-executing task. If the MP and TS flag are set, then a WAIT/FWAIT instruction and waiting instructions will cause a device-not-available exception (interrupt vector 7). The MP flag is used on the Intel 286 and Intel386 processors to support the use of a WAIT/FWAIT instruction to wait on a device other than a math coprocessor. The device reports its status through the BUSY# pin. Since the P6 family, Pentium, and Intel486 processors do not have such a pin, the MP flag has no relevant use and should be set to 1 for normal operation.

23.18.2 x87 FPU Status Word

This section identifies differences to the x87 FPU status word for the different IA-32 processors and math coprocessors, the reason for the differences, and their impact on software.

23.18.2.1 Condition Code Flags (C0 through C3)

The following information pertains to differences in the use of the condition code flags (C0 through C3) located in bits 8, 9, 10, and 14 of the x87 FPU status word.

After execution of an FINIT instruction or a hardware reset on a 32-bit x87 FPU, the condition code flags are set to 0. The same operations on a 16-bit IA-32 math coprocessor leave these flags intact (they contain their prior value). This difference in operation has no impact on software and provides a consistent state after reset.

Transcendental instruction results in the core range of the P6 family and Pentium processors may differ from the Intel486 DX processor and Intel 487 SX math coprocessor by 2 to 3 units in the last place (ulp)—(see “Transcendental Instruction Accuracy” in Chapter 8, “Programming with the x87 FPU,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). As a result, the value saved in the C1 flag may also differ.

After an incomplete FPREM/FPREM1 instruction, the C0, C1, and C3 flags are set to 0 on the 32-bit x87 FPUs. After the same operation on a 16-bit IA-32 math coprocessor, these flags are left intact.

On the 32-bit x87 FPUs, the C2 flag serves as an incomplete flag for the FTAN instruction. On the 16-bit IA-32 math coprocessors, the C2 flag is undefined for the FPTAN instruction. This difference has no impact on software, because Intel 287 or 8087 programs do not check C2 after an FPTAN instruction. The use of this flag on later processors allows fast checking of operand range.

23.18.2.2 Stack Fault Flag

When unmasked stack overflow or underflow occurs on a 32-bit x87 FPU, the IE flag (bit 0) and the SF flag (bit 6) of the x87 FPU status word are set to indicate a stack fault and condition code flag C1 is set or cleared to indicate overflow or underflow, respectively. When unmasked stack overflow or underflow occurs on a 16-bit IA-32 math coprocessor, only the IE flag is set. Bit 6 is reserved on these processors. The addition of the SF flag on a 32-bit x87 FPU has no impact on software. Existing exception handlers need not change, but may be upgraded to take advantage of the additional information.
23.18.3 x87 FPU Control Word

Only affine closure is supported for infinity control on a 32-bit x87 FPU. The infinity control flag (bit 12 of the x87 FPU control word) remains programmable on these processors, but has no effect. This change was made to conform to the IEEE Standard 754 for Floating-Point Arithmetic. On a 16-bit IA-32 math coprocessor, both affine and projective closures are supported, as determined by the setting of bit 12. After a hardware reset, the default value of bit 12 is projective. Software that requires projective infinity arithmetic may give different results.

23.18.4 x87 FPU Tag Word

When loading the tag word of a 32-bit x87 FPU, using an FLDENV, FRSTOR, or FXRSTOR (Pentium III processor only) instruction, the processor examines the incoming tag and classifies the location only as empty or non-empty. Thus, tag values of 00, 01, and 10 are interpreted by the processor to indicate a non-empty location. The tag value of 11 is interpreted by the processor to indicate an empty location. Subsequent operations on a non-empty register always examine the value in the register, not the value in its tag. The FSTENV, FSAVE, and FXSAVE (Pentium III processor only) instructions examine the non-empty registers and put the correct values in the tags before storing the tag word.

The corresponding tag for a 16-bit IA-32 math coprocessor is checked before each register access to determine the class of operand in the register; the tag is updated after every change to a register so that the tag always reflects the most recent status of the register. Software can load a tag with a value that disagrees with the contents of a register (for example, the register contains a valid value, but the tag says special). Here, the 16-bit IA-32 math coprocessors honor the tag and do not examine the register.

Software written to run on a 16-bit IA-32 math coprocessor may not operate correctly on a 16-bit x87 FPU, if it uses the FLDENV, FRSTOR, or FXRSTOR instructions to change tags to values (other than to empty) that are different from actual register contents.

The encoding in the tag word for the 32-bit x87 FPUs for unsupported data formats (including pseudo-zero and unnormal) is special (10B), to comply with IEEE Standard 754. The encoding in the 16-bit IA-32 math coprocessors for pseudo-zero and unnormal is valid (00B) and the encoding for other unsupported data formats is special (10B). Code that recognizes the pseudo-zero or unnormal format as valid must therefore be changed if it is ported to a 32-bit x87 FPU.

23.18.5 Data Types

This section discusses the differences of data types for the various x87 FPUs and math coprocessors.

23.18.5.1 NaNs

The 32-bit x87 FPUs distinguish between signaling NaNs (SNaNs) and quiet NaNs (QNaNs). These x87 FPUs only generate QNaNs and normally do not generate an exception upon encountering a QNaN. An invalid-operation exception (#I) is generated only upon encountering a SNaN, except for the FCOM, FIST, and FBSTP instructions, which also generates an invalid-operation exceptions for a QNaNs. This behavior matches IEEE Standard 754.

The 16-bit IA-32 math coprocessors only generate one kind of NaN (the equivalent of a QNaN), but the raise an invalid-operation exception upon encountering any kind of NaN.

When porting software written to run on a 16-bit IA-32 math coprocessor to a 32-bit x87 FPU, uninitialized memory locations that contain QNaNs should be changed to SNaNs to cause the x87 FPU or math coprocessor to fault when uninitialized memory locations are referenced.

23.18.5.2 Pseudo-zero, Pseudo-NaN, Pseudo-infinity, and Unnormal Formats

The 32-bit x87 FPUs neither generate nor support the pseudo-zero, pseudo-NaN, pseudo-infinity, and unnormal formats. Whenever they encounter them in an arithmetic operation, they raise an invalid-operation exception. The 16-bit IA-32 math coprocessors define and support special handling for these formats. Support for these formats was dropped to conform with IEEE Standard 754 for Floating-Point Arithmetic.
This change should not impact software ported from 16-bit IA-32 math coprocessors to 32-bit x87 FPUs. The 32-bit x87 FPUs do not generate these formats, and therefore will not encounter them unless software explicitly loads them in the data registers. The only affect may be in how software handles the tags in the tag word (see also: Section 23.18.4, "x87 FPU Tag Word").

23.18.6 Floating-Point Exceptions

This section identifies the implementation differences in exception handling for floating-point instructions in the various x87 FPUs and math coprocessors.

23.18.6.1 Denormal Operand Exception (#D)

When the denormal operand exception is masked, the 32-bit x87 FPUs automatically normalize denormalized numbers when possible; whereas, the 16-bit IA-32 math coprocessors return a denormal result. A program written to run on a 16-bit IA-32 math coprocessor that uses the denormal exception solely to normalize denormalized operands is redundant when run on the 32-bit x87 FPUs. If such a program is run on 32-bit x87 FPUs, performance can be improved by masking the denormal exception. Floating-point programs run faster when the FPU performs normalization of denormalized operands.

The denormal operand exception is not raised for transcendental instructions and the FXTRACT instruction on the 16-bit IA-32 math coprocessors. This exception is raised for these instructions on the 32-bit x87 FPUs. The exception handlers ported to these latter processors need to be changed only if the handlers gives special treatment to different opcodes.

23.18.6.2 Numeric Overflow Exception (#O)

On the 32-bit x87 FPUs, when the numeric overflow exception is masked and the rounding mode is set to chop (toward 0), the result is the largest positive or smallest negative number. The 16-bit IA-32 math coprocessors do not signal the overflow exception when the masked response is not ∞; that is, they signal overflow only when the rounding control is not set to round to 0. If rounding is set to chop (toward 0), the result is positive or negative ∞. Under the most common rounding modes, this difference has no impact on existing software.

If rounding is toward 0 (chop), a program on a 32-bit x87 FPU produces, under overflow conditions, a result that is different in the least significant bit of the significand, compared to the result on a 16-bit IA-32 math coprocessor. The reason for this difference is IEEE Standard 754 compatibility.

When the overflow exception is not masked, the precision exception is flagged on the 32-bit x87 FPUs. When the result is stored in the stack, the significand is rounded according to the precision control (PC) field of the FPU control word or according to the opcode. On the 16-bit IA-32 math coprocessors, the precision exception is not flagged and the significand is not rounded. The impact on existing software is that if the result is stored on the stack, a program running on a 32-bit x87 FPU produces a different result under overflow conditions than on a 16-bit IA-32 math coprocessor. The difference is apparent only to the exception handler. This difference is for IEEE Standard 754 compatibility.

23.18.6.3 Numeric Underflow Exception (#U)

When the underflow exception is masked on the 32-bit x87 FPUs, the underflow exception is signaled when the result is tiny and inexact (see Section 4.9.1.5, "Numeric Underflow Exception (#U)," in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). When the underflow exception is unmasked and the instruction is supposed to store the result on the stack, the significand is rounded to the appropriate precision (according to the PC flag in the FPU control word, for those instructions controlled by PC, otherwise to extended precision), after adjusting the exponent.

23.18.6.4 Exception Precedence

There is no difference in the precedence of the denormal-operand exception on the 32-bit x87 FPUs, whether it be masked or not. When the denormal-operand exception is not masked on the 16-bit IA-32 math coprocessors, it takes precedence over all other exceptions. This difference causes no impact on existing software, but some
unneeded normalization of denormalized operands is prevented on the Intel486 processor and Intel 387 math coprocessor.

23.18.6.5 CS and EIP For FPU Exceptions
On the Intel 32-bit x87 FPUs, the values from the CS and EIP registers saved for floating-point exceptions point to any prefixes that come before the floating-point instruction. On the 8087 math coprocessor, the saved CS and IP registers points to the floating-point instruction.

23.18.6.6 FPU Error Signals
The floating-point error signals to the P6 family, Pentium, and Intel486 processors do not pass through an interrupt controller; an INT# signal from an Intel 387, Intel 287 or 8087 math coprocessors does. If an 8086 processor uses another exception for the 8087 interrupt, both exception vectors should call the floating-point-error exception handler. Some instructions in a floating-point-error exception handler may need to be deleted if they use the interrupt controller. The P6 family, Pentium, and Intel486 processors have signals that, with the addition of external logic, support reporting for emulation of the interrupt mechanism used in many personal computers.

On the P6 family, Pentium, and Intel486 processors, an undefined floating-point opcode will cause an invalid-opcode exception (#UD, interrupt vector 6). Undefined floating-point opcodes, like legal floating-point opcodes, cause a device not available exception (#NM, interrupt vector 7) when either the TS or EM flag in control register CR0 is set. The P6 family, Pentium, and Intel486 processors do not check for floating-point error conditions on encountering an undefined floating-point opcode.

23.18.6.7 Assertion of the FERR# Pin
When using the MS-DOS compatibility mode for handing floating-point exceptions, the FERR# pin must be connected to an input to an external interrupt controller. An external interrupt is then generated when the FERR# output drives the input to the interrupt controller and the interrupt controller in turn drives the INTR pin on the processor.

For the P6 family and Intel386 processors, an unmasked floating-point exception always causes the FERR# pin to be asserted upon completion of the instruction that caused the exception. For the Pentium and Intel486 processors, an unmasked floating-point exception may cause the FERR# pin to be asserted either at the end of the instruction causing the exception or immediately before execution of the next floating-point instruction. (Note that the next floating-point instruction would not be executed until the pending unmasked exception has been handled.) See Appendix D, “Guidelines for Writing SIMD Floating-Point Exception Handlers,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for a complete description of the required mechanism for handling floating-point exceptions using the MS-DOS compatibility mode.

Using FERR# and IGNNE# to handle floating-point exception is deprecated by modern operating systems; this approach also limits newer processors to operate with one logical processor active.

23.18.6.8 Invalid Operation Exception On Denormals
An invalid-operation exception is not generated on the 32-bit x87 FPUs upon encountering a denormal value when executing a FSQRT, FDIV, or FPREM instruction or upon conversion to BCD or to integer. The operation proceeds by first normalizing the value. On the 16-bit IA-32 math coprocessors, upon encountering this situation, the invalid-operation exception is generated. This difference has no impact on existing software. Software running on the 32-bit x87 FPUs continues to execute in cases where the 16-bit IA-32 math coprocessors trap. The reason for this change was to eliminate an exception from being raised.

23.18.6.9 Alignment Check Exceptions (#AC)
If alignment checking is enabled, a misaligned data operand on the P6 family, Pentium, and Intel486 processors causes an alignment check exception (#AC) when a program or procedure is running at privilege-level 3, except for the stack portion of the FSAVE/FNSAVE, FXSAVE, FRSTOR, and FXRSTOR instructions.
23.18.6.10 Segment Not Present Exception During FLDENV
On the Intel486 processor, when a segment not present exception (#NP) occurs in the middle of an FLDENV instruction, it can happen that part of the environment is loaded and part not. In such cases, the FPU control word is left with a value of 007FH. The P6 family and Pentium processors ensure the internal state is correct at all times by attempting to read the first and last bytes of the environment before updating the internal state.

23.18.6.11 Device Not Available Exception (#NM)
The device-not-available exception (#NM, interrupt 7) will occur in the P6 family, Pentium, and Intel486 processors as described in Section 2.5, "Control Registers," Table 2-2, and Chapter 6, "Interrupt 7—Device Not Available Exception (#NM)."

23.18.6.12 Coprocessor Segment Overrun Exception
The coprocessor segment overrun exception (interrupt 9) does not occur in the P6 family, Pentium, and Intel486 processors. In situations where the Intel 387 math coprocessor would cause an interrupt 9, the P6 family, Pentium, and Intel486 processors simply abort the instruction. To avoid undetected segment overruns, it is recommended that the floating-point save area be placed in the same page as the TSS. This placement will prevent the FPU environment from being lost if a page fault occurs during the execution of an FLDENV, FRSTOR, or FXRSTOR instruction while the operating system is performing a task switch.

23.18.6.13 General Protection Exception (#GP)
A general-protection exception (#GP, interrupt 13) occurs if the starting address of a floating-point operand falls outside a segment's size. An exception handler should be included to report these programming errors.

23.18.6.14 Floating-Point Error Exception (#MF)
In real mode and protected mode (not including virtual-8086 mode), interrupt vector 16 must point to the floating-point exception handler. In virtual-8086 mode, the virtual-8086 monitor can be programmed to accommodate a different location of the interrupt vector for floating-point exceptions.

23.18.7 Changes to Floating-Point Instructions
This section identifies the differences in floating-point instructions for the various Intel FPU and math coprocessor architectures, the reason for the differences, and their impact on software.

23.18.7.1 FDIV, FPREM, and FSQRT Instructions
The 32-bit x87 FPUs support operations on denormalized operands and, when detected, an underflow exception can occur, for compatibility with the IEEE Standard 754. The 16-bit IA-32 math coprocessors do not operate on denormalized operands or return underflow results. Instead, they generate an invalid-operation exception when they detect an underflow condition. An existing underflow exception handler will require change only if it gives different treatment to different opcodes. Also, it is possible that fewer invalid-operation exceptions will occur.

23.18.7.2 FSSCALE Instruction
With the 32-bit x87 FPUs, the range of the scaling operand is not restricted. If (0 < | ST(1) | < 1), the scaling factor is 0; therefore, ST(0) remains unchanged. If the rounded result is not exact or if there was a loss of accuracy (masked underflow), the precision exception is signaled. With the 16-bit IA-32 math coprocessors, the range of the scaling operand is restricted. If (0 < | ST(1) | < 1), the result is undefined and no exception is signaled. The impact of this difference on exiting software is that different results are delivered on the 32-bit and 16-bit FPUs and math coprocessors when (0 < | ST(1) | < 1).
23.18.7.3  FPREM1 Instruction
The 32-bit x87 FPUs compute a partial remainder according to IEEE Standard 754. This instruction does not exist on the 16-bit IA-32 math coprocessors. The availability of the FPREM1 instruction has no impact on existing software.

23.18.7.4  FPREM Instruction
On the 32-bit x87 FPUs, the condition code flags C0, C3, C1 in the status word correctly reflect the three low-order bits of the quotient following execution of the FPREM instruction. On the 16-bit IA-32 math coprocessors, the quotient bits are incorrect when performing a reduction of \((64^N + M)\) when \(N \geq 1\) and \(M = 1\) or 2. This difference does not affect existing software; software that works around the bug should not be affected.

23.18.7.5  FUCOM, FUCOMP, and FUCOMPP Instructions
When executing the FUCOM, FUCOMP, and FUCOMPP instructions, the 32-bit x87 FPUs perform unordered compare according to IEEE Standard 754. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these new instructions has no impact on existing software.

23.18.7.6  FPTAN Instruction
On the 32-bit x87 FPUs, the range of the operand for the FPTAN instruction is much less restricted \(|ST(0)| < 2^{63}\) than on earlier math coprocessors. The instruction reduces the operand internally using an internal \(\pi/4\) constant that is more accurate. The range of the operand is restricted to \(|ST(0)| < \pi/4\) on the 16-bit IA-32 math coprocessors; the operand must be reduced to this range using FPREM. This change has no impact on existing software. See also sections 8.3.8 and section 8.3.10 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on the accuracy of the FPTAN instruction.

23.18.7.7  Stack Overflow
On the 32-bit x87 FPUs, if an FPU stack overflow occurs when the invalid-operation exception is masked, the FPU returns the real, integer, or BCD-integer indefinite value to the destination operand, depending on the instruction being executed. On the 16-bit IA-32 math coprocessors, the original operand remains unchanged following a stack overflow, but it is loaded into register ST(1). This difference has no impact on existing software.

23.18.7.8  FSIN, FCOS, and FSINCOS Instructions
On the 32-bit x87 FPUs, these instructions perform three common trigonometric functions. These instructions do not exist on the 16-bit IA-32 math coprocessors. The availability of these instructions has no impact on existing software, but using them provides a performance upgrade. See also sections 8.3.8 and section 8.3.10 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on the accuracy of the FSIN, FCOS, and FSINCOS instructions.

23.18.7.9  FPATAN Instruction
On the 32-bit x87 FPUs, the range of operands for the FPATAN instruction is unrestricted. On the 16-bit IA-32 math coprocessors, the absolute value of the operand in register ST(0) must be smaller than the absolute value of the operand in register ST(1). This difference has impact on existing software.

23.18.7.10  F2XM1 Instruction
The 32-bit x87 FPUs support a wider range of operands \((-1 < ST(0) < +1)\) for the F2XM1 instruction. The supported operand range for the 16-bit IA-32 math coprocessors is \(0 \leq ST(0) \leq 0.5\). This difference has no impact on existing software.
23.18.7.11  FLD Instruction

On the 32-bit x87 FPUs, when using the FLD instruction to load an extended-real value, a denormal-operand exception is not generated because the instruction is not arithmetic. The 16-bit IA-32 math coprocessors do report a denormal-operand exception in this situation. This difference does not affect existing software.

On the 32-bit x87 FPUs, loading a denormal value that is in single- or double-real format causes the value to be converted to extended-real format. Loading a denormal value on the 16-bit IA-32 math coprocessors causes the value to be converted to an unnormal. If the next instruction is FXTRACT or FXAM, the 32-bit x87 FPUs will give a different result than the 16-bit IA-32 math coprocessors. This change was made for IEEE Standard 754 compatibility.

On the 32-bit x87 FPUs, loading an SNaN that is in single- or double-real format causes the FPU to generate an invalid-operation exception. The 16-bit IA-32 math coprocessors do not raise an exception when loading a signaling NaN. The invalid-operation exception handler for 16-bit math coprocessor software needs to be updated to handle this condition when porting software to 32-bit FPUs. This change was made for IEEE Standard 754 compatibility.

23.18.7.12  FXTRACT Instruction

On the 32-bit x87 FPUs, if the operand is 0 for the FXTRACT instruction, the divide-by-zero exception is reported and \( -\infty \) is delivered to register ST(1). If the operand is \( +\infty \), no exception is reported. If the operand is 0 on the 16-bit IA-32 math coprocessors, 0 is delivered to register ST(1) and no exception is reported. If the operand is \( +\infty \), the invalid-operation exception is reported. These differences have no impact on existing software. Software usually bypasses 0 and \( \infty \). This change is due to the IEEE Standard 754 recommendation to fully support the “logb” function.

23.18.7.13  Load Constant Instructions

On 32-bit x87 FPUs, rounding control is in effect for the load constant instructions. Rounding control is not in effect for the 16-bit IA-32 math coprocessors. Results for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions are the same as for the 16-bit IA-32 math coprocessors when rounding control is set to round to nearest or round to \( +\infty \). They are the same for the FLDL2T instruction when rounding control is set to round to nearest, round to \( -\infty \), or round to zero. Results are different from the 16-bit IA-32 math coprocessors in the least significant bit of the mantissa if rounding control is set to round to \( -\infty \) or round to 0 for the FLDPI, FLDLN2, FLDLG2, and FLDL2E instructions; they are different for the FLDL2T instruction if round to \( +\infty \) is specified. These changes were implemented for compatibility with IEEE Standard 754 for Floating-Point Arithmetic recommendations.

23.18.7.14  FXAM Instruction

With the 32-bit x87 FPUs, if the FPU encounters an empty register when executing the FXAM instruction, it not generate combinations of C0 through C3 equal to 1101 or 1111. The 16-bit IA-32 math coprocessors may generate these combinations, among others. This difference has no impact on existing software; it provides a performance upgrade to provide repeatable results.

23.18.7.15  FSAVE and FSTENV Instructions

With the 32-bit x87 FPUs, the address of a memory operand pointer stored by FSAVE or FSTENV is undefined if the previous floating-point instruction did not refer to memory.

23.18.8  Transcendental Instructions

The floating-point results of the P6 family and Pentium processors for transcendental instructions in the core range may differ from the Intel486 processors by about 2 or 3 ulps (see “Transcendental Instruction Accuracy” in Chapter 8, “Programming with the x87 FPU,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). Condition code flag C1 of the status word may differ as a result. The exact threshold for underflow and overflow will vary by a few ulps. The P6 family and Pentium processors’ results will have a worst case error of less than 1 ulp when rounding to the nearest-even and less than 1.5 ulps when rounding in other modes. The transcendental
instructions are guaranteed to be monotonic, with respect to the input operands, throughout the domain supported by the instruction.

Transcendental instructions may generate different results in the round-up flag (C1) on the 32-bit x87 FPs. The round-up flag is undefined for these instructions on the 16-bit IA-32 math coprocessors. This difference has no impact on existing software.

### 23.18.9 Obsolete Instructions and Undefined Opcodes

The 8087 math coprocessor instructions FENI and FDISI, and the Intel 287 math coprocessor instruction FSETPM are treated as integer NOP instructions in the 32-bit x87 FPs. If these opcodes are detected in the instruction stream, no specific operation is performed and no internal states are affected. FSETPM informed the Intel 287 math coprocessor that the processor was in protected mode. The 32-bit x87 FPs handle all addressing and exception-pointer information, whether in protected mode or not.

For compatibility with prior generations there are a few reserved x87 opcodes which do not result in an invalid-opcode (#UD) exception, but rather result in the same behavior as existing defined x87 instructions. In the interest of standardization, it is recommended that the opcodes defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, be used for these operations for standardization.

- **DC07H through DCD7H** - Behaves the same as FCOM, D8D0H through D8D7H.
- **DCD8H through DCDFH** - Behaves the same as FCOMP, D8D8H through D8DFH.
- **DDC8H through DDCFH** - Behaves the same as FXCH, D9C8H through D9CFH.
- **DED0H through DED7H** - Behaves the same as FCOMP, D8D8H through D8DFH.
- **DFD0H through DFD7H** - Behaves the same as FSTP, DDD8H through DDDFH.
- **DFC8H through DFCFH** - Behaves the same as FXCH, D9C8H through D9CFH.
- **DFD8H through DFDFH** - Behaves the same as FSTP, DDD8H through DDDFH.

There are a few reserved x87 opcodes which provide unique behavior but do not provide capabilities which are not already available in the main instructions defined in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D.

- **D9D8H through D9DFH** - Behaves the same as FSTP (DDD8H through DDDFH) but won’t cause a stack underflow exception.
- **DFC0H through DFC7H** - Behaves the same as FFREE (DDC0H through DDD7H) with the addition of an x87 stack POP.

### 23.18.10 WAIT/FWAIT Prefix Differences

On the Intel486 processor, when a WAIT/FWAIT instruction precedes a floating-point instruction (one which itself automatically synchronizes with the previous floating-point instruction), the WAIT/FWAIT instruction is treated as a no-op. Pending floating-point exceptions from a previous floating-point instruction are processed not on the WAIT/FWAIT instruction but on the floating-point instruction following the WAIT/FWAIT instruction. In such a case, the report of a floating-point exception may appear one instruction later on the Intel486 processor than on a P6 family or Pentium FPU, or on Intel 387 math coprocessor.

### 23.18.11 Operands Split Across Segments and/or Pages

On the P6 family, Pentium, and Intel486 processor FPUs, when the first half of an operand to be written is inside a page or segment and the second half is outside, a memory fault can cause the first half to be stored but not the second half. In this situation, the Intel 387 math coprocessor stores nothing.

### 23.18.12 FPU Instruction Synchronization

On the 32-bit x87 FPs, all floating-point instructions are automatically synchronized; that is, the processor automatically waits until the previous floating-point instruction has completed before completing the next floating-point
instruction. No explicit WAIT/FWAIT instructions are required to assure this synchronization. For the 8087 math coprocessors, explicit waits are required before each floating-point instruction to ensure synchronization. Although 8087 programs having explicit WAIT instructions execute perfectly on the 32-bit IA-32 processors without reassembly, these WAIT instructions are unnecessary.

23.19 SERIALIZING INSTRUCTIONS

Certain instructions have been defined to serialize instruction execution to ensure that modifications to flags, registers, and memory are completed before the next instruction is executed (or in P6 family processor terminology “committed to machine state”). Because the P6 family processors use branch-prediction and out-of-order execution techniques to improve performance, instruction execution is not generally serialized until the results of an executed instruction are committed to machine state (see Chapter 2, “Intel® 64 and IA-32 Architectures,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

As a result, at places in a program or task where it is critical to have execution completed for all previous instructions before executing the next instruction (for example, at a branch, at the end of a procedure, or in multiprocessor dependent code), it is useful to add a serializing instruction. See Section 9.3, “Serializing Instructions,” for more information on serializing instructions.

23.20 FPU AND MATH COPROCESSOR INITIALIZATION

Table 10-1 shows the states of the FPUs in the P6 family, Pentium, Intel486 processors and of the Intel 387 math coprocessor and Intel 287 coprocessor following a power-up, reset, or INIT, or following the execution of an FINIT/FNINIT instruction. The following is some additional compatibility information concerning the initialization of x87 FPUs and math coprocessors.

23.20.1 Intel® 387 and Intel® 287 Math Coprocessor Initialization

Following an Intel386 processor reset, the processor identifies its coprocessor type (Intel® 287 or Intel® 387 DX math coprocessor) by sampling its ERROR# input some time after the falling edge of RESET# signal and before execution of the first floating-point instruction. The Intel 287 coprocessor keeps its ERROR# output in inactive state after hardware reset; the Intel 387 coprocessor keeps its ERROR# output in active state after hardware reset.

Upon hardware reset or execution of the FINIT/FNINIT instruction, the Intel 387 math coprocessor signals an error condition. The P6 family, Pentium, and Intel486 processors, like the Intel 287 coprocessor, do not.

23.20.2 Intel486 SX Processor and Intel 487 SX Math Coprocessor Initialization

When initializing an Intel486 SX processor and an Intel 487 SX math coprocessor, the initialization routine should check the presence of the math coprocessor and should set the FPU related flags (EM, MP, and NE) in control register CR0 accordingly (see Section 2.5, “Control Registers,” for a complete description of these flags). Table 23-2 gives the recommended settings for these flags when the math coprocessor is present. The FSTCW instruction will give a value of FFFFH for the Intel486 SX microprocessor and 037FH for the Intel 487 SX math coprocessor.
ARCHITECTURE COMPATIBILITY

The EM and MP flags in register CR0 are interpreted as shown in Table 23-3.

Table 23-2. Recommended Values of the EM, MP, and NE Flags for Intel486 SX Microprocessor/Intel 487 SX Math Coprocessor System

<table>
<thead>
<tr>
<th>CR0 Flags</th>
<th>Intel486 SX Processor Only</th>
<th>Intel 487 SX Math Coprocessor Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MP</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NE</td>
<td>1</td>
<td>0, for MS-DOS* systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1, for user-defined exception handler</td>
</tr>
</tbody>
</table>

Table 23-3. EM and MP Flag Interpretation

<table>
<thead>
<tr>
<th>EM</th>
<th>MP</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions ignore TS.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Floating-point instructions are passed to FPU; WAIT/FWAIT and other waiting-type instructions test TS.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions ignore TS.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Floating-point instructions trap to emulator; WAIT/FWAIT and other waiting-type instructions test TS.</td>
</tr>
</tbody>
</table>

Following is an example code sequence to initialize the system and check for the presence of Intel486 SX processor/Intel 487 SX math coprocessor.

```
fninit
fstcw mem_loc
mov ax, mem_loc
cmp ax, 037fh
jz Intel487_SX_Math_CoProcessor_present ;ax=037fh
jmp Intel486_SX_microprocessor_present ;ax=ffffh
```

If the Intel 487 SX math coprocessor is not present, the following code can be run to set the CR0 register for the Intel486 SX processor.

```
mov eax, cr0
and eax, fffffffdh ;make MP=0
or eax, 0024h ;make EM=1, NE=1
mov cr0, eax
```

This initialization will cause any floating-point instruction to generate a device not available exception (#NM), interrupt 7. The software emulation will then take control to execute these instructions. This code is not required if an Intel 487 SX math coprocessor is present in the system. In that case, the typical initialization routine for the Intel486 SX microprocessor will be adequate.

Also, when designing an Intel486 SX processor based system with an Intel 487 SX math coprocessor, timing loops should be independent of frequency and clocks per instruction. One way to attain this is to implement these loops in hardware and not in software (for example, BIOS).

23.21 CONTROL REGISTERS

The following sections identify the new control registers and control register flags and fields that were introduced to the 32-bit IA-32 in various processor families. See Figure 2-7 for the location of these flags and fields in the control registers.
The Pentium III processor introduced one new control flag in control register CR4:
• OSXMMEXCPT (bit 10) — The OS will set this bit if it supports unmasked SIMD floating-point exceptions.

The Pentium II processor introduced one new control flag in control register CR4:
• OSFXSR (bit 9) — The OS supports saving and restoring the Pentium III processor state during context switches.

The Pentium Pro processor introduced three new control flags in control register CR4:
• PAE (bit 5) — Physical address extension. Enables paging mechanism to reference extended physical addresses when set; restricts physical addresses to 32 bits when clear (see also: Section 23.22.1.1, “Physical Memory Addressing Extension”).
• PGE (bit 7) — Page global enable. Inhibits flushing of frequently-used or shared pages on CR3 writes (see also: Section 23.22.1.2, “Global Pages”).
• PCE (bit 8) — Performance-monitoring counter enable. Enables execution of the RDPMC instruction at any protection level.

The content of CR4 is 0H following a hardware reset.

Control register CR4 was introduced in the Pentium processor. This register contains flags that enable certain new extensions provided in the Pentium processor:
• VME — Virtual-8086 mode extensions. Enables support for a virtual interrupt flag in virtual-8086 mode (see Section 21.3, ”Interrupt and Exception Handling in Virtual-8086 Mode”).
• PVI — Protected-mode virtual interrupts. Enables support for a virtual interrupt flag in protected mode (see Section 21.4, ”Protected-Mode Virtual Interrupts”).
• TSD — Time-stamp disable. Restricts the execution of the RDTSC instruction to procedures running at privileged level 0.
• DE — Debugging extensions. Causes an undefined opcode (#UD) exception to be generated when debug registers DR4 and DR5 are references for improved performance (see Section 23.23.3, ”Debug Registers DR4 and DR5”).
• PSE — Page size extensions. Enables 4-MByte pages with 32-bit paging when set (see Section 4.3, ”32-Bit Paging”).
• MCE — Machine-check enable. Enables the machine-check exception, allowing exception handling for certain hardware error conditions (see Chapter 16, ”Machine-Check Architecture”).

The Intel486 processor introduced five new flags in control register CR0:
• NE — Numeric error. Enables the normal mechanism for reporting floating-point numeric errors.
• WP — Write protect. Write-protects read-only pages against supervisor-mode accesses.
• AM — Alignment mask. Controls whether alignment checking is performed. Operates in conjunction with the AC (Alignment Check) flag.
• NW — Not write-through. Enables write-throughs and cache invalidation cycles when clear and disables invalidation cycles and write-throughs that hit in the cache when set.
• CD — Cache disable. Enables the internal cache when clear and disables the cache when set.

The Intel486 processor introduced two new flags in control register CR3:
• PCD — Page-level cache disable. The state of this flag is driven on the PCD# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PCD# pin is used to control caching in an external cache on a cycle-by-cycle basis.
• PWT — Page-level write-through. The state of this flag is driven on the PWT# pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. The PWT# pin is used to control write through in an external cache on a cycle-by-cycle basis.
23.22 MEMORY MANAGEMENT FACILITIES

The following sections describe the new memory management facilities available in the various IA-32 processors and some compatibility differences.

23.22.1 New Memory Management Control Flags

The Pentium Pro processor introduced three new memory management features: physical memory addressing extension, the global bit in page-table entries, and general support for larger page sizes. These features are only available when operating in protected mode.

23.22.1.1 Physical Memory Addressing Extension

The new PAE (physical address extension) flag in control register CR4, bit 5, may enable additional address lines on the processor, allowing extended physical addresses. This option can only be used when paging is enabled, using a new page-table mechanism provided to support the larger physical address range (see Section 4.1, “Paging Modes and Control Bits”).

23.22.1.2 Global Pages

The new PGE (page global enable) flag in control register CR4, bit 7, provides a mechanism for preventing frequently used pages from being flushed from the translation lookaside buffer (TLB). When this flag is set, frequently used pages (such as pages containing kernel procedures or common data tables) can be marked global by setting the global flag in a page-directory or page-table entry.

On a task switch or a write to control register CR3 (which normally causes the TLBs to be flushed), the entries in the TLB marked global are not flushed. Marking pages global in this manner prevents unnecessary reloading of the TLB due to TLB misses on frequently used pages. See Section 4.10, “Caching Translation Information,” for a detailed description of this mechanism.

23.22.1.3 Larger Page Sizes

The P6 family processors support large page sizes. For 32-bit paging, this facility is enabled with the PSE (page size extension) flag in control register CR4, bit 4. When this flag is set, the processor supports either 4-KByte or 4-MByte page sizes. PAE paging and 4-level paging support 2-MByte pages regardless of the value of CR4.PSE (see Section 4.4, “PAE Paging,” and Section 4.5, “4-Level Paging and 5-Level Paging”). See Chapter 4, “Paging,” for more information about large page sizes.

23.22.2 CD and NW Cache Control Flags

The CD and NW flags in control register CR0 were introduced in the Intel486 processor. In the P6 family and Pentium processors, these flags are used to implement a writeback strategy for the data cache; in the Intel486 processor, they implement a write-through strategy. See Table 12-5 for a comparison of these bits on the P6 family, Pentium, and Intel486 processors. For complete information on caching, see Chapter 12, “Memory Cache Control.”

23.22.3 Descriptor Types and Contents

Operating-system code that manages space in descriptor tables often contains an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access rights values of 80H and 00H remain invalid for the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors. Other values that were invalid on the Intel 286 processor may be valid on the 32-bit processors because uses for these bits have been defined.

---

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.
23.22.4 Changes in Segment Descriptor Loads

On the Intel386 processor, loading a segment descriptor always causes a locked read and write to set the accessed bit of the descriptor. On the P6 family, Pentium, and Intel486 processors, the locked read and write occur only if the bit is not already set.

23.23 DEBUG FACILITIES

The P6 family and Pentium processors include extensions to the Intel486 processor debugging support for breakpoints. To use the new breakpoint features, it is necessary to set the DE flag in control register CR4.

23.23.1 Differences in Debug Register DR6

It is not possible to write a 1 to reserved bit 12 in debug status register DR6 on the P6 family and Pentium processors; however, it is possible to write a 1 in this bit on the Intel486 processor. See Table 10-1 for the different setting of this register following a power-up or hardware reset.

23.23.2 Differences in Debug Register DR7

The P6 family and Pentium processors determines the type of breakpoint access by the R/W0 through R/W3 fields in debug control register DR7 as follows:

00 Break on instruction execution only.
01 Break on data writes only.
10 Undefined if the DE flag in control register CR4 is cleared; break on I/O reads or writes but not instruction fetches if the DE flag in control register CR4 is set.
11 Break on data reads or writes but not instruction fetches.

On the P6 family and Pentium processors, reserved bits 11, 12, 14, and 15 are hard-wired to 0. On the Intel486 processor, however, bit 12 can be set. See Table 10-1 for the different settings of this register following a power-up or hardware reset.

23.23.3 Debug Registers DR4 and DR5

Although the DR4 and DR5 registers are documented as reserved, previous generations of processors aliased references to these registers to debug registers DR6 and DR7, respectively. When debug extensions are not enabled (the DE flag in control register CR4 is cleared), the P6 family and Pentium processors remain compatible with existing software by allowing these aliased references. When debug extensions are enabled (the DE flag is set), attempts to reference registers DR4 or DR5 will result in an invalid-opcode exception (#UD).

23.24 RECOGNITION OF BREAKPOINTS

For the Pentium processor, it is recommended that debuggers execute the LGDT instruction before returning to the program being debugged to ensure that breakpoints are detected. This operation does not need to be performed on the P6 family, Intel486, or Intel386 processors.

The implementation of test registers on the Intel486 processor used for testing the cache and TLB has been redesigned using MSRs on the P6 family and Pentium processors. (Note that MSRs used for this function are different on the P6 family and Pentium processors.) The MOV to and from test register instructions generate invalid-opcode exceptions (#UD) on the P6 family processors.
23.25 EXCEPTIONS AND/OR EXCEPTION CONDITIONS

This section describes the new exceptions and exception conditions added to the 32-bit IA-32 processors and implementation differences in existing exception handling. See Chapter 6, “Interrupt and Exception Handling,” for a detailed description of the IA-32 exceptions.

The Pentium III processor introduced new state with the XMM registers. Computations involving data in these registers can produce exceptions. A new MXCSR control/status register is used to determine which exception or exceptions have occurred. When an exception associated with the XMM registers occurs, an interrupt is generated.

- **SIMD floating-point exception (#XM, interrupt 19)** — New exceptions associated with the SIMD floating-point registers and resulting computations.

No new exceptions were added with the Pentium Pro and Pentium II processors. The set of available exceptions is the same as for the Pentium processor. However, the following exception condition was added to the IA-32 with the Pentium Pro processor:

- **Machine-check exception (#MC, interrupt 18)** — New exception conditions. Many exception conditions have been added to the machine-check exception and a new architecture has been added for handling and reporting on hardware errors. See Chapter 16, “Machine-Check Architecture,” for a detailed description of the new conditions.

The following exceptions and/or exception conditions were added to the IA-32 with the Pentium processor:

- **Machine-check exception (#MC, interrupt 18)** — New exception. This exception reports parity and other hardware errors. It is a model-specific exception and may not be implemented or implemented differently in future processors. The MCE flag in control register CR4 enables the machine-check exception. When this bit is clear (which it is at reset), the processor inhibits generation of the machine-check exception.

- **General-protection exception (#GP, interrupt 13)** — New exception condition added. An attempt to write a 1 to a reserved bit position of a special register causes a general-protection exception to be generated.

- **Page-fault exception (#PF, interrupt 14)** — New exception condition added. When a 1 is detected in any of the reserved bit positions of a page-table entry, page-directory entry, or page-directory pointer during address translation, a page-fault exception is generated.

The following exception was added to the Intel486 processor:

- **Alignment-check exception (#AC, interrupt 17)** — New exception. Reports unaligned memory references when alignment checking is being performed.

The following exceptions and/or exception conditions were added to the Intel386 processor:

- **Divide-error exception (#DE, interrupt 0)**
  - Change in exception handling. Divide-error exceptions on the Intel386 processors always leave the saved CS:IP value pointing to the instruction that failed. On the 8086 processor, the CS:IP value points to the next instruction.
  - Change in exception handling. The Intel386 processors can generate the largest negative number as a quotient for the IDIV instruction (80H and 8000H). The 8086 processor generates a divide-error exception instead.

- **Invalid-opcode exception (#UD, interrupt 6)** — New exception condition added. Improper use of the LOCK instruction prefix can generate an invalid-opcode exception.

- **Page-fault exception (#PF, interrupt 14)** — New exception condition added. If paging is enabled in a 16-bit program, a page-fault exception can be generated as follows. Paging can be used in a system with 16-bit tasks if all tasks use the same page directory. Because there is no place in a 16-bit TSS to store the PDBR register, switching to a 16-bit task does not change the value of the PDBR register. Tasks ported from the Intel 286 processor should be given 32-bit TSSs so they can make full use of paging.

- **General-protection exception (#GP, interrupt 13)** — New exception condition added. The Intel386 processor sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. A general-protection exception is generated if the limit on instruction length is violated. The 8086 processor has no instruction length limit.
23.25.1 Machine-Check Architecture

The Pentium Pro processor introduced a new architecture to the IA-32 for handling and reporting on machine-check exceptions. This machine-check architecture (described in detail in Chapter 16, “Machine-Check Architecture”) greatly expands the ability of the processor to report on internal hardware errors.

23.25.2 Priority of Exceptions

The priority of exceptions are broken down into several major categories:
1. Traps on the previous instruction
2. External interrupts
3. Faults on fetching the next instruction
4. Faults in decoding the next instruction
5. Faults on executing an instruction

There are no changes in the priority of these major categories between the different processors, however, exceptions within these categories are implementation dependent and may change from processor to processor.

23.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers

MMX instructions and a subset of SSE, SSE2, SSSE3 instructions operate on MMX registers. The exception conditions of these instructions are described in the following tables.

Table 23-4. Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H)</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘0’</td>
</tr>
<tr>
<td>MF</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If there is a pending X87 FPU exception</td>
</tr>
<tr>
<td>NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Legacy SSE: Memory operand is not 16-byte aligned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault</td>
</tr>
<tr>
<td>#XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1</td>
</tr>
</tbody>
</table>

Applicable Instructions: CVTPD2PI, CVTTPD2PI
### Table 23-5. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception

<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.EM[bit 2] = 1, CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H)</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is ‘O’</td>
</tr>
<tr>
<td>#MF</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If there is a pending X87 FPU exception</td>
</tr>
<tr>
<td>#NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>If any part of the operand lies outside the effective address space from 0 to FFFFH</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
<tr>
<td>SIMD Floating-point Exception, #XM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1</td>
</tr>
</tbody>
</table>

**Applicable Instructions:** CVTPI2PS, CVTPS2PI, CVTTPS2PI
<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If preceded by a LOCK prefix (F0H)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X X X If any corresponding CPUID feature flag is '0'</td>
</tr>
<tr>
<td>#MF¹</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If there is a pending X87 FPU exception</td>
</tr>
<tr>
<td>#NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal address in the SS segment</td>
</tr>
<tr>
<td>General Protection, #GP(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X X If the memory address is in a non-canonical form.</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>For a page fault</td>
</tr>
<tr>
<td>Alignment Check #AC(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. Applies to "CVTPI2PD xmm, mm" but not "CVTPI2PD xmm, m64".
ARCHITECTURE COMPATIBILITY

64-bit

Protected and
Compatibility

Virtual-8086

Exception

Real

Table 23-7. Exception Conditions for SIMD/MMX Instructions with Memory Reference

Cause of Exception

X

X

X

X

If CR0.EM[bit 2] = 1.

X

X

X

X

If preceded by a LOCK prefix (F0H)

X

X

X

X

If any corresponding CPUID feature flag is ‘0’

#MF

X

X

X

X

If there is a pending X87 FPU exception

#NM

X

X

X

X

If CR0.TS[bit 3]=1

Invalid Opcode, #UD

X

Stack, SS(0)

For an illegal address in the SS segment
X

For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.

X

General Protection,
#GP(0)

X
X

If a memory address referencing the SS segment is in a non-canonical form

X

If the memory address is in a non-canonical form.
If any part of the operand lies outside the effective address space from 0 to FFFFH

#PF(fault-code)

X

X

X

For a page fault

Alignment Check
#AC(0)

X

X

X

If alignment checking is enabled and an unaligned memory reference is made while
the current privilege level is 3.

Applicable Instructions

PABSB, PABSD, PABSW, PACKSSWB, PACKSSDW, PACKUSWB, PADDB, PADDD, PADDQ, PADDW, PADDSB,
PADDSW, PADDUSB, PADDUSW, PALIGNR, PAND, PANDN, PAVGB, PAVGW, PCMPEQB, PCMPEQD, PCMPEQW,
PCMPGTB, PCMPGTD, PCMPGTW, PHADDD, PHADDW, PHADDSW, PHSUBD, PHSUBW, PHSUBSW, PINSRW,
PMADDUBSW, PMADDWD, PMAXSW, PMAXUB, PMINSW, PMINUB, PMULHRSW, PMULHUW, PMULHW, PMULLW,
PMULUDQ, PSADBW, PSHUFB, PSHUFW, PSIGNB PSIGND PSIGNW, PSLLW, PSLLD, PSLLQ, PSRAD, PSRAW,
PSRLW, PSRLD, PSRLQ, PSUBB, PSUBD, PSUBQ, PSUBW, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW,
PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ, PXOR

Vol. 3B 23-25


<table>
<thead>
<tr>
<th>Exception</th>
<th>Real</th>
<th>Virtual-8086</th>
<th>Protected and Compatibility</th>
<th>64-bit</th>
<th>Cause of Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Opcode, #UD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.EM[bit 2] = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If ModR/M.mod ≠ 11b¹</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If preceded by a LOCK prefix (F0H)</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If any corresponding CPUID feature flag is '0'</td>
</tr>
<tr>
<td>#MF</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If there is a pending X87 FPU exception</td>
</tr>
<tr>
<td>#NM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>If CR0.TS[bit 3]=1</td>
</tr>
<tr>
<td>Stack, SS(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>For an illegal address in the SS segment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>If a memory address referencing the SS segment is in a non-canonical form</td>
</tr>
<tr>
<td>#GP(0)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the destination operand is in a non-writable segment.²</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the DS, ES, FS, or GS register contains a NULL segment selector.³</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>For a page fault</td>
</tr>
<tr>
<td>#AC(0)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.</td>
</tr>
</tbody>
</table>

**Applicable Instructions**
- MASKMOVQ, MOVNTQ, "MOVQ (mmreg)"

**NOTES:**
1. Applies to MASKMOVQ only.
2. Applies to MASKMOVQ and MOVQ (mmreg) only.
3. Applies to MASKMOVQ only.
The following differences in handling interrupts are found among the IA-32 processors.

### 23.26.1 Interrupt Propagation Delay
External hardware interrupts may be recognized on different instruction boundaries on the P6 family, Pentium, Intel486, and Intel386 processors, due to the superscaler designs of the P6 family and Pentium processors. Therefore, the EIP pushed onto the stack when servicing an interrupt may be different for the P6 family, Pentium, Intel486, and Intel386 processors.

### 23.26.2 NMI Interrupts
After an NMI interrupt is recognized by the P6 family, Pentium, Intel486, Intel386, and Intel 286 processors, the NMI interrupt is masked until the first IRET instruction is executed, unlike the 8086 processor.

### 23.26.3 IDT Limit
The LIDT instruction can be used to set a limit on the size of the IDT. A double-fault exception (#DF) is generated if an interrupt or exception attempts to read a vector beyond the limit. Shutdown then occurs on the 32-bit IA-32 processors if the double-fault handler vector is beyond the limit. (The 8086 processor does not have a shutdown mode nor a limit.)

### 23.27 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)
The Advanced Programmable Interrupt Controller (APIC), referred to in this book as the local APIC, was introduced into the IA-32 processors with the Pentium processor (beginning with the 735/90 and 815/100 models) and is included in the Pentium 4, Intel Xeon, and P6 family processors. The features and functions of the local APIC are derived from the Intel 82489DX external APIC, which was used with the Intel486 and early Pentium processors. Additional refinements of the local APIC architecture were incorporated in the Pentium 4 and Intel Xeon processors.
23.27.1 Software Visible Differences Between the Local APIC and the 82489DX

The following features in the local APIC features differ from those found in the 82489DX external APIC:

- When the local APIC is disabled by clearing the APIC software enable/disable flag in the spurious-interrupt vector MSR, the state of its internal registers are unaffected, except that the mask bits in the LVT are all set to block local interrupts to the processor. Also, the local APIC ceases accepting IPIs except for INIT, SMI, NMI, and start-up IPIs. In the 82489DX, when the local unit is disabled, all the internal registers including the IRR, ISR, and TMR are cleared and the mask bits in the LVT are set. In this state, the 82489DX local unit will accept only the reset deassert message.

- In the local APIC, NMI and INIT (except for INIT deassert) are always treated as edge triggered interrupts, even if programmed otherwise. In the 82489DX, these interrupts are always level triggered.

- In the local APIC, IPIs generated through the ICR are always treated as edge triggered (except INIT Deassert). In the 82489DX, the ICR can be used to generate either edge or level triggered IPIs.

- In the local APIC, the logical destination register supports 8 bits; in the 82489DX, it supports 32 bits.

- In the local APIC, the APIC ID register is 4 bits wide; in the 82489DX, it is 8 bits wide.

- The remote read delivery mode provided in the 82489DX and local APIC for Pentium processors is not supported in the local APIC in the Pentium 4, Intel Xeon, and P6 family processors.

- For the 82489DX, in the lowest priority delivery mode, all the target local APICs specified by the destination field participate in the lowest priority arbitration. For the local APIC, only those local APICs which have free interrupt slots will participate in the lowest priority arbitration.

23.27.2 New Features Incorporated in the Local APIC for the P6 Family and Pentium Processors

The local APIC in the Pentium and P6 family processors have the following new features not found in the 82489DX external APIC.

- Cluster addressing is supported in logical destination mode.

- Focus processor checking can be enabled/disabled.

- Interrupt input signal polarity can be programmed for the LINT0 and LINT1 pins.

- An SMI IPI is supported through the ICR and I/O redirection table.

- An error status register is incorporated into the LVT to log and report APIC errors.

In the P6 family processors, the local APIC incorporates an additional LVT register to handle performance monitoring counter interrupts.

23.27.3 New Features Incorporated in the Local APIC of the Pentium 4 and Intel Xeon Processors

The local APIC in the Pentium 4 and Intel Xeon processors has the following new features not found in the P6 family and Pentium processors and in the 82489DX.

- The local APIC ID is extended to 8 bits.

- An thermal sensor register is incorporated into the LVT to handle thermal sensor interrupts.

- The the ability to deliver lowest-priority interrupts to a focus processor is no longer supported.

- The flat cluster logical destination mode is not supported.

23.28 TASK SWITCHING AND TSS

This section identifies the implementation differences of task switching, additions to the TSS and the handling of TSSs and TSS segment selectors.
23.28.1 P6 Family and Pentium Processor TSS
When the virtual mode extensions are enabled (by setting the VME flag in control register CR4), the TSS in the P6 family and Pentium processors contain an interrupt redirection bit map, which is used in virtual-8086 mode to redirect interrupts back to an 8086 program.

23.28.2 TSS Selector Writes
During task state saves, the Intel486 processor writes 2-byte segment selectors into a 32-bit TSS, leaving the upper 16 bits undefined. For performance reasons, the P6 family and Pentium processors write 4-byte segment selectors into the TSS, with the upper 2 bytes being 0. For compatibility reasons, code should not depend on the value of the upper 16 bits of the selector in the TSS.

23.28.3 Order of Reads/Writes to the TSS
The order of reads and writes into the TSS is processor dependent. The P6 family and Pentium processors may generate different page-fault addresses in control register CR2 in the same TSS area than the Intel486 and Intel386 processors, if a TSS crosses a page boundary (which is not recommended).

23.28.4 Using A 16-Bit TSS with 32-Bit Constructs
Task switches using 16-bit TSSs should be used only for pure 16-bit code. Any new code written using 32-bit constructs (operands, addressing, or the upper word of the EFLAGS register) should use only 32-bit TSSs. This is due to the fact that the 32-bit processors do not save the upper 16 bits of EFLAGS to a 16-bit TSS. A task switch back to a 16-bit task that was executing in virtual mode will never re-enable the virtual mode, as this flag was not saved in the upper half of the EFLAGS value in the TSS. Therefore, it is strongly recommended that any code using 32-bit constructs use a 32-bit TSS to ensure correct behavior in a multitasking environment.

23.28.5 Differences in I/O Map Base Addresses
The Intel486 processor considers the TSS segment to be a 16-bit segment and wraps around the 64K boundary. Any I/O accesses check for permission to access this I/O address at the I/O base address plus the I/O offset. If the I/O map base address exceeds the specified limit of 0DFFFFH, an I/O access will wrap around and obtain the permission for the I/O address at an incorrect location within the TSS. A TSS limit violation does not occur in this situation on the Intel486 processor. However, the P6 family and Pentium processors consider the TSS to be a 32-bit segment and a limit violation occurs when the I/O base address plus the I/O offset is greater than the TSS limit. By following the recommended specification for the I/O base address to be less than 0DFFFFH, the Intel486 processor will not wrap around and access incorrect locations within the TSS for I/O port validation and the P6 family and Pentium processors will not experience general-protection exceptions (#GP). Figure 23-1 demonstrates the different areas accessed by the Intel486 and the P6 family and Pentium processors.
23.29 CACHE MANAGEMENT

The P6 family processors include two levels of internal caches: L1 (level 1) and L2 (level 2). The L1 cache is divided into an instruction cache and a data cache; the L2 cache is a general-purpose cache. See Section 12.1, “Internal Caches, TLBs, and Buffers,” for a description of these caches. (Note that although the Pentium II processor L2 cache is physically located on a separate chip in the cassette, it is considered an internal cache.)

The Pentium processor includes separate level 1 instruction and data caches. The data cache supports a writeback (or alternatively write-through, on a line by line basis) policy for memory updates.

The Intel486 processor includes a single level 1 cache for both instructions and data.

The meaning of the CD and NW flags in control register CR0 have been redefined for the P6 family and Pentium processors. For these processors, the recommended value (00B) enables writeback for the data cache of the Pentium processor and for the L1 data cache and L2 cache of the P6 family processors. In the Intel486 processor, setting these flags to (00B) enables write-through for the cache.

External system hardware can force the Pentium processor to disable caching or to use the write-through cache policy should that be required. In the P6 family processors, the MTRRs can be used to override the CD and NW flags (see Table 12-6).

The P6 family and Pentium processors support page-level cache management in the same manner as the Intel486 processor by using the PCD and PWT flags in control register CR3, the page-directory entries, and the page-table entries. The Intel486 processor, however, is not affected by the state of the PWT flag since the internal cache of the Intel486 processor is a write-through cache.

23.29.1 Self-Modifying Code with Cache Enabled

On the Intel486 processor, a write to an instruction in the cache will modify it in both the cache and memory. If the instruction was prefetched before the write, however, the old version of the instruction could be the one executed. To prevent this problem, it is necessary to flush the instruction prefetch unit of the Intel486 processor by coding a jump instruction immediately after any write that modifies an instruction. The P6 family and Pentium processors, however, check whether a write may modify an instruction that has been prefetched for execution. This check is based on the linear address of the instruction. If the linear address of an instruction is found to be present in the
prefetch queue, the P6 family and Pentium processors flush the prefetch queue, eliminating the need to code a jump instruction after any writes that modify an instruction.

Because the linear address of the write is checked against the linear address of the instructions that have been prefetched, special care must be taken for self-modifying code to work correctly when the physical addresses of the instruction and the written data are the same, but the linear addresses differ. In such cases, it is necessary to execute a serializing operation to flush the prefetch queue after the write and before executing the modified instruction. See Section 9.3, “Serializing Instructions,” for more information on serializing instructions.

NOTE
The check on linear addresses described above is not in practice a concern for compatibility. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. System software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction must execute a serializing operation, such as IRET, before the modified instruction is executed.

23.29.2 Disabling the L3 Cache
A unified third-level (L3) cache in processors based on Intel NetBurst microarchitecture (see Section 12.1, "Internal Caches, TLBs, and Buffers") provides the third-level cache disable flag, bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches (see Section 12.5.4, "Disabling and Enabling the L3 Cache"). The third-level cache disable flag applies only to processors based on Intel NetBurst microarchitecture. Processors with L3 and based on other microarchitectures do not support the third-level cache disable flag.

23.30 PAGING
This section identifies enhancements made to the paging mechanism and implementation differences in the paging mechanism for various IA-32 processors.

23.30.1 Large Pages
The Pentium processor extended the memory management/paging facilities of the IA-32 to allow large (4 MBytes) pages sizes (see Section 4.3, "32-Bit Paging"). The first P6 family processor (the Pentium Pro processor) added a 2 MByte page size to the IA-32 in conjunction with the physical address extension (PAE) feature (see Section 4.4, "PAE Paging").

The availability of large pages with 32-bit paging on any IA-32 processor can be determined via feature bit 3 (PSE) of register EDX after the CPUID instruction has been execution with an argument of 1. (Large pages are always available with PAE paging and 4-level paging.) Intel processors that do not support the CPUID instruction support only 32-bit paging and do not support page size enhancements. (See “CPUID—CPU Identification” in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, for more information on the CPUID instruction.)

23.30.2 PCD and PWT Flags
The PCD and PWT flags were introduced to the IA-32 in the Intel486 processor to control the caching of pages:

- PCD (page-level cache disable) flag—Controls caching on a page-by-page basis.
- PWT (page-level write-through) flag—Controls the write-through/writeback caching policy on a page-by-page basis. Since the internal cache of the Intel486 processor is a write-through cache, it is not affected by the state of the PWT flag.
23.30.3 Enabling and Disabling Paging

Paging is enabled and disabled by loading a value into control register CR0 that modifies the PG flag. For backward and forward compatibility with all IA-32 processors, Intel recommends that the following operations be performed when enabling or disabling paging:

1. Execute a MOV CR0, REG instruction to either set (enable paging) or clear (disable paging) the PG flag.
2. Execute a near JMP instruction.

The sequence bounded by the MOV and JMP instructions should be identity mapped (that is, the instructions should reside on a page whose linear and physical addresses are identical).

For the P6 family processors, the MOV CR0, REG instruction is serializing, so the jump operation is not required. However, for backwards compatibility, the JMP instruction should still be included.

23.31 Stack Operations and Supervisor Software

This section identifies the differences in the stack mechanism for the various IA-32 processors.

23.31.1 Selector Pushes and Pops

When pushing a segment selector onto the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors decrement the ESP register by the operand size and then write 2 bytes. If the operand size is 32-bits, the upper two bytes of the write are not modified. The Pentium processor decrements the ESP register by the operand size and determines the size of the write by the operand size. If the operand size is 32-bits, the upper two bytes are written as 0s.

When popping a segment selector from the stack, the Pentium 4, Intel Xeon, P6 family, and Intel486 processors read 2 bytes and increment the ESP register by the operand size of the instruction. The Pentium processor determines the size of the read from the operand size and increments the ESP register by the operand size.

It is possible to align a 32-bit selector push or pop such that the operation generates an exception on a Pentium processor and not on a Pentium 4, Intel Xeon, P6 family, or Intel486 processor. This could occur if the third and/or fourth byte of the operation lies beyond the limit of the segment or if the third and/or fourth byte of the operation is located on a non-present or inaccessible page.

For a POP-to-memory instruction that meets the following conditions:
• The stack segment size is 16-bit.
• Any 32-bit addressing form with the SIB byte specifying ESP as the base register.
• The initial stack pointer is FFFCH (32-bit operand) or FFFEH (16-bit operand) and will wrap around to 0H as a result of the POP operation.

The result of the memory write is implementation-specific. For example, in P6 family processors, the result of the memory write is SS:0H plus any scaled index and displacement. In Pentium processors, the result of the memory write may be either a stack fault (real mode or protected mode with stack segment size of 64 KByte), or write to SS:10000H plus any scaled index and displacement (protected mode and stack segment size exceeds 64 KByte).

23.31.2 Error Code Pushes

The Intel486 processor implements the error code pushed on the stack as a 16-bit value. When pushed onto a 32-bit stack, the Intel486 processor only pushes 2 bytes and updates ESP by 4. The P6 family and Pentium processors’ error code is a full 32 bits with the upper 16 bits set to zero. The P6 family and Pentium processors, therefore, push 4 bytes and update ESP by 4. Any code that relies on the state of the upper 16 bits may produce inconsistent results.
23.31.3 Fault Handling Effects on the Stack

During the handling of certain instructions, such as CALL and PUSHA, faults may occur in different sequences for the different processors. For example, during far calls, the Intel486 processor pushes the old CS and EIP before a possible branch fault is resolved. A branch fault is a fault from a branch instruction occurring from a segment limit or access rights violation. If a branch fault is taken, the Intel486 and P6 family processors will have corrupted memory below the stack pointer. However, the ESP register is backed up to make the instruction restartable. The P6 family processors issue the branch before the pushes. Therefore, if a branch fault does occur, these processors do not corrupt memory below the stack pointer. This implementation difference, however, does not constitute a compatibility problem, as only values at or above the stack pointer are considered to be valid. Other operations that encounter faults may also corrupt memory below the stack pointer and this behavior may vary on different implementations.

23.31.4 Interlevel RET/IRET From a 16-Bit Interrupt or Call Gate

If a call or interrupt is made from a 32-bit stack environment through a 16-bit gate, only 16 bits of the old ESP can be pushed onto the stack. On the subsequent RET/IRET, the 16-bit ESP is popped but the full 32-bit ESP is updated since control is being resumed in a 32-bit stack environment. The Intel486 processor writes the SS selector into the upper 16 bits of ESP. The P6 family and Pentium processors write zeros into the upper 16 bits.

23.32 MIXING 16- AND 32-BIT SEGMENTS

The features of the 16-bit Intel 286 processor are an object-code compatible subset of those of the 32-bit IA-32 processors. The D (default operation size) flag in segment descriptors indicates whether the processor treats a code or data segment as a 16-bit or 32-bit segment; the B (default stack size) flag in segment descriptors indicates whether the processor treats a stack segment as a 16-bit or 32-bit segment.

The segment descriptors used by the Intel 286 processor are supported by the 32-bit IA-32 processors if the Intel-reserved word (highest word) of the descriptor is clear. On the 32-bit IA-32 processors, this word includes the upper bits of the base address and the segment limit.

The segment descriptors for data segments, code segments, local descriptor tables (there are no descriptors for global descriptor tables), and task gates are the same for the 16- and 32-bit processors. Other 16-bit descriptors (TSS segment, call gate, interrupt gate, and trap gate) are supported by the 32-bit processors.

The 32-bit processors also have descriptors for TSS segments, call gates, interrupt gates, and trap gates that support the 32-bit architecture. Both kinds of descriptors can be used in the same system.

For those segment descriptors common to both 16- and 32-bit processors, clear bits in the reserved word cause the 32-bit processors to interpret these descriptors exactly as an Intel 286 processor does, that is:

- Base Address — The upper 8 bits of the 32-bit base address are clear, which limits base addresses to 24 bits.
- Limit — The upper 4 bits of the limit field are clear, restricting the value of the limit field to 64 KBytes.
- Granularity bit — The G (granularity) flag is clear, indicating the value of the 16-bit limit is interpreted in units of 1 byte.
- Big bit — In a data-segment descriptor, the B flag is clear in the segment descriptor used by the 32-bit processors, indicating the segment is no larger than 64 KBytes.
- Default bit — In a code-segment descriptor, the D flag is clear, indicating 16-bit addressing and operands are the default. In a stack-segment descriptor, the D flag is clear, indicating use of the SP register (instead of the ESP register) and a 64-KByte maximum segment limit.

For information on mixing 16- and 32-bit code in applications, see Chapter 22, “Mixing 16-Bit and 32-Bit Code.”

23.33 SEGMENT AND ADDRESS WRAPAROUND

This section discusses differences in segment and address wraparound between the P6 family, Pentium, Intel486, Intel386, Intel 286, and 8086 processors.
23.33.1 Segment Wraparound

On the 8086 processor, an attempt to access a memory operand that crosses offset 65,535 or 0FFFFH or offset 0 (for example, moving a word to offset 65,535 or pushing a word when the stack pointer is set to 1) causes the offset to wrap around modulo 65,536 or 010000H. With the Intel 286 processor, any base and offset combination that addresses beyond 16 MBytes wraps around to the 1 MByte of the address space. The P6 family, Pentium, Intel486, and Intel386 processors in real-address mode generate an exception in these cases:

• A general-protection exception (#GP) if the segment is a data segment (that is, if the CS, DS, ES, FS, or GS register is being used to address the segment).
• A stack-fault exception (#SS) if the segment is a stack segment (that is, if the SS register is being used).

An exception to this behavior occurs when a stack access is data aligned, and the stack pointer is pointing to the last aligned piece of data that size at the top of the stack (ESP is FFFFFFFCH). When this data is popped, no segment limit violation occurs and the stack pointer will wrap around to 0.

The address space of the P6 family, Pentium, and Intel486 processors may wrap around at 1 MByte in real-address mode. An external A20M# pin forces wraparound if enabled. On Intel 8086 processors, it is possible to specify addresses greater than 1 MByte. For example, with a selector value FFFFH and an offset of FFFFH, the effective address would be 10FFEH (1 MByte plus 65519 bytes). The 8086 processor, which can form addresses up to 20 bits long, truncates the uppermost bit, which "wraps" this address to FFEFH. However, the P6 family, Pentium, and Intel486 processors do not truncate this bit if A20M# is not enabled.

If a stack operation wraps around the address limit, shutdown occurs. (The 8086 processor does not have a shutdown mode or a limit.)

The behavior when executing near the limit of a 4-GByte selector (limit = FFFFFFFFH) is different between the Pentium Pro and the Pentium 4 family of processors. On the Pentium Pro, instructions which cross the limit -- for example, a two byte instruction such as INC EAX that is encoded as FFH C0H starting exactly at the limit faults for a segment violation (a one byte instruction at FFFFFFFFH does not cause an exception). Using the Pentium 4 microprocessor family, neither of these situations causes a fault.

Segment wraparound and the functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

23.34 STORE BUFFERS AND MEMORY ORDERING

The Pentium 4, Intel Xeon, and P6 family processors provide a store buffer for temporary storage of writes (stores) to memory (see Section 12.10, "Store Buffer"). Writes stored in the store buffer(s) are always written to memory in program order, with the exception of "fast string" store operations (see Section 9.2.4, "Fast-String Operation and Out-of-Order Stores").

The Pentium processor has two store buffers, one corresponding to each of the pipelines. Writes in these buffers are always written to memory in the order they were generated by the processor core.

It should be noted that only memory writes are buffered and I/O writes are not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors do not synchronize the completion of memory writes on the bus and instruction execution after a write. An I/O, locked, or serializing instruction needs to be executed to synchronize writes with the next instruction (see Section 9.3, "Serializing Instructions").

The Pentium 4, Intel Xeon, and P6 family processors use processor ordering to maintain consistency in the order that data is read (loaded) and written (stored) in a program and the order the processor actually carries out the reads and writes. With this type of ordering, reads can be carried out speculatively and in any order, reads can pass buffered writes, and writes to memory are always carried out in program order. (See Section 9.2, "Memory Ordering," for more information about processor ordering.) The Pentium III processor introduced a new instruction to serialize writes and make them globally visible. Memory ordering issues can arise between a producer and a consumer of data. The SFENCE instruction provides a performance-efficient way of ensuring ordering between routines that produce weakly-ordered results and routines that consume this data.

No re-ordering of reads occurs on the Pentium processor, except under the condition noted in Section 9.2.1, "Memory Ordering in the Intel® Pentium® and Intel486™ Processors,” and in the following paragraph describing the Intel486 processor.
Specifically, the store buffers are flushed before the IN instruction is executed. No reads (as a result of cache miss) are reordered around previously generated writes sitting in the store buffers. The implication of this is that the store buffers will be flushed or emptied before a subsequent bus cycle is run on the external bus.

On both the Intel486 and Pentium processors, under certain conditions, a memory read will go onto the external bus before the pending memory writes in the buffer even though the writes occurred earlier in the program execution. A memory read will only be reordered in front of all writes pending in the buffers if all writes pending in the buffers are cache hits and the read is a cache miss. Under these conditions, the Intel486 and Pentium processors will not read from an external memory location that needs to be updated by one of the pending writes.

During a locked bus cycle, the Intel486 processor will always access external memory, it will never look for the location in the on-chip cache. All data pending in the Intel486 processor’s store buffers will be written to memory before a locked cycle is allowed to proceed to the external bus. Thus, the locked bus cycle can be used for eliminating the possibility of reordering read cycles on the Intel486 processor. The Pentium processor does check its cache on a read-modify-write access and, if the cache line has been modified, writes the contents back to memory before locking the bus. The P6 family processors write to their cache on a read-modify-write operation (if the access does not split across a cache line) and does not write back to system memory. If the access does split across a cache line, it locks the bus and accesses system memory.

I/O reads are never reordered in front of buffered memory writes on an IA-32 processor. This ensures an update of all memory locations before reading the status from an I/O device.

### 23.35 BUS LOCKING

The Intel 286 processor performs the bus locking differently than the Intel P6 family, Pentium, Intel486, and Intel386 processors. Programs that use forms of memory locking specific to the Intel 286 processor may not run properly when run on later processors.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and Intel 286 configurations lock the entire physical memory space. Programmers should not depend on this.

On the Intel 286 processor, the LOCK prefix is sensitive to IOPL. If the CPL is greater than the IOPL, a general-protection exception (#GP) is generated. On the Intel386 DX, Intel486, and Pentium, and P6 family processors, no check against IOPL is performed.

The Pentium processor automatically asserts the LOCK# signal when acknowledging external interrupts. After signaling an interrupt request, an external interrupt controller may use the data bus to send the interrupt vector to the processor. After receiving the interrupt request signal, the processor asserts LOCK# to ensure that no other data appears on the data bus until the interrupt vector is received. This bus locking does not occur on the P6 family processors.

### 23.36 BUS HOLD

Unlike the 8086 and Intel 286 processors, but like the Intel386 and Intel486 processors, the P6 family and Pentium processors respond to requests for control of the bus from other potential bus masters, such as DMA controllers, between transfers of parts of an unaligned operand, such as two words which form a doubleword. Unlike the Intel386 processor, the P6 family, Pentium, and Intel486 processors respond to bus hold during reset initialization.

### 23.37 MODEL-SPECIFIC EXTENSIONS TO THE IA-32

Certain extensions to the IA-32 are specific to a processor or family of IA-32 processors and may not be implemented or implemented in the same way in future processors. The following sections describe these model-specific extensions. The CPUID instruction indicates the availability of some of the model-specific features.
23.37.1 Model-Specific Registers

The Pentium processor introduced a set of model-specific registers (MSRs) for use in controlling hardware functions and performance monitoring. To access these MSRs, two new instructions were added to the IA-32 architecture: read MSR (RDMSR) and write MSR (WRMSR). The MSRs in the Pentium processor are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

The P6 family processors greatly increased the number of MSRs available to software. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, for a complete list of the available MSRs. The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine-check architecture, and the MTRRs. These registers are accessible using the RDMSR and WRMSR instructions. Specific information on some of these new MSRs is provided in the following sections. As with the Pentium processor MSR, the P6 family processor MSRs are not guaranteed to be duplicated or provided in the next generation IA-32 processors.

23.37.2 RDMSR and WRMSR Instructions

The RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions recognize a much larger number of model-specific registers in the P6 family processors. (See "RDMSR—Read from Model Specific Register" and "WRMSR—Write to Model Specific Register" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D, for more information.)

23.37.3 Memory Type Range Registers

Memory type range registers (MTRRs) are a new feature introduced into the IA-32 in the Pentium Pro processor. MTRRs allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped I/O.

MTRRs are MSRs that contain an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal memory map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), uncacheable, speculatable, write-combining (WC), write-through (WT), write-protected (WP), and writeback (WB).

Earlier IA-32 processors (such as the Intel486 and Pentium processors) used the KEN# (cache enable) pin and external logic to maintain an external memory map and signal cacheable accesses to the processor. The MTRR mechanism simplifies hardware designs by eliminating the KEN# pin and the external logic required to drive it.


23.37.4 Machine-Check Exception and Architecture

The Pentium processor introduced a new exception called the machine-check exception (#MC, interrupt 18). This exception is used to detect hardware-related errors, such as a parity error on a read cycle.

The P6 family processors extend the types of errors that can be detected and that generate a machine-check exception. It also provides a new machine-check architecture for recording information about a machine-check error and provides extended recovery capability.

The machine-check architecture provides several banks of reporting registers for recording machine-check errors. Each bank of registers is associated with a specific hardware unit in the processor. The primary focus of the machine checks is on bus and interconnect operations; however, checks are also made of translation lookaside buffer (TLB) and cache operations.
The machine-check architecture can correct some errors automatically and allow for reliable restart of instruction execution. It also collects sufficient information for software to use in correcting other machine errors not corrected by hardware.

See Chapter 16, “Machine-Check Architecture,” for more information on the machine-check exception and the machine-check architecture.

### 23.37.5 Performance-Monitoring Counters

The P6 family and Pentium processors provide two performance-monitoring counters for use in monitoring internal hardware operations. The number of performance monitoring counters and associated programming interfaces may be implementation specific for Pentium 4 processors, Pentium M processors. Later processors may have implemented these as part of an architectural performance monitoring feature. The architectural and non-architectural performance monitoring interfaces for different processor families are described in Chapter 20, "Performance Monitoring." [https://perfmon-events.intel.com/](https://perfmon-events.intel.com/) lists all the events that can be counted for architectural performance monitoring events and non-architectural events. The counters are set up, started, and stopped using two MSRs and the RDMSR and WRMSR instructions. For the P6 family processors, the current count for a particular counter can be read using the new RDPMC instruction.

The performance-monitoring counters are useful for debugging programs, optimizing code, diagnosing system failures, or refining hardware designs. See Chapter 20, "Performance Monitoring," for more information on these counters.

### 23.38 TWO WAYS TO RUN INTEL 286 PROCESSOR TASKS

When porting 16-bit programs to run on 32-bit IA-32 processors, there are two approaches to consider:

- Porting an entire 16-bit software system to a 32-bit processor, complete with the old operating system, loader, and system builder. Here, all tasks will have 16-bit TSSs. The 32-bit processor is being used as if it were a faster version of the 16-bit processor.
- Porting selected 16-bit applications to run in a 32-bit processor environment with a 32-bit operating system, loader, and system builder. Here, the TSSs used to represent 286 tasks should be changed to 32-bit TSSs. It is possible to mix 16 and 32-bit TSSs, but the benefits are small and the problems are great. All tasks in a 32-bit software system should have 32-bit TSSs. It is not necessary to change the 16-bit object modules themselves; TSSs are usually constructed by the operating system, by the loader, or by the system builder. See Chapter 22, "Mixing 16-Bit and 32-Bit Code," for more detailed information about mixing 16-bit and 32-bit code.

Because the 32-bit processors use the contents of the reserved word of 16-bit segment descriptors, 16-bit programs that place values in this word may not run correctly on the 32-bit processors.

### 23.39 INITIAL STATE OF PENTIUM, PENTIUM PRO AND PENTIUM 4 PROCESSORS

Table 23-10 shows the state of the flags and other registers following power-up for the Pentium, Pentium Pro and Pentium 4 processors. The state of control register CR0 is 60000010H (see Figure 10-1 "Contents of CR0 Register after Reset" in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A). This places the processor in real-address mode with paging disabled.

<table>
<thead>
<tr>
<th>Register</th>
<th>Pentium 4 Processor</th>
<th>Pentium Pro Processor</th>
<th>Pentium Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS¹</td>
<td>00000002H</td>
<td>00000002H</td>
<td>00000002H</td>
</tr>
<tr>
<td>EIP</td>
<td>0000FF00H</td>
<td>0000FF00H</td>
<td>0000FF00H</td>
</tr>
<tr>
<td>CR0</td>
<td>60000010H²</td>
<td>60000010H²</td>
<td>60000010H²</td>
</tr>
<tr>
<td>CR2, CR3, CR4</td>
<td>00000000H</td>
<td>00000000H</td>
<td>00000000H</td>
</tr>
<tr>
<td>Register</td>
<td>Pentium 4 Processor</td>
<td>Pentium Pro Processor</td>
<td>Pentium Processor</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>-----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>CS</td>
<td>Selector = F000H</td>
<td>Selector = F000H</td>
<td>Selector = F000H</td>
</tr>
<tr>
<td></td>
<td>Base = FFFF0000H</td>
<td>Base = FFFF0000H</td>
<td>Base = FFFF0000H</td>
</tr>
<tr>
<td></td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
</tr>
<tr>
<td>SS, DS, ES, FS, GS</td>
<td>Selector = 0000H</td>
<td>Selector = 0000H</td>
<td>Selector = 0000H</td>
</tr>
<tr>
<td></td>
<td>Base = 00000000H</td>
<td>Base = 00000000H</td>
<td>Base = 00000000H</td>
</tr>
<tr>
<td></td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
<td>Limit = FFFFH</td>
</tr>
<tr>
<td>EDX</td>
<td>00000FxxH</td>
<td>000n06xxH^3</td>
<td>000005xxH</td>
</tr>
<tr>
<td>EAX</td>
<td>0^4</td>
<td>0^4</td>
<td>0^4</td>
</tr>
<tr>
<td>EBX, ECX, ESI, EDI, EBP, ESP</td>
<td>00000000H</td>
<td>00000000H</td>
<td>00000000H</td>
</tr>
<tr>
<td>ST0 through ST7^5</td>
<td>Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged</td>
<td>Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged</td>
<td>Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged</td>
</tr>
<tr>
<td>x87 FPU Control Word^5</td>
<td>Pwr up or Reset: 0040H FINIT/FNINIT: 037FH</td>
<td>Pwr up or Reset: 0040H FINIT/FNINIT: 037FH</td>
<td>Pwr up or Reset: 0040H FINIT/FNINIT: 037FH</td>
</tr>
<tr>
<td>x87 FPU Status Word^5</td>
<td>Pwr up or Reset: 0000H FINIT/FNINIT: 0000H</td>
<td>Pwr up or Reset: 0000H FINIT/FNINIT: 0000H</td>
<td>Pwr up or Reset: 0000H FINIT/FNINIT: 0000H</td>
</tr>
<tr>
<td>x87 FPU Tag Word^5</td>
<td>Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH</td>
<td>Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH</td>
<td>Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH</td>
</tr>
<tr>
<td>x87 FPU Data Operand and CS Seg. Selectors^5</td>
<td>Pwr up or Reset: 0000H FINIT/FNINIT: 0000H</td>
<td>Pwr up or Reset: 0000H FINIT/FNINIT: 0000H</td>
<td>Pwr up or Reset: 0000H FINIT/FNINIT: 0000H</td>
</tr>
<tr>
<td>x87 FPU Data Operand and Inst. Pointers^5</td>
<td>Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H</td>
<td>Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H</td>
<td>Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H</td>
</tr>
<tr>
<td>MM0 through MM7^5</td>
<td>Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged</td>
<td>Pentium II and Pentium III Processors Only—Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged</td>
<td>Pentium with MMX Technology Only—Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged</td>
</tr>
<tr>
<td>XMM0 through XMM7</td>
<td>Pwr up or Reset: 0H INIT: Unchanged</td>
<td>If CPUID.01H:SSE is 1 —Pwr up or Reset: 0H INIT: Unchanged</td>
<td>NA</td>
</tr>
<tr>
<td>MXCSR</td>
<td>Pwr up or Reset: 1F80H INIT: Unchanged</td>
<td>Pentium III processor only—Pwr up or Reset: 1F80H INIT: Unchanged</td>
<td>NA</td>
</tr>
<tr>
<td>GDTR, IDTR</td>
<td>Base = 00000000H Limit = FFFFH AR = Present, R/W</td>
<td>Base = 00000000H Limit = FFFFH AR = Present, R/W</td>
<td>Base = 00000000H Limit = FFFFH AR = Present, R/W</td>
</tr>
<tr>
<td>LDTR, Task Register</td>
<td>Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W</td>
<td>Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W</td>
<td>Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W</td>
</tr>
<tr>
<td>DR0, DR1, DR2, DR3</td>
<td>000000000H</td>
<td>000000000H</td>
<td>000000000H</td>
</tr>
<tr>
<td>DR6</td>
<td>FFFF0FF0H</td>
<td>FFFF0FF0H</td>
<td>FFFF0FF0H</td>
</tr>
<tr>
<td>Register</td>
<td>Pentium 4 Processor</td>
<td>Pentium Pro Processor</td>
<td>Pentium Processor</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------</td>
<td>-----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>DR7</td>
<td>00000400H</td>
<td>00000400H</td>
<td>00000400H</td>
</tr>
<tr>
<td>Time-Stamp Counter</td>
<td>Power up or Reset: 0H INIT: Unchanged</td>
<td>Power up or Reset: 0H INIT: Unchanged</td>
<td>Power up or Reset: 0H INIT: Unchanged</td>
</tr>
<tr>
<td>Perf. Counters and Event Select</td>
<td>Power up or Reset: 0H INIT: Unchanged</td>
<td>Power up or Reset: 0H INIT: Unchanged</td>
<td>Power up or Reset: 0H INIT: Unchanged</td>
</tr>
<tr>
<td>All Other MSRs</td>
<td>Pwr up or Reset: Undefined INIT: Unchanged</td>
<td>Pwr up or Reset: Undefined INIT: Unchanged</td>
<td>Pwr up or Reset: Undefined INIT: Unchanged</td>
</tr>
<tr>
<td>Data and Code Cache, TLBs</td>
<td>Invalid⁶</td>
<td>Invalid⁶</td>
<td>Invalid⁶</td>
</tr>
<tr>
<td>Fixed MTRRs</td>
<td>Pwr up or Reset: Disabled INIT: Unchanged</td>
<td>Pwr up or Reset: Disabled INIT: Unchanged</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>Variable MTRRs</td>
<td>Pwr up or Reset: Disabled INIT: Unchanged</td>
<td>Pwr up or Reset: Disabled INIT: Unchanged</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>Machine-Check Architecture</td>
<td>Pwr up or Reset: Undefined INIT: Unchanged</td>
<td>Pwr up or Reset: Undefined INIT: Unchanged</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>APIC</td>
<td>Pwr up or Reset: Enabled INIT: Unchanged</td>
<td>Pwr up or Reset: Enabled INIT: Unchanged</td>
<td>Pwr up or Reset: Enabled INIT: Unchanged</td>
</tr>
<tr>
<td>R8-R15⁷</td>
<td>0000000000000000H</td>
<td>0000000000000000H</td>
<td>N.A.</td>
</tr>
<tr>
<td>XMM8-XMM15⁷</td>
<td>Pwr up or Reset: 0H INIT: Unchanged</td>
<td>Pwr up or Reset: 0H INIT: Unchanged</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where “n” is the Extended Model Value for the respective processor.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
7. If the processor supports IA-32e mode.
ARCHITECTURE COMPATIBILITY
24.1 OVERVIEW

This chapter describes the basics of virtual machine architecture and an overview of the virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environments.

Information about VMX instructions is provided in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B. Other aspects of VMX and system programming considerations are described in chapters of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

24.2 VIRTUAL MACHINE ARCHITECTURE

Virtual-machine extensions define processor-level support for virtual machines on IA-32 processors. Two principal classes of software are supported:

- **Virtual-machine monitors (VMM)** — A VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents guest software (see next paragraph) with an abstraction of a virtual processor and allows it to execute directly on a logical processor. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O.

- **Guest software** — Each virtual machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software. Each operates independently of other virtual machines and uses on the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The software stack acts as if it were running on a platform with no VMM. Software executing in a virtual machine must operate with reduced privilege so that the VMM can retain control of platform resources.

24.3 INTRODUCTION TO VMX OPERATION

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation. In general, a VMM will run in VMX root operation and guest software will run in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits.

Processor behavior in VMX root operation is very much as it is outside VMX operation. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited (see Section 24.8).

Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

There is no software-visible bit whose setting indicates whether a logical processor is in VMX non-root operation. This fact may allow a VMM to prevent guest software from determining that it is running in a virtual machine.

Because VMX operation places restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.
24.4 LIFE CYCLE OF VMM SOFTWARE

Figure 24-1 illustrates the life cycle of a VMM and its guest software as well as the interactions between them. The following items summarize that life cycle:

- Software enters VMX operation by executing a VMXON instruction.
- Using VM entries, a VMM can then enter guests into virtual machines (one at a time). The VMM effects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.
- VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the virtual machine using a VM entry.
- Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction.

![Image of the life cycle of a VMM and its guest software](image-url)

Figure 24-1. Interaction of a Virtual-Machine Monitor and Guests

24.5 VIRTUAL-MACHINE CONTROL STRUCTURE

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual-machine control structure (VMCS).

Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions.

A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor.

24.6 DISCOVERING SUPPORT FOR VMX

Before system software enters into VMX operation, it must discover the presence of VMX support in the processor. System software can determine whether a processor supports VMX operation using CPUID. If CPUID.1:ECX.VMX[bit 5] = 1, then VMX operation is supported. See Chapter 3, “Instruction Set Reference, A-L” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

The VMX architecture is designed to be extensible so that future processors in VMX operation can support additional features not present in first-generation implementations of the VMX architecture. The availability of extensible VMX features is reported to software using a set of VMX capability MSRs (see Appendix A, “VMX Capability Reporting Facility”).
24.7 ENABLING AND ENTERING VMX OPERATION

Before system software can enter VMX operation, it enables VMX by setting CR4.VMXE[bit 13] = 1. VMX operation is then entered by executing the VMXON instruction. VMXON causes an invalid-opcode exception (#UD) if executed with CR4.VMXE = 0. Once in VMX operation, it is not possible to clear CR4.VMXE (see Section 24.8). System software leaves VMX operation by executing the VMXOFF instruction. CR4.VMXE can be cleared outside of VMX operation after executing VMXOFF.

VMXON is also controlled by the IA32_FEATURE_CONTROL MSR (MSR address 3AH). This MSR is cleared to zero when a logical processor is reset. The relevant bits of the MSR are:

- **Bit 0 is the lock bit.** If this bit is clear, VMXON causes a general-protection exception. If the lock bit is set, WRMSR to this MSR causes a general-protection exception; the MSR cannot be modified until a power-up reset condition. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX. To enable VMX support in a platform, BIOS must set bit 1, bit 2, or both (see below), as well as the lock bit.

- **Bit 1 enables VMXON in SMX operation.** If this bit is clear, execution of VMXON in SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support both VMX operation (see Section 24.6) and SMX operation (see Chapter 7, “Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D) cause general-protection exceptions.

- **Bit 2 enables VMXON outside SMX operation.** If this bit is clear, execution of VMXON outside SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support VMX operation (see Section 24.6) cause general-protection exceptions.

**NOTE**

A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 7, “Safer Mode Extensions Reference,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

Before executing VMXON, software should allocate a naturally aligned 4-KByte region of memory that a logical processor may use to support VMX operation. This region is called the **VMXON region**. The address of the VMXON region (the VMXON pointer) is provided in an operand to VMXON. Section 25.11.5, “VMXON Region,” details how software should initialize and access the VMXON region.

24.8 RESTRICTIONS ON VMX OPERATION

VMX operation places restrictions on processor operation. These are detailed below:

- In VMX operation, processors may fix certain bits in CR0 and CR4 to specific values and not support other values. VMXON fails if any of these bits contains an unsupported value (see “VMXON—Enter VMX Operation” in Chapter 31). Any attempt to set one of these bits to an unsupported value while in VMX operation (including VMX root operation) using any of the CLTS, LMSW, or MOV CR instructions causes a general-protection exception. VM entry or VM exit cannot set any of these bits to an unsupported value. Software should consult the VMX capability MSRs IA32_VMX_CR0_FIXED0 and IA32_VMX_CR0_FIXED1 to determine how bits in CR0 are fixed (see Appendix A.7). For CR4, software should consult the VMX capability MSRs IA32_VMX_CR4_FIXED0 and IA32_VMX_CR4_FIXED1 (see Appendix A.8).

**NOTES**

The first processors to support VMX operation require that the following bits be 1 in VMX operation: CR0.PE, CR0.NE, CR0.PG, and CR4.VMXE. The restrictions on CR0.PE and CR0.PG imply that VMX

1. Future processors may require that a different amount of memory be reserved. If so, this fact is reported to software using the VMX capability-reporting mechanism.
operation is supported only in paged protected mode (including IA-32e mode). Therefore, guest software cannot be run in unpaged protected mode or in real-address mode.

Later processors support a VM-execution control called “unrestricted guest” (see Section 25.6.2). If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation (even if the capability MSR IA32_VMX_CR0_FIXED0 reports otherwise). Such processors allow guest software to run in unpaged protected mode or in real-address mode.

• VMXON fails if a logical processor is in A20M mode (see “VMXON—Enter VMX Operation” in Chapter 31). Once the processor is in VMX operation, A20M interrupts are blocked. Thus, it is impossible to be in A20M mode in VMX operation.

• The INIT signal is blocked whenever a logical processor is in VMX root operation. It is not blocked in VMX non-root operation. Instead, INITs cause VM exits (see Section 26.2, “Other Causes of VM Exits”).

• Intel® Processor Trace (Intel PT) can be used in VMX operation only if IA32_VMX_MISC[14] is read as 1 (see Appendix A.6). On processors that support Intel PT but which do not allow it to be used in VMX operation, execution of VMXON clears IA32_RTIT_CTL.TraceEn (see “VMXON—Enter VMX Operation” in Chapter 31); any attempt to write IA32_RTIT_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “unrestricted guest” VM-execution control were 0. See Section 25.6.2.
25.1 OVERVIEW

A logical processor uses virtual-machine control data structures (VMCSs) while it is in VMX operation. These manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.

A VMM can use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM can use a different VMCS for each virtual processor.

A logical processor associates a region in memory with each VMCS. This region is called the VMCS region. Software references a specific VMCS using the 64-bit physical address of the region (a VMCS pointer). VMCS pointers must be aligned on a 4-KByte boundary (bits 11:0 must be zero). These pointers must not set bits beyond the processor’s physical-address width.

A logical processor may maintain a number of VMCSs that are active. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. At any given time, at most one of the active VMCSs is the current VMCS. (This document frequently uses the term “the VMCS” to refer to the current VMCS.) The VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS.

The following items describe how a logical processor determines which VMCSs are active and which is current:

- The memory operand of the VMPTRLD instruction is the address of a VMCS. After execution of the instruction, that VMCS is both active and current on the logical processor. Any other VMCS that had been active remains so, but no other VMCS is current.
- The VMCS link pointer field in the current VMCS (see Section 25.4.2) is itself the address of a VMCS. If VM entry is performed successfully with the 1-setting of the "VMCS shadowing" VM-execution control, the VMCS referenced by the VMCS link pointer field becomes active on the logical processor. The identity of the current VMCS does not change.
- The memory operand of the VMCLEAR instruction is also the address of a VMCS. After execution of the instruction, the launch state of that VMCS is "clear".

The VMPTRST instruction stores the address of the logical processor’s current VMCS into a specified memory location (it stores the value FFFFFFFF_FFFFFFFFH if there is no current VMCS).

The launch state of a VMCS determines which VM-entry instruction should be used with that VMCS: the VMLAUNCH instruction requires a VMCS whose launch state is "clear"; the VMRESUME instruction requires a VMCS whose launch state is "launched". A logical processor maintains a VMCS’s launch state in the corresponding VMCS region. The following items describe how a logical processor manages the launch state of a VMCS:

- If the launch state of the current VMCS is "clear", successful execution of the VMLAUNCH instruction changes the launch state to "launched".
- The memory operand of the VMCLEAR instruction is the address of a VMCS. After execution of the instruction, the launch state of that VMCS is "clear".
- There are no other ways to modify the launch state of a VMCS (it cannot be modified using VMWRITE) and there is no direct way to discover it (it cannot be read using VMREAD).

1. The amount of memory required for a VMCS region is at most 4 KBytes. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix A.1).
2. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, these pointers must not set any bits in the range 63:32; see Appendix A.1.
Figure 25-1 illustrates the different states of a VMCS. It uses “X” to refer to the VMCS and “Y” to refer to any other VMCS. Thus: “VMPTRLD X” always makes X current and active; “VMPTRLD Y” always makes X not current (because it makes Y current); VMLAUNCH makes the launch state of X “launched” if X was current and its launch state was “clear”; and VMCLEAR X always makes X inactive and not current and makes its launch state “clear”.

The figure does not illustrate operations that do not modify the VMCS state relative to these parameters (e.g., execution of VMPTRLD X when X is already current). Note that VMCLEAR X makes X “inactive, not current, and clear,” even if X’s current state is not defined (e.g., even if X has not yet been initialized). See Section 25.11.3.

Figure 25-1. States of VMCS X

Because a shadow VMCS (see Section 25.10) cannot be used for VM entry, the launch state of a shadow VMCS is not meaningful. Figure 25-1 does not illustrate all the ways in which a shadow VMCS may be made active.

### 25.2 FORMAT OF THE VMCS REGION

A VMCS region comprises up to 4-KBytes. The format of a VMCS region is given in Table 25-1.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bits 30:0: VMCS revision identifier</td>
</tr>
<tr>
<td></td>
<td>Bit 31: shadow-VMCS indicator (see Section 25.10)</td>
</tr>
<tr>
<td>4</td>
<td>VMX-abort indicator</td>
</tr>
<tr>
<td>8</td>
<td>VMCS data (implementation-specific format)</td>
</tr>
</tbody>
</table>

1. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix A.1).
The first 4 bytes of the VMCS region contain the **VMCS revision identifier** at bits 30:0.¹ Processors that maintain VMCS data in different formats (see below) use different VMCS revision identifiers. These identifiers enable software to avoid using a VMCS region formatted for one processor on a processor that uses a different format.² Bit 31 of this 4-byte region indicates whether the VMCS is a shadow VMCS (see Section 25.10).

Software should write the VMCS revision identifier to the VMCS region before using that region for a VMCS. The VMCS revision identifier is never written by the processor; VMPTRLD fails if its operand references a VMCS region whose VMCS revision identifier differs from that used by the processor. (VMPTRLD also fails if the shadow-VMCS indicator is 1 and the processor does not support the 1-setting of the “VMCS shadowing” VM-execution control; see Section 25.6.2) Software can discover the VMCS revision identifier that a processor uses by reading the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

Software should clear or set the shadow-VMCS indicator depending on whether the VMCS is to be an ordinary VMCS or a shadow VMCS (see Section 25.10). VMPTRLD fails if the shadow-VMCS indicator is set and the processor does not support the 1-setting of the “VMCS shadowing” VM-execution control. Software can discover support for this setting by reading the VMX capability MSR IA32_VMX_PROCBASED_CTLS2 (see Appendix A.3.3).

The next 4 bytes of the VMCS region are used for the **VMX-abort indicator**. The contents of these bits do not control processor operation in any way. A logical processor writes a non-zero value into these bits if a VMX abort occurs (see Section 28.7). Software may also write into this field.

The remainder of the VMCS region is used for **VMCS data** (those parts of the VMCS that control VMX non-root operation and the VMX transitions). The format of these data is implementation-specific. VMCS data are discussed in Section 25.3 through Section 25.9. To ensure proper behavior in VMX operation, software should maintain the VMCS region and related structures (enumerated in Section 25.11.4) in writeback cacheable memory. Future implementations may allow or require a different memory type³. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

### 25.3 ORGANIZATION OF VMCS DATA

The VMCS data are organized into six logical groups:

- **Guest-state area.** Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries.
- **Host-state area.** Processor state is loaded from the host-state area on VM exits.
- **VM-execution control fields.** These fields control processor behavior in VMX non-root operation. They determine in part the causes of VM exits.
- **VM-exit control fields.** These fields control VM exits.
- **VM-entry control fields.** These fields control VM entries.
- **VM-exit information fields.** These fields receive information on VM exits and describe the cause and the nature of VM exits. On some processors, these fields are read-only.⁴

The VM-execution control fields, the VM-exit control fields, and the VM-entry control fields are sometimes referred to collectively as VMX controls.

---

¹. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field. For all processors produced prior to this change, bit 31 of the VMCS revision identifier was 0.

². Logical processors that use the same VMCS revision identifier use the same size for VMCS regions.

³. Alternatively, software may map any of these regions or structures with the UC memory type. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly. In addition, the processor will continue to use the memory type reported in the VMX capability MSR IA32_VMX_BASIC with exceptions noted in Appendix A.1.

⁴. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).
25.4 GUEST-STATE AREA

This section describes fields contained in the guest-state area of the VMCS. VM entries load processor state from these fields and VM exits store processor state into these fields. See Section 27.3.2 and Section 28.3 for details.

25.4.1 Guest Register State

The following fields in the guest-state area correspond to processor registers:

- Control registers CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- Debug register DR7 (64 bits; 32 bits on processors that do not support Intel 64 architecture).
- RSP, RIP, and RFLAGS (64 bits each; 32 bits on processors that do not support Intel 64 architecture).\(^1\)
- The following fields for each of the registers CS, SS, DS, ES, FS, GS, LDTR, and TR:
  - Selector (16 bits).
  - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture). The base-address fields for CS, SS, DS, and ES have only 32 architecturally-defined bits; nevertheless, the corresponding VMCS fields have 64 bits on processors that support Intel 64 architecture.
  - Segment limit (32 bits). The limit field is always a measure in bytes.
  - Access rights (32 bits). The format of this field is given in Table 25-2 and detailed as follows:
    - The low 16 bits correspond to bits 23:8 of the upper 32 bits of a 64-bit segment descriptor. While bits 19:16 of code-segment and data-segment descriptors correspond to the upper 4 bits of the segment limit, the corresponding bits (bits 11:8) are reserved in this VMCS field.
    - Bit 16 indicates an unusable segment. Attempts to use such a segment fault except in 64-bit mode. In general, a segment register is unusable if it has been loaded with a null selector.\(^2\)
    - Bits 31:17 are reserved.

\(^1\) This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

\(^2\) There are a few exceptions to this statement. For example, a segment with a non-null selector may be unusable following a task switch that fails after its commit point; see “Interrupt 10—Invalid TSS Exception (#TS)” in Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. In contrast, the TR register is usable after processor reset despite having a null selector; see Table 11-1 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

### Table 25-2. Format of Access Rights

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>Segment type</td>
</tr>
<tr>
<td>4</td>
<td>S — Descriptor type (0 = system; 1 = code or data)</td>
</tr>
<tr>
<td>6:5</td>
<td>DPL — Descriptor privilege level</td>
</tr>
<tr>
<td>7</td>
<td>P — Segment present</td>
</tr>
<tr>
<td>11:8</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>AVL — Available for use by system software</td>
</tr>
</tbody>
</table>

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.

2. There are a few exceptions to this statement. For example, a segment with a non-null selector may be unusable following a task switch that fails after its commit point; see “Interrupt 10—Invalid TSS Exception (#TS)” in Section 6.14, “Exception and Interrupt Handling in 64-bit Mode,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. In contrast, the TR register is usable after processor reset despite having a null selector; see Table 11-1 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
The base address, segment limit, and access rights compose the "hidden" part (or "descriptor cache") of each segment register. These data are included in the VMCS because it is possible for a segment register's descriptor cache to be inconsistent with the segment descriptor in memory (in the GDT or the LDT) referenced by the segment register's selector.

The value of the DPL field for SS is always equal to the logical processor's current privilege level (CPL).1

On some processors, executions of VMWRITE ignore attempts to write non-zero values to any of bits 11:8 or bits 31:17. On such processors, VMREAD always returns 0 for those bits, and VM entry treats those bits as if they were all 0 (see Section 27.3.1.2).

- The following fields for each of the registers GDTR and IDTR:
  - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture).
  - Limit (32 bits). The limit fields contain 32 bits even though these fields are specified as only 16 bits in the architecture.

- The following MSRs:
  - IA32_DEBUGCTL (64 bits)
  - IA32_SYSENTER_CS (32 bits)
  - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture)
  - IA32_PERF_GLOBAL_CTRL (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-entry control.
  - IA32_PAT (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_PAT" VM-entry control or that of the "save IA32_PAT" VM-exit control.
  - IA32_EFER (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_EFER" VM-entry control or that of the "save IA32_EFER" VM-exit control.
  - IA32_BNDCFGS (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_BNDCFGS" VM-entry control or that of the "clear IA32_BNDCFGS" VM-exit control.
  - IA32_RTIT_CTL (64 bits). This field is supported only on processors that support either the 1-setting of the "load IA32_RTIT_CTL" VM-entry control or that of the "clear IA32_RTIT_CTL" VM-exit control.
  - IA32_INTERRUPT_SSP_TABLE_ADDR (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-entry control.

The following is a table for the format of access rights:

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Field</th>
</tr>
</thead>
</table>
| 13              | Reserved (except for CS)
|                 | L — 64-bit mode active (for CS only)      |
| 14              | D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment) |
| 15              | G — Granularity                            |
| 16              | Segment unusable (0 = usable; 1 = unusable) |
| 31:17           | Reserved                                   |

The base address, segment limit, and access rights compose the "hidden" part (or "descriptor cache") of each segment register. These data are included in the VMCS because it is possible for a segment register's descriptor cache to be inconsistent with the segment descriptor in memory (in the GDT or the LDT) referenced by the segment register's selector.

The value of the DPL field for SS is always equal to the logical processor's current privilege level (CPL).1

1. In protected mode, CPL is also associated with the RPL field in the CS selector. However, the RPL fields are not meaningful in real-address mode or in virtual-8086 mode.
— IA32_PKRS (64 bits). This field is supported only on processors that support the 1-setting of the “load PKRS” VM-entry control.

• The shadow-stack pointer register SSP (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the “load CET state” VM-entry control.

• The register SMBASE (32 bits). This register contains the base address of the logical processor’s SMRAM image.

### 25.4.2 Guest Non-Register State

In addition to the register state described in Section 25.4.1, the guest-state area includes the following fields that characterize guest state but which do not correspond to processor registers:

- **Activity state** (32 bits). This field identifies the logical processor’s activity state. When a logical processor is executing instructions normally, it is in the **active state**. Execution of certain instructions and the occurrence of certain events may cause a logical processor to transition to an **inactive state** in which it ceases to execute instructions.

  The following activity states are defined:

  — 0: **Active**. The logical processor is executing instructions normally.

  — 1: **HLT**. The logical processor is inactive because it executed the HLT instruction.

  — 2: **Shutdown**. The logical processor is inactive because it incurred a **triple fault**\(^1\) or some other serious error.

  — 3: **Wait-for-SIPI**. The logical processor is inactive because it is waiting for a startup-IPI (SIPI).

  Future processors may include support for other activity states. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine what activity states are supported.

- **Interruptibility state** (32 bits). The IA-32 architecture includes features that permit certain events to be blocked for a period of time. This field contains information about such blocking. Details and the format of this field are given in Table 25-3.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Bit Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Blocking by STI</td>
<td>See the “STI—Set Interrupt Flag” section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B. Execution of STI with RFLAGS.IF = 0 blocks maskable interrupts on the instruction boundary following its execution.(^1) Setting this bit indicates that this blocking is in effect.</td>
</tr>
<tr>
<td>1</td>
<td>Blocking by MOV SS</td>
<td>See Section 6.8.3, “Masking Exceptions and Interrupts When Switching Stacks,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. Execution of a MOV to SS or a POP to SS blocks or suppresses certain debug exceptions as well as interrupts (maskable and nonmaskable) on the instruction boundary following its execution. Setting this bit indicates that this blocking is in effect.(^2) This document uses the term “blocking by MOV SS,” but it applies equally to POP SS.</td>
</tr>
<tr>
<td>2</td>
<td>Blocking by SMI</td>
<td>See Section 32.2, “System Management Interrupt (SMI).” System-management interrupts (SMIs) are disabled while the processor is in system-management mode (SMM). Setting this bit indicates that blocking of SMIs is in effect.</td>
</tr>
</tbody>
</table>

1. Execution of the MWAIT instruction may put a logical processor into an inactive state. However, this VMCS field never reflects this state. See Section 28.1.

2. A triple fault occurs when a logical processor encounters an exception while attempting to deliver a double fault.
VIRTUAL MACHINE CONTROL STRUCTURES

Table 25-3. Format of Interruptibility State (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Bit Name</th>
<th>Notes</th>
</tr>
</thead>
</table>
| 3               | Blocking by NMI        | See Section 6.7.1, “Handling Multiple NMIs,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A and Section 32.8, “NMI Handling While in SMM.” Delivery of a non-maskable interrupt (NMI) or a system-management interrupt (SMI) blocks subsequent NMIs until the next execution of IRET. See Section 26.3 for how this behavior of IRET may change in VMX non-root operation. Setting this bit indicates that blocking of NMIs is in effect. Clearing this bit does not imply that NMIs are not (temporarily) blocked for other reasons.
|                 |                        | If the “virtual NMIs” VM-execution control (see Section 25.6.1) is 1, this bit does not control the blocking of NMIs. Instead, it refers to “virtual-NMI blocking” (the fact that guest software is not ready for an NMI). |
| 4               | Enclave interruption   | Set to 1 if the VM exit occurred while the logical processor was in enclave mode. Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode. A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified. |
| 31:5            | Reserved               | VM entry will fail if these bits are not 0. See Section 27.3.1.5.     |

NOTES:
1. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.
2. System-management interrupts may also be inhibited on the instruction boundary following such an execution of MOV or POP.

- Pending debug exceptions (64 bits; 32 bits on processors that do not support Intel 64 architecture). IA-32 processors may recognize one or more debug exceptions without immediately delivering them. This field contains information about such exceptions. This field is described in Table 25-4.

Table 25-4. Format of Pending-Debug-Exceptions

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Bit Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>B3 – B0</td>
<td>When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if the corresponding enabling bit in DR7 is not set.</td>
</tr>
<tr>
<td>10:4</td>
<td>Reserved</td>
<td>VM entry fails if these bits are not 0. See Section 27.3.1.5.</td>
</tr>
<tr>
<td>11</td>
<td>BLD</td>
<td>When set, this bit indicates that a bus lock was asserted while OS bus-lock detection was enabled and CPL &gt; 0 (see Section 18.3.1.6, “OS Bus-Lock Detection”).</td>
</tr>
<tr>
<td>12</td>
<td>Enabled breakpoint</td>
<td>When set, this bit indicates that at least one data or I/O breakpoint was met and was enabled in DR7; the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled; or a bus lock was asserted while CPL &gt; 0 and OS bus-lock detection had been enabled.</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
<td>VM entry fails if this bit is not 0. See Section 27.3.1.5.</td>
</tr>
</tbody>
</table>

1. For example, execution of a MOV to SS or a POP to SS may inhibit some debug exceptions for one instruction. See Section 6.8.3 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. In addition, certain events incident to an instruction (for example, an INIT signal) may take priority over debug traps generated by that instruction. See Table 6-2 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
VIRTUAL MACHINE CONTROL STRUCTURES

Table 25-4. Format of Pending-Debug-Exceptions (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Bit Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>BS</td>
<td>When set, this bit indicates that a debug exception would have been triggered by single-step execution mode.</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td>VM entry fails if this bit is not 0. See Section 27.3.1.5.</td>
</tr>
<tr>
<td>16</td>
<td>RTM</td>
<td>When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 16.3.7, “RTM-Enabled Debugger Support,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).²</td>
</tr>
<tr>
<td>63:17</td>
<td>Reserved</td>
<td>VM entry fails if these bits are not 0. See Section 27.3.1.5. Bits 63:32 exist only on processors that support Intel 64 architecture.</td>
</tr>
</tbody>
</table>

NOTES:
1. In general, the format of this field matches that of DR6. However, DR6 clears bit 11 to indicate detection of a bus lock, while this field sets the bit to indicate that condition.
2. In general, the format of this field matches that of DR6. However, DR6 clears bit 16 to indicate an RTM-related exception, while this field sets the bit to indicate that condition.

- **VMCS link pointer** (64 bits). If the “VMCS shadowing” VM-execution control is 1, the VMREAD and VMWRITE instructions access the VMCS referenced by this pointer (see Section 25.10). Otherwise, software should set this field to FFFFFFFF_FFFFFFFFH to avoid VM-entry failures (see Section 27.3.1.5).
- **VMX-preemption timer value** (32 bits). This field is supported only on processors that support the 1-setting of the “activate VMX-preemption timer” VM-execution control. This field contains the value that the VMX-preemption timer will use following the next VM entry with that setting. See Section 26.5.1 and Section 27.7.4.
- **Page-directory-pointer-table entries** (PDPTEs; 64 bits each). These four (4) fields (PDPTE0, PDPTE1, PDPTE2, and PDPTE3) are supported only on processors that support the 1-setting of the “enable EPT” VM-execution control. They correspond to the PDPTEs referenced by CR3 when PAE paging is in use (see Section 4.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). They are used only if the “enable EPT” VM-execution control is 1.
- **Guest interrupt status** (16 bits). This field is supported only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control. It characterizes part of the guest’s virtual-APIC state and does not correspond to any processor or APIC registers. It comprises two 8-bit subfields:
  - **Requesting virtual interrupt (RVI).** This is the low byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is requesting service. (The value 0 implies that there is no such interrupt.)
  - **Servicing virtual interrupt (SVI).** This is the high byte of the guest interrupt status. The processor treats this value as the vector of the highest priority virtual interrupt that is in service. (The value 0 implies that there is no such interrupt.)

See Chapter 30 for more information on the use of this field.

- **PML index** (16 bits). This field is supported only on processors that support the 1-setting of the “enable PML” VM-execution control. It contains the logical index of the next entry in the page-modification log. Because the page-modification log comprises 512 entries, the PML index is typically a value in the range 0–511. Details of the page-modification log and use of the PML index are given in Section 29.3.6.

### 25.5 HOST-STATE AREA

This section describes fields contained in the host-state area of the VMCS. As noted earlier, processor state is loaded from these fields on every VM exit (see Section 28.5).

All fields in the host-state area correspond to processor registers:
• CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
• RSP and RIP (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
• Selector fields (16 bits each) for the segment registers CS, SS, DS, ES, FS, GS, and TR. There is no field in the host-state area for the LDTR selector.
• Base-address fields for FS, GS, TR, GDTR, and IDTR (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
• The following MSRs:
  — IA32_SYSENTER_CS (32 bits)
  — IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture).
  — IA32_PERF_GLOBAL_CTRL (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-exit control.
  — IA32_PAT (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_PAT" VM-exit control.
  — IA32_EFER (64 bits). This field is supported only on processors that support the 1-setting of the "load IA32_EFER" VM-exit control.
  — IA32_S_CET (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-exit control.
  — IA32_INTERRUPT_SSP_TABLE_ADDR (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-exit control.
  — IA32_PKRS (64 bits). This field is supported only on processors that support the 1-setting of the "load PKRS" VM-exit control.
• The shadow-stack pointer register SSP (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is supported only on processors that support the 1-setting of the "load CET state" VM-exit control.

In addition to the state identified here, some processor state components are loaded with fixed values on every VM exit; there are no fields corresponding to these components in the host-state area. See Section 28.5 for details of how state is loaded on VM exits.

25.6 VM-EXECUTION CONTROL FIELDS

The VM-execution control fields govern VMX non-root operation. These are described in Section 25.6.1 through Section 25.6.8.

25.6.1 Pin-Based VM-Execution Controls

The pin-based VM-execution controls constitute a 32-bit vector that governs the handling of asynchronous events (for example: interrupts). Table 25-5 lists the controls. See Chapter 28 for how these controls affect processor behavior in VMX non-root operation.

---

1. Some asynchronous events cause VM exits regardless of the settings of the pin-based VM-execution controls (see Section 26.2).
VIRTUAL MACHINE CONTROL STRUCTURES

Table 25-5. Definitions of Pin-Based VM-Execution Controls

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>External-interrupt exiting</td>
<td>If this control is 1, external interrupts cause VM exits. Otherwise, they are delivered normally through the guest interrupt-descriptor table (IDT). If this control is 1, the value of RFLAGS.IF does not affect interrupt blocking.</td>
</tr>
<tr>
<td>3</td>
<td>NMI exiting</td>
<td>If this control is 1, non-maskable interrupts (NMIs) cause VM exits. Otherwise, they are delivered normally using descriptor 2 of the IDT. This control also determines interactions between IRET and blocking by NMI (see Section 26.3).</td>
</tr>
<tr>
<td>5</td>
<td>Virtual NMI</td>
<td>If this control is 1, NMIs are never blocked and the “blocking by NMI” bit (bit 3) in the interruptibility-state field indicates “virtual-NMI blocking” (see Table 25-3). This control also interacts with the “NMI-window exiting” VM-execution control (see Section 25.6.2).</td>
</tr>
<tr>
<td>6</td>
<td>Activate VMX-preemption timer</td>
<td>If this control is 1, the VMX-preemption timer counts down in VMX non-root operation; see Section 26.5.1. A VM exit occurs when the timer counts down to zero; see Section 26.2.</td>
</tr>
<tr>
<td>7</td>
<td>Process posted interrupts</td>
<td>If this control is 1, the processor treats interrupts with the posted-interrupt notification vector (see Section 25.6.8) specially, updating the virtual-APIC page with posted-interrupt requests (see Section 30.6).</td>
</tr>
</tbody>
</table>

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_PINBASED_CTLS and IA32_VMX_TRUE_PINBASED_CTLS (see Appendix A.3.1) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 27.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 2, and 4. The VMX capability MSR IA32_VMX_PINBASED_CTLS will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_PINBASED_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

25.6.2 Processor-Based VM-Execution Controls

The processor-based VM-execution controls constitute three vectors that govern the handling of synchronous events, mainly those caused by the execution of specific instructions. These are the primary processor-based VM-execution controls (32 bits), the secondary processor-based VM-execution controls (32 bits), and the tertiary VM-execution controls (64 bits).

Table 25-6 lists the primary processor-based VM-execution controls. See Chapter 26 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 25-6. Definitions of Primary Processor-Based VM-Execution Controls

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Interrupt-window exiting</td>
<td>If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 25.4.2).</td>
</tr>
<tr>
<td>3</td>
<td>Use TSC offsetting</td>
<td>This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the I32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 25.6.5 and Section 26.3).</td>
</tr>
<tr>
<td>7</td>
<td>HLT exiting</td>
<td>This control determines whether executions of HLT cause VM exits.</td>
</tr>
<tr>
<td>9</td>
<td>INVLPG exiting</td>
<td>This determines whether executions of INVLPG cause VM exits.</td>
</tr>
<tr>
<td>10</td>
<td>MWAIT exiting</td>
<td>This control determines whether executions of MWAIT cause VM exits.</td>
</tr>
<tr>
<td>11</td>
<td>RDPMC exiting</td>
<td>This control determines whether executions of RDPMC cause VM exits.</td>
</tr>
</tbody>
</table>

1. Some instructions cause VM exits regardless of the settings of the processor-based VM-execution controls (see Section 26.1.2), as do task switches (see Section 26.2).
Table 25-6. Definitions of Primary Processor-Based VM-Execution Controls (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>RDTSC exiting</td>
<td>This control determines whether executions of RDTSC and RDTSCP cause VM exits.</td>
</tr>
<tr>
<td>15</td>
<td>CR3-load exiting</td>
<td>In conjunction with the CR3-target controls (see Section 25.6.7), this control determines whether executions of MOV to CR3 cause VM exits. See Section 26.1.3. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.</td>
</tr>
<tr>
<td>16</td>
<td>CR3-store exiting</td>
<td>This control determines whether executions of MOV from CR3 cause VM exits. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.</td>
</tr>
<tr>
<td>17</td>
<td>Activate tertiary controls</td>
<td>This control determines whether the tertiary processor-based VM-execution controls are used. If this control is 0, the logical processor operates as if all the tertiary processor-based VM-execution controls were also 0.</td>
</tr>
<tr>
<td>19</td>
<td>CR8-load exiting</td>
<td>This control determines whether executions of MOV to CR8 cause VM exits.</td>
</tr>
<tr>
<td>20</td>
<td>CR8-store exiting</td>
<td>This control determines whether executions of MOV from CR8 cause VM exits.</td>
</tr>
<tr>
<td>21</td>
<td>Use TPR shadow</td>
<td>Setting this control to 1 enables TPR virtualization and other APIC-virtualization features. See Chapter 30.</td>
</tr>
<tr>
<td>22</td>
<td>NMI-window exiting</td>
<td>If this control is 1, a VM exit occurs at the beginning of any instruction if there is no virtual-NMI blocking (see Section 25.4.2).</td>
</tr>
<tr>
<td>23</td>
<td>MOV-DR exiting</td>
<td>This control determines whether executions of MOV DR cause VM exits.</td>
</tr>
<tr>
<td>24</td>
<td>Unconditional I/O exiting</td>
<td>This control determines whether executions of I/O instructions (IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD) cause VM exits.</td>
</tr>
<tr>
<td>25</td>
<td>Use I/O bitmaps</td>
<td>This control determines whether I/O bitmaps are used to restrict executions of I/O instructions (see Section 25.6.4 and Section 26.1.3). For this control, “0” means “do not use I/O bitmaps” and “1” means “use I/O bitmaps.” If the I/O bitmaps are used, the setting of the “unconditional I/O exiting” control is ignored.</td>
</tr>
<tr>
<td>27</td>
<td>Monitor trap flag</td>
<td>If this control is 1, the monitor trap flag debugging feature is enabled. See Section 26.5.2.</td>
</tr>
<tr>
<td>28</td>
<td>Use MSR bitmaps</td>
<td>This control determines whether MSR bitmaps are used to control execution of the RDMSR and WRMSR instructions (see Section 25.6.9 and Section 26.1.3). For this control, “0” means “do not use MSR bitmaps” and “1” means “use MSR bitmaps.” If the MSR bitmaps are not used, all executions of the RDMSR and WRMSR instructions cause VM exits.</td>
</tr>
<tr>
<td>29</td>
<td>MONITOR exiting</td>
<td>This control determines whether executions of MONITOR cause VM exits.</td>
</tr>
<tr>
<td>30</td>
<td>PAUSE exiting</td>
<td>This control determines whether executions of PAUSE cause VM exits.</td>
</tr>
<tr>
<td>31</td>
<td>Activate secondary controls</td>
<td>This control determines whether the secondary processor-based VM-execution controls are used. If this control is 0, the logical processor operates as if all the secondary processor-based VM-execution controls were also 0.</td>
</tr>
</tbody>
</table>

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_PROCBASED_CTLS and IA32_VMX_TRUE_PROCBASED_CTLS (see Appendix A.3.2) to determine how to set reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 27.2.1.1).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 1, 4–6, 8, 13–16, and 26. The VMX capability MSR IA32_VMX_PROCBASED_CTLS will always report that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_PROCBASED_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

Bit 31 of the primary processor-based VM-execution controls determines whether the secondary processor-based VM-execution controls are used. If that bit is 0, VM entry and VMX non-root operation function as if all the secondary processor-based VM-execution controls were 0. Processors that support only the 0-setting of bit 31 of
the primary processor-based VM-execution controls do not support the secondary processor-based VM-execution controls.

Table 25-7 lists the secondary processor-based VM-execution controls. See Chapter 26 for more details of how these controls affect processor behavior in VMX non-root operation.

**Table 25-7. Definitions of Secondary Processor-Based VM-Execution Controls**

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Virtualize APIC accesses</td>
<td>If this control is 1, the logical processor treats specially accesses to the page with the APIC-access address. See Section 30.4.</td>
</tr>
<tr>
<td>1</td>
<td>Enable EPT</td>
<td>If this control is 1, extended page tables (EPT) are enabled. See Section 29.3.</td>
</tr>
<tr>
<td>2</td>
<td>Descriptor-table exiting</td>
<td>This control determines whether executions of LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, and STR cause VM exits.</td>
</tr>
<tr>
<td>3</td>
<td>Enable RDTSCP</td>
<td>If this control is 0, any execution of RDTSCP causes an invalid-opcode exception (#UD).</td>
</tr>
<tr>
<td>4</td>
<td>Virtualize x2APIC mode</td>
<td>If this control is 1, the logical processor treats specially RDMSR and WRMSR to APIC MSRs (in the range 800H–8FFH). See Section 30.5.</td>
</tr>
<tr>
<td>5</td>
<td>Enable VPID</td>
<td>If this control is 1, cached translations of linear addresses are associated with a virtual-processor identifier (VPID). See Section 29.1.</td>
</tr>
<tr>
<td>6</td>
<td>WBINVD exiting</td>
<td>This control determines whether executions of WBINVD and WNOINVD cause VM exits.</td>
</tr>
<tr>
<td>7</td>
<td>Unrestricted guest</td>
<td>This control determines whether guest software may run in unpaged protected mode or in real-address mode.</td>
</tr>
<tr>
<td>8</td>
<td>APIC-register virtualization</td>
<td>If this control is 1, the logical processor virtualizes certain APIC accesses. See Section 30.4 and Section 30.5.</td>
</tr>
<tr>
<td>9</td>
<td>Virtual-interrupt delivery</td>
<td>This control enables the evaluation and delivery of pending virtual interrupts as well as the emulation of writes to the APIC registers that control interrupt prioritization.</td>
</tr>
<tr>
<td>10</td>
<td>PAUSE-loop exiting</td>
<td>This control determines whether a series of executions of PAUSE can cause a VM exit (see Section 25.6.13 and Section 26.1.3).</td>
</tr>
<tr>
<td>11</td>
<td>RDRAND exiting</td>
<td>This control determines whether executions of RDRAND cause VM exits.</td>
</tr>
<tr>
<td>12</td>
<td>Enable INVPCID</td>
<td>If this control is 0, any execution of INVPCID causes a #UD.</td>
</tr>
<tr>
<td>13</td>
<td>Enable VM functions</td>
<td>Setting this control to 1 enables use of the VMFUNC instruction in VMX non-root operation. See Section 26.5.6.</td>
</tr>
<tr>
<td>14</td>
<td>VMCS shadowing</td>
<td>If this control is 1, executions of VMREAD and VMWRITE in VMX non-root operation may access a shadow VMCS (instead of causing VM exits). See Section 25.10 and Section 31.3.</td>
</tr>
<tr>
<td>15</td>
<td>Enable ENCLS exiting</td>
<td>If this control is 1, executions of ENCLS consult the ENCLS-exiting bitmap to determine whether the instruction causes a VM exit. See Section 25.6.16 and Section 26.1.3.</td>
</tr>
<tr>
<td>16</td>
<td>RDSEED exiting</td>
<td>This control determines whether executions of RDSEED cause VM exits.</td>
</tr>
<tr>
<td>17</td>
<td>Enable PML</td>
<td>If this control is 1, an access to a guest-physical address that sets an EPT dirty bit first adds an entry to the page-modification log. See Section 29.3.6.</td>
</tr>
<tr>
<td>18</td>
<td>EPT-violation #VE</td>
<td>If this control is 1, EPT violations may cause virtualization exceptions (#VE) instead of VM exits. See Section 26.5.7.</td>
</tr>
<tr>
<td>19</td>
<td>Conceal VMX from PT</td>
<td>If this control is 1, Intel Processor Trace suppresses from PIPs an indication that the processor was in VMX non-root operation and omits a VMCS packet from any PSB+ produced in VMX non-root operation (see Chapter 33).</td>
</tr>
<tr>
<td>20</td>
<td>Enable XSAVES/XRSTORS</td>
<td>If this control is 0, any execution of XSAVES or XRSTORS causes a #UD.</td>
</tr>
<tr>
<td>21</td>
<td>PASID translation</td>
<td>If this control is 1, PASID translation is performed for executions of ENQCMD and ENQCMDS. See Section 26.5.8.</td>
</tr>
<tr>
<td>22</td>
<td>Mode-based execute control for EPT</td>
<td>If this control is 1, EPT execute permissions are based on whether the linear address being accessed is supervisor mode or user mode. See Chapter 29.</td>
</tr>
</tbody>
</table>
Table 25-7. Definitions of Secondary Processor-Based VM-Execution Controls (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Sub-page write permissions for EPT</td>
<td>If this control is 1, EPT write permissions may be specified at the granularity of 128 bytes. See Section 29.3.4.</td>
</tr>
<tr>
<td>24</td>
<td>Intel PT uses guest physical addresses</td>
<td>If this control is 1, all output addresses used by Intel Processor Trace are treated as guest-physical addresses and translated using EPT. See Section 26.5.4.</td>
</tr>
<tr>
<td>25</td>
<td>Use TSC scaling</td>
<td>This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC multiplier field (see Section 25.6.5 and Section 26.3).</td>
</tr>
<tr>
<td>26</td>
<td>Enable user wait and pause</td>
<td>If this control is 0, any execution of TPAUSE, UMONITOR, or UMWAIT causes a #UD.</td>
</tr>
<tr>
<td>27</td>
<td>Enable PCONFIG</td>
<td>If this control is 0, any execution of PCONFIG causes a #UD.</td>
</tr>
<tr>
<td>28</td>
<td>Enable ENCLV exiting</td>
<td>If this control is 1, executions of ENCLV consult the ENCLV-exiting bitmap to determine whether the instruction causes a VM exit. See Section 25.6.17 and Section 26.1.3.</td>
</tr>
<tr>
<td>30</td>
<td>VMM bus-lock detection</td>
<td>This control determines whether assertion of a bus lock causes a VM exit. See Section 26.2.</td>
</tr>
<tr>
<td>31</td>
<td>Instruction timeout</td>
<td>If this control is 1, a VM exit occurs if certain operations prevent the processor from reaching an instruction boundary within a specified amount of time. See Section 25.6.25 and Section 26.2.</td>
</tr>
</tbody>
</table>

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_PROC-BASED_CTL32 (see Appendix A.3.3) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 27.2.1.1).

Bit 17 of the primary processor-based VM-execution controls determines whether the tertiary processor-based VM-execution controls are used. If that bit is 0, VM entry and VMX non-root operation function as if all the tertiary processor-based VM-execution controls were 0. Processors that support only the 0-setting of bit 17 of the primary processor-based VM-execution controls do not support the tertiary processor-based VM-execution controls.

Table 25-8 lists the tertiary processor-based VM-execution controls. See Chapter 26 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 25-8. Definitions of Tertiary Processor-Based VM-Execution Controls

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOADIWKEY exiting</td>
<td>This control determines whether executions of LOADIWKEY cause VM exits.</td>
</tr>
<tr>
<td>1</td>
<td>Enable HLAT</td>
<td>This control enables hypervisor-managed linear-address translation. See Section 4.5.1.</td>
</tr>
<tr>
<td>2</td>
<td>EPT paging-write control</td>
<td>If this control is 1, EPT permissions can be specified to allow writes only for paging-related updates. See Section 29.3.3.2.</td>
</tr>
<tr>
<td>3</td>
<td>Guest-paging verification</td>
<td>If this control is 1, EPT permissions can be specified to prevent accesses using linear addresses whose translation has certain properties. See Section 29.3.3.2.</td>
</tr>
<tr>
<td>4</td>
<td>IPI virtualization</td>
<td>If this control is 1, virtualization of interprocessor interrupts (IPIs) is enabled. See Section 30.1.6.</td>
</tr>
<tr>
<td>7</td>
<td>Virtualize IA32_SPEC_CTRL</td>
<td>If this control is 1, the operation of the RDMSR and WRMSR instructions is changed when accessing the IA32_SPEC_CTRL MSR. See Section 26.3.</td>
</tr>
</tbody>
</table>

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_PROC-BASED_CTL33 (see Appendix A.3.4) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 27.2.1.1).
25.6.3 Exception Bitmap

The exception bitmap is a 32-bit field that contains one bit for each exception. When an exception occurs, its vector is used to select a bit in this field. If the bit is 1, the exception causes a VM exit. If the bit is 0, the exception is delivered normally through the IDT, using the descriptor corresponding to the exception’s vector.

Whether a page fault (exception with vector 14) causes a VM exit is determined by bit 14 in the exception bitmap as well as the error code produced by the page fault and two 32-bit fields in the VMCS (the page-fault error-code mask and page-fault error-code match). See Section 26.2 for details.

25.6.4 I/O-Bitmap Addresses

The VM-execution control fields include the 64-bit physical addresses of I/O bitmaps A and B (each of which are 4 KBytes in size). I/O bitmap A contains one bit for each I/O port in the range 0000H through 7FFFH; I/O bitmap B contains bits for ports in the range 8000H through FFFFH.

A logical processor uses these bitmaps if and only if the “use I/O bitmaps” control is 1. If the bitmaps are used, execution of an I/O instruction causes a VM exit if any bit in the I/O bitmaps corresponding to a port it accesses is 1. See Section 26.1.3 for details. If the bitmaps are used, their addresses must be 4-KByte aligned.

25.6.5 Time-Stamp Counter Offset and Multiplier

The VM-execution control fields include a 64-bit TSC-offset field. If the “RDTSC exiting” control is 0 and the “use TSC offsetting” control is 1, this field controls executions of the RDTSC and RDTSCP instructions. It also controls executions of the RDMSR instruction that read from the IA32_TIME_STAMP_COUNTER MSR. For all of these, the value of the TSC offset is added to the value of the time-stamp counter, and the sum is returned to guest software in EDX:EAX.

Processors that support the 1-setting of the “use TSC scaling” control also support a 64-bit TSC-multiplier field. If this control is 1 (and the “RDTSC exiting” control is 0 and the “use TSC offsetting” control is 1), this field also affects the executions of the RDTSC, RDTSCP, and RDMSR instructions identified above. Specifically, the contents of the time-stamp counter is first multiplied by the TSC multiplier before adding the TSC offset.

See Chapter 26 for a detailed treatment of the behavior of RDTSC, RDTSCP, and RDMSR in VMX non-root operation.

25.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4

VM-execution control fields include guest/host masks and read shadows for the CR0 and CR4 registers. These fields control executions of instructions that access those registers (including CLTS, LMSW, MOV CR, and SMSW). They are 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not.

In general, bits set to 1 in a guest/host mask correspond to bits “owned” by the host:

- Guest attempts to set them (using CLTS, LMSW, or MOV to CR) to values differing from the corresponding bits in the corresponding read shadow cause VM exits.
- Guest reads (using MOV from CR or SMSW) return values for these bits from the corresponding read shadow.

Bits cleared to 0 correspond to bits “owned” by the guest; guest attempts to modify them succeed and guest reads return values for these bits from the control register itself.

See Chapter 28 for details regarding how these fields affect VMX non-root operation.

25.6.7 CR3-Target Controls

The VM-execution control fields include a set of 4 CR3-target values and a CR3-target count. The CR3-target values each have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not. The CR3-target count has 32 bits on all processors.
An execution of MOV to CR3 in VMX non-root operation does not cause a VM exit if its source operand matches one of these values. If the CR3-target count is \( n \), only the first \( n \) CR3-target values are considered; if the CR3-target count is 0, MOV to CR3 always causes a VM exit.

There are no limitations on the values that can be written for the CR3-target values. VM entry fails (see Section 27.2) if the CR3-target count is greater than 4.

Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine the number of values supported.

### 25.6.8 Controls for APIC Virtualization

There are three mechanisms by which software accesses registers of the logical processor’s local APIC:

- If the local APIC is in xAPIC mode, it can perform memory-mapped accesses to addresses in the 4-KByte page referenced by the physical address in the IA32_APIC_BASE MSR (see Section 11.4.4, “Local APIC Status and Location,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, and the Intel® 64 Architecture Processor Topology Enumeration Technical Paper).

- If the local APIC is in x2APIC mode, it can access the local APIC’s registers using the RDMSR and WRMSR instructions (see the Intel® 64 Architecture Processor Topology Enumeration Technical Paper).

- In 64-bit mode, it can access the local APIC’s task-priority register (TPR) using the MOV CR8 instruction.

Several processor-based VM-execution controls (see Section 25.6.2) control such accesses. These are “use TPR shadow”, “virtualize APIC accesses”, “virtualize x2APIC mode”, “virtual-interrupt delivery”, “APIC-register virtualization”, and “IPI virtualization”. These controls interact with the following fields:

- **APIC-access address** (64 bits). This field contains the physical address of the 4-KByte APIC-access page. If the “virtualize APIC accesses” VM-execution control is 1, access to this page may cause VM exits or be virtualized by the processor. See Section 30.4.

- **Virtual-APIC address** (64 bits). This field contains the physical address of the 4-KByte virtual-APIC page. The processor uses the virtual-APIC page to virtualize certain accesses to APIC registers and to manage virtual interrupts; see Chapter 30.

Depending on the setting of the controls indicated earlier, the virtual-APIC page may be accessed by the following operations:

- The MOV CR8 instructions (see Section 30.3).
- Accesses to the APIC-access page if, in addition, the “virtualize APIC accesses” VM-execution control is 1 (see Section 30.4).
- The RDMSR and WRMSR instructions if, in addition, the value of ECX is in the range 800H–8FFH (indicating an APIC MSR) and the “virtualize x2APIC mode” VM-execution control is 1 (see Section 30.5).

If the “use TPR shadow” VM-execution control is 1, VM entry ensures that the virtual-APIC address is 4-KByte aligned. The virtual-APIC address exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.

- **TPR threshold** (32 bits). Bits 3:0 of this field determine the threshold below which bits 7:4 of VTPR (see Section 30.1.1) cannot fall. If the “virtual-interrupt delivery” VM-execution control is 0, a VM exit occurs after an operation (e.g., an execution of MOV to CR8) that reduces the value of those bits below the TPR threshold. See Section 30.1.2.

The TPR threshold exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.

- **EOI-exit bitmap** (4 fields; 64 bits each). These fields are supported only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control. They are used to determine which virtualized writes to the APIC’s EOI register cause VM exits:

---

1. If the local APIC does not support x2APIC mode, it is always in xAPIC mode.
VIRTUAL MACHINE CONTROL STRUCTURES

— EOI_EXIT0 contains bits for vectors from 0 (bit 0) to 63 (bit 63).
— EOI_EXIT1 contains bits for vectors from 64 (bit 0) to 127 (bit 63).
— EOI_EXIT2 contains bits for vectors from 128 (bit 0) to 191 (bit 63).
— EOI_EXIT3 contains bits for vectors from 192 (bit 0) to 255 (bit 63).
See Section 30.1.4 for more information on the use of this field.

• Posted-interrupt notification vector (16 bits). This field is supported only on processors that support the 1-setting of the "process posted interrupts" VM-execution control. Its low 8 bits contain the interrupt vector that is used to notify a logical processor that virtual interrupts have been posted. See Section 30.6 for more information on the use of this field.

• Posted-interrupt descriptor address (64 bits). This field is supported only on processors that support the 1-setting of the "process posted interrupts" VM-execution control. It is the physical address of a 64-byte aligned posted interrupt descriptor. See Section 30.6 for more information on the use of this field.

• PID-pointer table address (64 bits). This field contains the physical address of the PID-pointer table. If the "IPI virtualization" VM-execution control is 1, the logical processor uses entries in this table to virtualize IPIs. See Section 30.1.6.

• Last PID-pointer index (16 bits). This field contains the index of the last entry in the PID-pointer table.

25.6.9 MSR-Bitmap Address

On processors that support the 1-setting of the "use MSR bitmaps" VM-execution control, the VM-execution control fields include the 64-bit physical address of four contiguous MSR bitmaps, which are each 1-KByte in size. This field does not exist on processors that do not support the 1-setting of that control. The four bitmaps are:

• Read bitmap for low MSRs (located at the MSR-bitmap address). This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.

• Read bitmap for high MSRs (located at the MSR-bitmap address plus 1024). This contains one bit for each MSR address in the range C0000000H to C0001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.

• Write bitmap for low MSRs (located at the MSR-bitmap address plus 2048). This contains one bit for each MSR address in the range 00000000H to 00001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

• Write bitmap for high MSRs (located at the MSR-bitmap address plus 3072). This contains one bit for each MSR address in the range C0000000H to C0001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

A logical processor uses these bitmaps if and only if the "use MSR bitmaps" control is 1. If the bitmaps are used, an execution of RDMSR or WRMSR causes a VM exit if the value of RCX is in neither of the ranges covered by the bitmaps or if the appropriate bit in the MSR bitmaps (corresponding to the instruction and the RCX value) is 1. See Section 26.1.3 for details. If the bitmaps are used, their address must be 4-KByte aligned.

25.6.10 Executive-VMCS Pointer

The executive-VMCS pointer is a 64-bit field used in the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). SMM VM exits save this field as described in Section 32.15.2. VM entries that return from SMM use this field as described in Section 32.15.4.
25.6.11 Extended-Page-Table Pointer (EPTP)

The extended-page-table pointer (EPTP) contains the address of the base of EPT PML4 table (see Section 29.3.2), as well as other EPT configuration information. The format of this field is shown in Table 25-9.

Table 25-9. Format of Extended-Page-Table Pointer

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>EPT paging-structure memory type (see Section 29.3.7): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved.¹</td>
</tr>
<tr>
<td>5:3</td>
<td>This value is 1 less than the EPT page-walk length (see Section 29.3.2)</td>
</tr>
<tr>
<td>6</td>
<td>Setting this control to 1 enables accessed and dirty flags for EPT (see Section 29.3.5)²</td>
</tr>
<tr>
<td>7</td>
<td>Setting this control to 1 enables enforcement of access rights for supervisor shadow-stack pages (see Section 29.3.3.2)³</td>
</tr>
<tr>
<td>11:8</td>
<td>Reserved</td>
</tr>
<tr>
<td>N–1:12</td>
<td>Bits N–1:12 of the physical address of the 4-KByte aligned EPT paging-structure (an EPT PML4 table with 4-level EPT and an EPT PML5 table with 5-level EPT)⁴</td>
</tr>
<tr>
<td>63:N</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

NOTES:
1. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine what EPT paging-structure memory types are supported.
2. Not all processors support accessed and dirty flags for EPT. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_-CAP (see Appendix A.10) to determine whether the processor supports this feature.
3. Not all processors enforce access rights for shadow-stack pages. Software should read the VMX capability MSR IA32_VMX-_EPT_VPID_CAP (see Appendix A.10) to determine whether the processor supports this feature.
4. N is the physical-address width supported by the logical processor. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

The EPTP exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.

25.6.12 Virtual-Processor Identifier (VPID)

The virtual-processor identifier (VPID) is a 16-bit field. It exists only on processors that support the 1-setting of the “enable VPID” VM-execution control. See Section 29.1 for details regarding the use of this field.

25.6.13 Controls for PAUSE-Loop Exiting

On processors that support the 1-setting of the “PAUSE-loop exiting” VM-execution control, the VM-execution control fields include the following 32-bit fields:

- **PLE_Gap.** Software can configure this field as an upper bound on the amount of time between two successive executions of PAUSE in a loop.
- **PLE_Window.** Software can configure this field as an upper bound on the amount of time a guest is allowed to execute in a PAUSE loop.

These fields measure time based on a counter that runs at the same rate as the timestamp counter (TSC). See Section 26.1.3 for more details regarding PAUSE-loop exiting.
25.6.14 VM-Function Controls

The VM-function controls constitute a 64-bit vector that governs use of the VMFUNC instruction in VMX non-root operation. This field is supported only on processors that support the 1-settings of both the “activate secondary controls” primary processor-based VM-execution control and the “enable VM functions” secondary processor-based VM-execution control.

Table 25-10 lists the VM-function controls. See Section 26.5.6 for more details of how these controls affect processor behavior in VMX non-root operation.

Table 25-10. Definitions of VM-Function Controls

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EPTP switching</td>
<td>The EPTP-switching VM function changes the EPT pointer to a value chosen from the EPTP list. See Section 26.5.6.3.</td>
</tr>
</tbody>
</table>

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_VMFUNC (see Appendix A.11) to determine which bits are reserved. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 27.2.1.1).

Processors that support the 1-setting of the “EPTP switching” VM-function control also support a 64-bit field called the EPTP-list address. This field contains the physical address of the 4-KByte EPTP list. The EPTP list comprises 512 8-Byte entries (each an EPTP value) and is used by the EPTP-switching VM function (see Section 26.5.6.3).

25.6.15 VMCS Shadowing Bitmap Addresses

On processors that support the 1-setting of the “VMCS shadowing” VM-execution control, the VM-execution control fields include the 64-bit physical addresses of the VMREAD bitmap and the VMWRITE bitmap. Each bitmap is 4 KBytes in size and thus contains 32 KBits. The addresses are the VMREAD-bitmap address and the VMWRITE-bitmap address.

If the “VMCS shadowing” VM-execution control is 1, executions of VMREAD and VMWRITE may consult these bitmaps (see Section 25.10 and Section 31.3).

25.6.16 ENCLS-Exiting Bitmap

The ENCLS-exiting bitmap is a 64-bit field. If the “enable ENCLS exiting” VM-execution control is 1, execution of ENCLS causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the bit is 0, the instruction executes normally. See Section 26.1.3 for more information.

25.6.17 ENCLV-Exiting Bitmap

The ENCLV-exiting bitmap is a 64-bit field. If the “enable ENCLV exiting” VM-execution control is 1, execution of ENCLV causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the bit is 0, the instruction executes normally. See Section 26.1.3 for more information.

25.6.18 PCONFIG-Exiting Bitmap

The PCONFIG-exiting bitmap is a 64-bit field. If the “enable PCONFIG” VM-execution control is 1, execution of PCONFIG causes a VM exit if the bit in this field corresponding to the value of EAX is 1. If the control is 0, any execution of PCONFIG causes a #UD. See Section 26.1.3 for more information.
25.6.19 Control Field for Page-Modification Logging

The **PML address** is a 64-bit field. It is the 4-KByte aligned address of the **page-modification log**. The page-modification log consists of 512 64-bit entries. It is used for the page-modification logging feature. Details of the page-modification logging are given in Section 29.3.6.

If the "enable PML" VM-execution control is 1, VM entry ensures that the PML address is 4-KByte aligned. The PML address exists only on processors that support the 1-setting of the "enable PML" VM-execution control.

25.6.20 Controls for Virtualization Exceptions

On processors that support the 1-setting of the "EPT-violation #VE" VM-execution control, the VM-execution control fields include the following:

- **Virtualization-exception information address** (64 bits). This field contains the physical address of the virtualization-exception information area. When a logical processor encounters a virtualization exception, it saves virtualization-exception information at the virtualization-exception information address; see Section 26.5.7.2.

- **EPTP index** (16 bits). When an EPT violation causes a virtualization exception, the processor writes the value of this field to the virtualization-exception information area. The EPTP-switching VM function updates this field (see Section 26.5.6.3).

25.6.21 XSS-Exiting Bitmap

On processors that support the 1-setting of the "enable XSAVES/XRSTORS" VM-execution control, the VM-execution control fields include a 64-bit **XSS-exiting bitmap**. If the "enable XSAVES/XRSTORS" VM-execution control is 1, executions of XSAVES and XRSTORS may consult this bitmap (see Section 26.1.3 and Section 26.3).

25.6.22 Sub-Page-Permission-Table Pointer (SPPTP)

If the sub-page write-permission feature of EPT is enabled, EPT write permissions may be determined at a 128-byte granularity (see Section 29.3.4). These permissions are determined using a hierarchy of sub-page-permission structures in memory.

The root of this hierarchy is referenced by a VM-execution control field called the **sub-page-permission-table pointer** (SPPTP). The SPPTP contains the address of the base of the root SPP table (see Section 29.3.4.2). The format of this field is shown in Table 25-9.

![Table 25-11. Format of Sub-Page-Permission-Table Pointer](image)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>N-1:12</td>
<td>Bits N-1:12 of the physical address of the 4-KByte aligned root SPP table</td>
</tr>
<tr>
<td>63:N1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**NOTES:**
1. N is the processor's physical-address width. Software can determine this width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

The SPPTP exists only on processors that support the 1-setting of the "sub-page write permissions for EPT" VM-execution control.
25.6.23 Fields Related to Hypervisor-Managed Linear-Address Translation

Two fields are used when the “enable HLAT” VM-execution control is 1, enabling HLAT paging:

- The hypervisor-managed linear-address translation pointer (HLAT pointer or HLATP) is used by HLAT paging to locate and access the first paging structure used for linear-address translation (see Section 4.5). The format of this field is shown in Table 25-12.

Table 25-12. Format of Hypervisor-Managed Linear-Address Translation Pointer

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the first HLAT paging structure during linear-address translation.</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the first HLAT paging structure during linear-address translation.</td>
</tr>
<tr>
<td>11:5</td>
<td>Reserved</td>
</tr>
<tr>
<td>N-1:12</td>
<td>Guest-physical address (4KB-aligned) of the first HLAT paging structure during linear-address translation.</td>
</tr>
<tr>
<td>63:N</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

NOTES:
1. N is the physical-address width supported by the logical processor. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- The HLAT prefix size. The value of this field determines which linear address are subject to HLAT paging. See Section 4.5.1.

These fields exist only on processors that support the 1-setting of the “enable HLAT” VM-execution control.

25.6.24 Fields Related to PASID Translation

Two 64-bit VM-execution control fields are used when the “PASID translation” VM-execution control is 1, enabling translation of PASIDs for executions of ENQCMD and ENQCMDS: the low PASID directory address and the high PASID directory address. These are the physical addresses of the low PASID directory and the high PASID directory, respectively. These fields exist only on processors that support the 1-setting of the “PASID translation” VM-execution control.

See Section 26.5.8 for information on the PASID-translation process for ENQCMD and ENQCMDS.

25.6.25 Instruction-Timeout Control

On processors that support the 1-setting of the “instruction timeout” VM-execution control, the VM-execution control fields include a 32-bit instruction-timeout control. The processor interprets the value of this field as an amount of time as measured in units of crystal clock cycles. If the “instruction timeout” VM-execution control is 1, a VM exit occurs if certain operations prevent the processor from reaching an instruction boundary within this amount of time.

1. CPUID.15H:ECX enumerates the nominal frequency of the core crystal clock in Hz.
25.6.26 Fields Controlling Virtualization of the IA32_SPEC_CTRL MSR

On processors that support the 1-setting of the “virtualize IA32_SPEC_CTRL” VM-execution control, the VM-execution control fields include the following 64-bit fields:

- **IA32_SPEC_CTRL mask.** Setting a bit in this field prevents guest software from modifying the corresponding bit in the IA32_SPEC_CTRL MSR.
- **IA32_SPEC_CTRL shadow.** This field contains the value that guest software expects to be in the IA32_SPEC_CTRL MSR.

Section 26.3 discusses how these fields are used in VMX non-root operation.

25.7 VM-EXIT CONTROL FIELDS

The VM-exit control fields govern the behavior of VM exits. They are discussed in Section 25.7.1 and Section 25.7.2.

25.7.1 VM-Exit Controls

The VM-exit controls constitute two vectors that govern the basic operation of VM exits. These are the **primary VM-exit controls** (32 bits) and the **secondary VM-exits controls** (64 bits).

Table 25-13 lists the primary VM-exit controls. See Chapter 28 for complete details of how these controls affect VM exits.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Save debug controls</td>
<td>This control determines whether DR7 and the IA32_DEBUGCTL MSR are saved on VM exit. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.</td>
</tr>
<tr>
<td>9</td>
<td>Host address-space size</td>
<td>On processors that support Intel 64 architecture, this control determines whether a logical processor is in 64-bit mode after the next VM exit. Its value is loaded into CS.L, IA32_EFER.LME, and IA32_EFER.LMA on every VM exit. This control must be 0 on processors that do not support Intel 64 architecture.</td>
</tr>
<tr>
<td>12</td>
<td>Load IA32_PERF_GLOBAL_CTRL</td>
<td>This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM exit.</td>
</tr>
<tr>
<td>15</td>
<td>Acknowledge interrupt on exit</td>
<td>This control affects VM exits due to external interrupts:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If such a VM exit occurs and this control is 1, the logical processor acknowledges the interrupt controller, acquiring the interrupt's vector. The vector is stored in the VM-exit interruption-information field, which is marked valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If such a VM exit occurs and this control is 0, the interrupt is not acknowledged and the VM-exit interruption-information field is marked invalid.</td>
</tr>
<tr>
<td>18</td>
<td>Save IA32_PAT</td>
<td>This control determines whether the IA32_PAT MSR is saved on VM exit.</td>
</tr>
<tr>
<td>19</td>
<td>Load IA32_PAT</td>
<td>This control determines whether the IA32_PAT MSR is loaded on VM exit.</td>
</tr>
<tr>
<td>20</td>
<td>Save IA32_EFER</td>
<td>This control determines whether the IA32_EFER MSR is saved on VM exit.</td>
</tr>
<tr>
<td>21</td>
<td>Load IA32_EFER</td>
<td>This control determines whether the IA32_EFER MSR is loaded on VM exit.</td>
</tr>
<tr>
<td>22</td>
<td>Save VMX-preemption timer value</td>
<td>This control determines whether the value of the VMX-preemption timer is saved on VM exit.</td>
</tr>
<tr>
<td>23</td>
<td>Clear IA32_BNDCFGS</td>
<td>This control determines whether the IA32_BNDCFGS MSR is cleared on VM exit.</td>
</tr>
<tr>
<td>24</td>
<td>Conceal VMX from PT</td>
<td>If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM exit or a VMCS packet on an SMM VM exit (see Chapter 33).</td>
</tr>
</tbody>
</table>
All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_EXIT_CTLS and IA32_VMX_TRUE_EXIT_CTLS (see Appendix A.4) to determine how it should set the reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 27.2.1.2).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 0–8, 10, 11, 13, 14, 16, and 17. The VMX capability MSR IA32_VMX_EXIT_CTLS always reports that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_EXIT_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

Bit 31 of the primary processor-based VM-exit controls determines whether the secondary VM-exit controls are used. If that bit is 0, VM entries and VM exits function as if all the secondary VM-exit controls were 0. Processors that support only the 0-setting of bit 31 of the primary VM-exit controls do not support the secondary VM-exit controls.

Table 25-14 lists the secondary VM-exit controls. See Chapter 28 for more details of how these controls affect VM exits.

### Table 25-13. Definitions of Primary VM-Exit Controls (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Clear IA32_RTIT_CTL</td>
<td>This control determines whether the IA32_RTIT_CTL MSR is cleared on VM exit.</td>
</tr>
<tr>
<td>26</td>
<td>Clear IA32_LBR_CTL</td>
<td>This control determines whether the IA32_LBR_CTL MSR is cleared on VM exit.</td>
</tr>
<tr>
<td>27</td>
<td>Clear UINV</td>
<td>This control determines whether UINV is cleared on VM exit.</td>
</tr>
<tr>
<td>28</td>
<td>Load CET state</td>
<td>This control determines whether CET-related MSRs and SSP are loaded on VM exit.</td>
</tr>
<tr>
<td>29</td>
<td>Load PKRS</td>
<td>This control determines whether the IA32_PKRS MSR is loaded on VM exit.</td>
</tr>
<tr>
<td>30</td>
<td>Save IA32_PERF_GLOBAL_CTL</td>
<td>This control determines whether the IA32_PERF_GLOBAL_CTL MSR is saved on VM exit.</td>
</tr>
<tr>
<td>31</td>
<td>Activate secondary controls</td>
<td>This control determines whether the secondary VM-exit controls are used. If this control is 0, the logical processor operates as if all the secondary VM-exit controls were also 0.</td>
</tr>
</tbody>
</table>

### Notes:

1. Since the Intel 64 architecture specifies that IA32_EFER.LMA is always set to the logical-AND of CR0.PG and IA32_EFER.LME, and since CR0.PG is always 1 in VMX root operation, IA32_EFER.LMA is always identical to IA32_EFER.LME in VMX root operation.

All other bits in this field are reserved to 0. Software should consult the VMX capability MSRs IA32_VMX_EXIT_CTLS and IA32_VMX_TRUE_EXIT_CTLS (see Appendix A.4) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 27.2.1.2).

Bit 31 of the primary processor-based VM-exit controls determines whether the secondary VM-exit controls are used. If that bit is 0, VM entries and VM exits function as if all the secondary VM-exit controls were 0. Processors that support only the 0-setting of bit 31 of the primary VM-exit controls do not support the secondary VM-exit controls.

Table 25-14 lists the secondary VM-exit controls. See Chapter 28 for more details of how these controls affect VM exits.

### Table 25-14. Definitions of Secondary VM-Exit Controls

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Prematurely busy shadow stack</td>
<td>If this control is 1, VM exits that cause a shadow stack to become prematurely busy (see Section 17.2.3, “Supervisor Shadow Stack Token,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) indicate this fact and save additional information into the VMCS.</td>
</tr>
</tbody>
</table>

All other bits in this field are reserved to 0. Software should consult the VMX capability MSR IA32_VMX_EXIT_CTLS2 (see Appendix A.4.2) to determine which bits may be set to 1. Failure to clear reserved bits causes subsequent VM entries to fail (see Section 27.2.1.2).

### 25.7.2 VM-Exit Controls for MSRs

A VMM may specify lists of MSRs to be stored and loaded on VM exits. The following VM-exit control fields determine how MSRs are stored on VM exits:

- **VM-exit MSR-store count** (32 bits). This field specifies the number of MSRs to be stored on VM exit. It is recommended that this count not exceed 512. If otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.
- **VM-exit MSR-store address** (64 bits). This field contains the physical address of the VM-exit MSR-store area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-store count. The format of each entry is given in Table 25-15. If the VM-exit MSR-store count is not zero, the address must be 16-byte aligned.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>MSR index</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved</td>
</tr>
<tr>
<td>127:64</td>
<td>MSR data</td>
</tr>
</tbody>
</table>

  Table 25-15. Format of an MSR Entry

See Section 28.4 for how this area is used on VM exits.

The following VM-exit control fields determine how MSRs are loaded on VM exits:

- **VM-exit MSR-load count** (32 bits). This field contains the number of MSRs to be loaded on VM exit. It is recommended that this count not exceed 512. Otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.\(^1\)

- **VM-exit MSR-load address** (64 bits). This field contains the physical address of the VM-exit MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-load count (see Table 25-15). If the VM-exit MSR-load count is not zero, the address must be 16-byte aligned.

See Section 28.6 for how this area is used on VM exits.

### 25.8 VM-ENTRY CONTROL FIELDS

The VM-entry control fields govern the behavior of VM entries. They are discussed in Sections 25.8.1 through 25.8.3.

#### 25.8.1 VM-Entry Controls

The **VM-entry controls** constitute a 32-bit vector that governs the basic operation of VM entries. Table 25-16 lists the controls supported. See Chapter 25 for how these controls affect VM entries.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Load debug controls</td>
<td>This control determines whether DR7 and the IA32_DEBUGCTL MSR are loaded on VM entry. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.</td>
</tr>
<tr>
<td>9</td>
<td>IA-32e mode guest</td>
<td>On processors that support Intel 64 architecture, this control determines whether the logical processor is in IA-32e mode after VM entry. Its value is loaded into IA32_EFER.LMA as part of VM entry.(^1) This control must be 0 on processors that do not support Intel 64 architecture.</td>
</tr>
<tr>
<td>10</td>
<td>Entry to SMM</td>
<td>This control determines whether the logical processor is in system-management mode (SMM) after VM entry. This control must be 0 for any VM entry from outside SMM.</td>
</tr>
<tr>
<td>11</td>
<td>Deactivate dual-monitor treatment</td>
<td>If set to 1, the default treatment of SMIs and SMM is in effect after the VM entry (see Section 32.15.7). This control must be 0 for any VM entry from outside SMM.</td>
</tr>
</tbody>
</table>

---

1. Future implementations may allow more MSRs to be stored reliably. Software should consult the VMX capability MSR IA32_VMX_MISC to determine the number supported (see Appendix A.6).

1. Future implementations may allow more MSRs to be loaded reliably. Software should consult the VMX capability MSR IA32_VMX_MISC to determine the number supported (see Appendix A.6).
VIRTUAL MACHINE CONTROL STRUCTURES

All other bits in this field are reserved, some to 0 and some to 1. Software should consult the VMX capability MSRs IA32_VMX_ENTRY_CTLS and IA32_VMX_TRUE_ENTRY_CTLS (see Appendix A.5) to determine how it should set the reserved bits. Failure to set reserved bits properly causes subsequent VM entries to fail (see Section 27.2.1.3).

The first processors to support the virtual-machine extensions supported only the 1-settings of bits 0–8 and 12. The VMX capability MSR IA32_VMX_ENTRY_CTLS always reports that these bits must be 1. Logical processors that support the 0-settings of any of these bits will support the VMX capability MSR IA32_VMX_TRUE_ENTRY_CTLS MSR, and software should consult this MSR to discover support for the 0-settings of these bits. Software that is not aware of the functionality of any one of these bits should set that bit to 1.

### 25.8.2 VM-Entry Controls for MSRs

A VMM may specify a list of MSRs to be loaded on VM entries. The following VM-entry control fields manage this functionality:

- **VM-entry MSR-load count** (32 bits). This field contains the number of MSRs to be loaded on VM entry. It is recommended that this count not exceed 512. Otherwise, unpredictable processor behavior (including a machine check) may result during VM entry.¹

- **VM-entry MSR-load address** (64 bits). This field contains the physical address of the VM-entry MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-entry MSR-load count. The format of entries is described in Table 25-15. If the VM-entry MSR-load count is not zero, the address must be 16-byte aligned.

See Section 27.4 for details of how this area is used on VM entries.

### 25.8.3 VM-Entry Controls for Event Injection

1. Future implementations may allow more MSRs to be loaded reliably. Software should consult the VMX capability MSR IA32_VMX_MISC to determine the number supported (see Appendix A.6).

---

**Table 25-16. Definitions of VM-Entry Controls (Contd.)**

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Load IA32_PERFGLOBAL_CTRL</td>
<td>This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM entry.</td>
</tr>
<tr>
<td>14</td>
<td>Load IA32_PAT</td>
<td>This control determines whether the IA32_PAT MSR is loaded on VM entry.</td>
</tr>
<tr>
<td>15</td>
<td>Load IA32_EFER</td>
<td>This control determines whether the IA32_EFER MSR is loaded on VM entry.</td>
</tr>
<tr>
<td>16</td>
<td>Load IA32_BNDCFGFS</td>
<td>This control determines whether the IA32_BNDCFGFS MSR is loaded on VM entry.</td>
</tr>
<tr>
<td>17</td>
<td>Conceal VMX from PT</td>
<td>If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM entry or a VMCS packet on a VM entry that returns from SMM (see Chapter 33).</td>
</tr>
<tr>
<td>18</td>
<td>Load IA32_RTIT_CTL</td>
<td>This control determines whether the IA32_RTIT_CTL MSR is loaded on VM entry.</td>
</tr>
<tr>
<td>19</td>
<td>Load UINV</td>
<td>This control determines whether UINV is loaded on VM entry.</td>
</tr>
<tr>
<td>20</td>
<td>Load CET state</td>
<td>This control determines whether CET-related MSRs and SSP are loaded on VM entry.</td>
</tr>
<tr>
<td>21</td>
<td>Load guest IA32_LBR_CTL</td>
<td>This control determines whether the IA32_LBR_CTL MSR is loaded on VM entry.</td>
</tr>
<tr>
<td>22</td>
<td>Load PKRS</td>
<td>This control determines whether the IA32_PKRS MSR is loaded on VM entry.</td>
</tr>
</tbody>
</table>

NOTES:
1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control. If it is read as 1, every VM exit stores the value of IA32_EFER.LMA into the “IA-32e mode guest” VM-entry control (see Section 28.2).
VM entry can be configured to conclude by delivering an event through the IDT (after all guest state and MSRs have been loaded). This process is called **event injection** and is controlled by the following three VM-entry control fields:

- **VM-entry interruption-information field** (32 bits). This field provides details about the event to be injected. Table 25-17 describes the field.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Vector of interrupt or exception</td>
</tr>
<tr>
<td>10:8</td>
<td>Interruption type:</td>
</tr>
<tr>
<td></td>
<td>0: External interrupt</td>
</tr>
<tr>
<td></td>
<td>1: Reserved</td>
</tr>
<tr>
<td></td>
<td>2: Non-maskable interrupt (NMI)</td>
</tr>
<tr>
<td></td>
<td>3: Hardware exception (e.g., #PF)</td>
</tr>
<tr>
<td></td>
<td>4: Software interrupt (INT n)</td>
</tr>
<tr>
<td></td>
<td>5: Privileged software exception (INT1)</td>
</tr>
<tr>
<td></td>
<td>6: Software exception (INT3 or INTO)</td>
</tr>
<tr>
<td></td>
<td>7: Other event</td>
</tr>
<tr>
<td>11</td>
<td>Deliver error code (0 = do not deliver; 1 = deliver)</td>
</tr>
<tr>
<td>30:12</td>
<td>Reserved</td>
</tr>
<tr>
<td>31</td>
<td>Valid</td>
</tr>
</tbody>
</table>

  — The **vector** (bits 7:0) determines which entry in the IDT is used or which other event is injected.
  
  — The **interruption type** (bits 10:8) determines details of how the injection is performed. In general, a VMM should use the type hardware exception for all exceptions other than the following:
    • breakpoint exceptions (#BP; a VMM should use the type software exception);
    • overflow exceptions (#OF a VMM should use the use type software exception); and
    • those debug exceptions (#DB) that are generated by INT1 (a VMM should use the use type privileged software exception).1
  
  The type **other event** is used for injection of events that are not delivered through the IDT.2

  — For exceptions, the **deliver-error-code bit** (bit 11) determines whether delivery pushes an error code on the guest stack.

  — VM entry injects an event if and only if the **valid bit** (bit 31) is 1. The valid bit in this field is cleared on every VM exit (see Section 28.2).

- **VM-entry exception error code** (32 bits). This field is used if and only if the valid bit (bit 31) and the deliver-error-code bit (bit 11) are both set in the VM-entry interruption-information field.

- **VM-entry instruction length** (32 bits). For injection of events whose type is software interrupt, software exception, or privileged software exception, this field is used to determine the value of RIP that is pushed on the stack.

See Section 27.6 for details regarding the mechanics of event injection, including the use of the interruption type and the VM-entry instruction length.

VM exits clear the valid bit (bit 31) in the VM-entry interruption-information field.

### 25.9 VM-EXIT INFORMATION FIELDS

The VMCS contains a section of fields that contain information about the most recent VM exit.

---

1. The type hardware exception should be used for all other debug exceptions.
2. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT n with values 1 or 3 for n.
On some processors, attempts to write to these fields with VMWRITE fail (see “VMWRITE—Write Field to Virtual-Machine Control Structure” in Chapter 31).

### 25.9.1 Basic VM-Exit Information

The following VM-exit information fields provide basic information about a VM exit:

- **Exit reason** (32 bits). This field encodes the reason for the VM exit and has the structure given in Table 25-18.

#### Table 25-18. Format of Exit Reason

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:0</td>
<td>Basic exit reason.</td>
</tr>
<tr>
<td>16</td>
<td>Always cleared to 0.</td>
</tr>
<tr>
<td>24:17</td>
<td>Not currently defined.</td>
</tr>
<tr>
<td>25</td>
<td>A VM exit saves this bit as 1 to indicate that the VM exit caused a shadow stack to become prematurely busy.</td>
</tr>
<tr>
<td>26</td>
<td>A VM exit saves this bit as 1 to indicate that the VM exit occurred after assertion of a bus lock while the &quot;VMM bus-lock detection&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>27</td>
<td>A VM exit saves this bit as 1 to indicate that the VM exit was incident to enclave mode.</td>
</tr>
<tr>
<td>28</td>
<td>Pending MTF VM exit.</td>
</tr>
<tr>
<td>29</td>
<td>VM exit from VMX root operation.</td>
</tr>
<tr>
<td>30</td>
<td>Not currently defined.</td>
</tr>
<tr>
<td>31</td>
<td>VM-entry failure (0 = true VM exit; 1 = VM-entry failure)</td>
</tr>
</tbody>
</table>

- Bits 15:0 provide basic information about the cause of the VM exit (if bit 31 is clear) or of the VM-entry failure (if bit 31 is set). Appendix C enumerates the basic exit reasons.
- Bit 16 is always cleared to 0.
- Bit 25 is set to 1 if the "prematurely busy shadow stack" VM-exit control is 1 and the VM exit caused a shadow stack to become prematurely busy (see Section 26.4.3). Otherwise, the bit is cleared.
- Bit 26 is set to 1 if the VM exit occurred after assertion of a bus lock while the "VMM bus-lock detection" VM-execution control was 1. Such VM exits include those that occur due to the 1-setting of that control as well as others that might occur during execution of an instruction that asserted a bus lock.
- Bit 27 is set to 1 if the VM exit occurred while the logical processor was in enclave mode.
- A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interrupt (bit 4 of the field is 1). See Section 28.2.1 for details.
- Bit 28 is set only by an SMM VM exit (see Section 32.15.2) that took priority over an MTF VM exit (see Section 26.5.2) that would have occurred had the SMM VM exit not occurred. See Section 32.15.2.3.
- Bit 29 is set if and only if the processor was in VMX root operation at the time the VM exit occurred. This can happen only for SMM VM exits. See Section 32.15.2.
- Because some VM-entry failures load processor state from the host-state area (see Section 27.8), software must be able to distinguish such cases from true VM exits. Bit 31 is used for that purpose.

- **Exit qualification** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field contains additional information about the cause of VM exits due to the following: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); task switches; INVEPT; INVLPQ; INVLPG; INVVPID; LGDT; LIDT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMXCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; XRSTORS; XSAVES; control-

1. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).
register accesses; MOV DR; I/O instructions; and MWAIT. The format of the field depends on the cause of the VM exit. See Section 28.2.1 for details.

- **Guest-linear address** (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is used in the following cases:
  - VM exits due to attempts to execute LMSW with a memory operand.
  - VM exits due to attempts to execute INS or OUTS.
  - VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions.
  - Certain VM exits due to EPT violations

  See Section 28.2.1 and Section 32.15.2.3 for details of when and how this field is used.

- **Guest-physical address** (64 bits). This field is used by VM exits due to EPT violations and EPT misconfigurations. See Section 28.2.1 for details of when and how this field is used.

### 25.9.2 Information for VM Exits Due to Vectored Events

Event-specific information is provided for VM exits due to the following vectored events: exceptions (including those generated by the instructions INT3, INTO, INT1, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs). This information is provided in the following fields:

- **VM-exit interruption information** (32 bits). This field receives basic information associated with the event causing the VM exit. Table 25-19 describes this field.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Vector of interrupt or exception</td>
</tr>
<tr>
<td>10:8</td>
<td>Interruption type:</td>
</tr>
<tr>
<td></td>
<td>0: External interrupt</td>
</tr>
<tr>
<td></td>
<td>1: Not used</td>
</tr>
<tr>
<td></td>
<td>2: Non-maskable interrupt (NMI)</td>
</tr>
<tr>
<td></td>
<td>3: Hardware exception</td>
</tr>
<tr>
<td></td>
<td>4: Not used</td>
</tr>
<tr>
<td></td>
<td>5: Privileged software exception</td>
</tr>
<tr>
<td></td>
<td>6: Software exception</td>
</tr>
<tr>
<td></td>
<td>7: Not used</td>
</tr>
<tr>
<td>11</td>
<td>Error code valid (0 = invalid; 1 = valid)</td>
</tr>
<tr>
<td>12</td>
<td>NMI unblocking due to IRET</td>
</tr>
<tr>
<td>30:13</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>31</td>
<td>Valid</td>
</tr>
</tbody>
</table>

- **VM-exit interruption error code** (32 bits). For VM exits caused by hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

  Section 28.2.2 provides details of how these fields are saved on VM exits.

### 25.9.3 Information for VM Exits That Occur During Event Delivery

Additional information is provided for VM exits that occur during event delivery in VMX non-root operation. This information is provided in the following fields:

1. This includes cases in which the event delivery was caused by event injection as part of VM entry; see Section 27.6.1.2.
VIRTUAL MACHINE CONTROL STRUCTURES

• **IDT-vectoring information** (32 bits). This field receives basic information associated with the event that was being delivered when the VM exit occurred. Table 25-20 describes this field.

Table 25-20. Format of the IDT-Vectoring Information Field

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Vector of interrupt or exception</td>
</tr>
<tr>
<td>10:8</td>
<td>Interruption type:</td>
</tr>
<tr>
<td></td>
<td>0: External interrupt</td>
</tr>
<tr>
<td></td>
<td>1: Not used</td>
</tr>
<tr>
<td></td>
<td>2: Non-maskable interrupt (NMI)</td>
</tr>
<tr>
<td></td>
<td>3: Hardware exception</td>
</tr>
<tr>
<td></td>
<td>4: Software interrupt</td>
</tr>
<tr>
<td></td>
<td>5: Privileged software exception</td>
</tr>
<tr>
<td></td>
<td>6: Software exception</td>
</tr>
<tr>
<td></td>
<td>7: Not used</td>
</tr>
<tr>
<td>11</td>
<td>Error code valid (0 = invalid; 1 = valid)</td>
</tr>
<tr>
<td>30:12</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>31</td>
<td>Valid</td>
</tr>
</tbody>
</table>

• **IDT-vectoring error code** (32 bits). For VM exits the occur during delivery of hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

See Section 28.2.4 provides details of how these fields are saved on VM exits.

25.9.4 Information for VM Exits Due to Instruction Execution

The following fields are used for VM exits caused by attempts to execute certain instructions in VMX non-root operation:

• **VM-exit instruction length** (32 bits). For VM exits resulting from instruction execution, this field receives the length in bytes of the instruction whose execution led to the VM exit.¹ See Section 28.2.5 for details of when and how this field is used.

• **VM-exit instruction information** (32 bits). This field is used for VM exits due to attempts to execute INS, INVEPT, INVPID, LIDT, LGDT, LLDT, LTR, OUTS, SIDT, SGDT, SLDT, STR, VMCLEAR, VMPTLRD, VMPTLRST, VMREAD, VMWRITE, or VMXON.² The format of the field depends on the cause of the VM exit. See Section 28.2.5 for details.

The following fields (64 bits each; 32 bits on processors that do not support Intel 64 architecture) are used only for VM exits due to SMIs that arrive immediately after retirement of I/O instructions. They provide information about that I/O instruction:

• **I/O RCX**. The value of RCX before the I/O instruction started.

• **I/O RSI**. The value of RSI before the I/O instruction started.

• **I/O RDI**. The value of RDI before the I/O instruction started.

• **I/O RIP**. The value of RIP before the I/O instruction started (the RIP that addressed the I/O instruction).

1. This field is also used for VM exits that occur during the delivery of a software interrupt or software exception.

2. Whether the processor provides this information on VM exits due to attempts to execute INS or OUTS can be determined by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).
25.9.5 VM-Instruction Error Field

The 32-bit VM-instruction error field does not provide information about the most recent VM exit. In fact, it is not modified on VM exits. Instead, it provides information about errors encountered by a non-faulting execution of one of the VMX instructions.

25.10 VMCS TYPES: ORDINARY AND SHADOW

Every VMCS is either an ordinary VMCS or a shadow VMCS. A VMCS’s type is determined by the shadow-VMCS indicator in the VMCS region (this is the value of bit 31 of the first 4 bytes of the VMCS region; see Table 25-1): 0 indicates an ordinary VMCS, while 1 indicates a shadow VMCS. Shadow VMCSs are supported only on processors that support the 1-setting of the "VMCS shadowing" VM-execution control (see Section 25.6.2).

A shadow VMCS differs from an ordinary VMCS in two ways:

- An ordinary VMCS can be used for VM entry but a shadow VMCS cannot. Attempts to perform VM entry when the current VMCS is a shadow VMCS fail (see Section 27.1).
- The VMREAD and VMWRITE instructions can be used in VMX non-root operation to access a shadow VMCS but not an ordinary VMCS. This fact results from the following:
  - If the “VMCS shadowing” VM-execution control is 0, execution of the VMREAD and VMWRITE instructions in VMX non-root operation always cause VM exits (see Section 26.1.3).
  - If the “VMCS shadowing” VM-execution control is 1, execution of the VMREAD and VMWRITE instructions in VMX non-root operation can access the VMCS referenced by the VMCS link pointer (see Section 31.3).
  - If the “VMCS shadowing” VM-execution control is 1, VM entry ensures that any VMCS referenced by the VMCS link pointer is a shadow VMCS (see Section 27.3.1.5).

In VMX root operation, both types of VMCSs can be accessed with the VMREAD and VMWRITE instructions.

Software should not modify the shadow-VMCS indicator in the VMCS region of a VMCS that is active. Doing so may cause the VMCS to become corrupted (see Section 25.11.1). Before modifying the shadow-VMCS indicator, software should execute VMCLEAR for the VMCS to ensure that it is not active.

25.11 SOFTWARE USE OF THE VMCS AND RELATED STRUCTURES

This section details guidelines that software should observe when using a VMCS and related structures. It also provides descriptions of consequences for failing to follow guidelines.

25.11.1 Software Use of Virtual-Machine Control Structures

To ensure proper processor behavior, software should observe certain guidelines when using an active VMCS.

No VMCS should ever be active on more than one logical processor. If a VMCS is to be “migrated” from one logical processor to another, the first logical processor should execute VMCLEAR for the VMCS (to make it inactive on that logical processor and to ensure that all VMCS data are in memory) before the other logical processor executes VMPTRLD for the VMCS (to make it active on the second logical processor). A VMCS that is made active on more than one logical processor may become corrupted (see below).

Software should not modify the shadow-VMCS indicator (see Table 25-1) in the VMCS region of a VMCS that is active. Doing so may cause the VMCS to become corrupted. Before modifying the shadow-VMCS indicator, software should execute VMCLEAR for the VMCS to ensure that it is not active.

Software should use the VMREAD and VMWRITE instructions to access the different fields in the current VMCS (see Section 25.11.2). Software should never access or modify the VMCS data of an active VMCS using ordinary

---

1. As noted in Section 25.1, execution of the VMPTRLD instruction makes a VMCS is active. In addition, VM entry makes active any shadow VMCS referenced by the VMCS link pointer in the current VMCS. If a shadow VMCS is made active by VM entry, it is necessary to execute VMCLEAR for that VMCS before allowing that VMCS to become active on another logical processor.
memory operations, in part because the format used to store the VMCS data is implementation-specific and not architecturally defined, and also because a logical processor may maintain some VMCS data of an active VMCS on the processor and not in the VMCS region. The following items detail some of the hazards of accessing VMCS data using ordinary memory operations:

- Any data read from a VMCS with an ordinary memory read does not reliably reflect the state of the VMCS. Results may vary from time to time or from logical processor to logical processor.
- Writing to a VMCS with an ordinary memory write is not guaranteed to have a deterministic effect on the VMCS. Doing so may cause the VMCS to become corrupted (see below).

(Software can avoid these hazards by removing any linear-address mappings to a VMCS region before executing a VMPTRLD for that region and by not remapping it until after executing VMCLEAR for that region.)

If a logical processor leaves VMX operation, any VMCSs active on that logical processor may be corrupted (see below). To prevent such corruption of a VMCS that may be used either after a return to VMX operation or on another logical processor, software should execute VMCLEAR for that VMCS before executing the VMXOFF instruction or removing power from the processor (e.g., as part of a transition to the S3 and S4 power states).

This section has identified operations that may cause a VMCS to become corrupted. These operations may cause the VMCS’s data to become undefined. Behavior may be unpredictable if that VMCS used subsequently on any logical processor. The following items detail some hazards of VMCS corruption:

- VM entries may fail for unexplained reasons or may load undesired processor state.
- The processor may not correctly support VMX non-root operation as documented in Chapter 26 and may generate unexpected VM exits.
- VM exits may load undesired processor state, save incorrect state into the VMCS, or cause the logical processor to transition to a shutdown state.

### 25.11.2 VMREAD, VMWRITE, and Encodings of VMCS Fields

Every field of the VMCS is associated with a 32-bit value that is its **encoding**. The encoding is provided in an operand to VMREAD and VMWRITE when software wishes to read or write that field. These instructions fail if given, in 64-bit mode, an operand that sets an encoding bit beyond bit 32. See Chapter 31 for a description of these instructions.

The structure of the 32-bit encodings of the VMCS components is determined principally by the width of the fields and their function in the VMCS. See Table 25-21.

#### Table 25-21. Structure of VMCS Component Encoding

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Access type (0 = full; 1 = high); must be full for 16-bit, 32-bit, and natural-width fields</td>
</tr>
<tr>
<td>9:1</td>
<td>Index</td>
</tr>
<tr>
<td>11:10</td>
<td>Type:</td>
</tr>
<tr>
<td></td>
<td>0: control</td>
</tr>
<tr>
<td></td>
<td>1: VM-exit information</td>
</tr>
<tr>
<td></td>
<td>2: guest state</td>
</tr>
<tr>
<td></td>
<td>3: host state</td>
</tr>
<tr>
<td>12</td>
<td>Reserved (must be 0)</td>
</tr>
<tr>
<td>14:13</td>
<td>Width:</td>
</tr>
<tr>
<td></td>
<td>0: 16-bit</td>
</tr>
<tr>
<td></td>
<td>1: 64-bit</td>
</tr>
<tr>
<td></td>
<td>2: 32-bit</td>
</tr>
<tr>
<td></td>
<td>3: natural-width</td>
</tr>
<tr>
<td>31:15</td>
<td>Reserved (must be 0)</td>
</tr>
</tbody>
</table>
The following items detail the meaning of the bits in each encoding:

- **Field width.** Bits 14:13 encode the width of the field.
  - A value of 0 indicates a 16-bit field.
  - A value of 1 indicates a 64-bit field.
  - A value of 2 indicates a 32-bit field.
  - A value of 3 indicates a natural-width field. Such fields have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not.

Fields whose encodings use value 1 are specially treated to allow 32-bit software access to all 64 bits of the field. Such access is allowed by defining, for each such field, an encoding that allows direct access to the high 32 bits of the field. See below.

- **Field type.** Bits 11:10 encode the type of VMCS field: control, guest-state, host-state, or VM-exit information. (The last category also includes the VM-instruction error field.)

- **Index.** Bits 9:1 distinguish components with the same field width and type.

- **Access type.** Bit 0 must be 0 for all fields except for 64-bit fields (those with field-width 1; see above). A VMREAD or VMWRITE using an encoding with this bit cleared to 0 accesses the entire field. For a 64-bit field with field-width 1, a VMREAD or VMWRITE using an encoding with this bit set to 1 accesses only the high 32 bits of the field.

Appendix B gives the encodings of all fields in the VMCS.

The following describes the operation of VMREAD and VMWRITE based on processor mode, VMCS-field width, and access type:

- **16-bit fields:**
  - A VMREAD returns the value of the field in bits 15:0 of the destination operand; other bits of the destination operand are cleared to 0.
  - A VMWRITE writes the value of bits 15:0 of the source operand into the VMCS field; other bits of the source operand are not used.

- **32-bit fields:**
  - A VMREAD returns the value of the field in bits 31:0 of the destination operand; in 64-bit mode, bits 63:32 of the destination operand are cleared to 0.
  - A VMWRITE writes the value of bits 31:0 of the source operand into the VMCS field; in 64-bit mode, bits 63:32 of the source operand are not used.

- **64-bit fields and natural-width fields using the full access type outside IA-32e mode.**
  - A VMREAD returns the value of bits 31:0 of the field in its destination operand; bits 63:32 of the field are ignored.
  - A VMWRITE writes the value of its source operand to bits 31:0 of the field and clears bits 63:32 of the field.

- **64-bit fields and natural-width fields using the full access type in 64-bit mode (only on processors that support Intel 64 architecture).**
  - A VMREAD returns the value of the field in bits 63:0 of the destination operand
  - A VMWRITE writes the value of bits 63:0 of the source operand into the VMCS field.

- **64-bit fields using the high access type.**
  - A VMREAD returns the value of bits 63:32 of the field in bits 31:0 of the destination operand; in 64-bit mode, bits 63:32 of the destination operand are cleared to 0.
  - A VMWRITE writes the value of bits 31:0 of the source operand to bits 63:32 of the field; in 64-bit mode, bits 63:32 of the source operand are not used.

Software seeking to read a 64-bit field outside IA-32e mode can use VMREAD with the full access type (reading bits 31:0 of the field) and VMREAD with the high access type (reading bits 63:32 of the field); the order of the two VMREAD executions is not important. Software seeking to modify a 64-bit field outside IA-32e mode should first
use VMWRITE with the full access type (establishing bits 31:0 of the field while clearing bits 63:32) and then use VMWRITE with the high access type (establishing bits 63:32 of the field).

### 25.11.3 Initializing a VMCS

Software should initialize fields in a VMCS (using VMWRITE) before using the VMCS for VM entry. Failure to do so may result in unpredictable behavior; for example, a VM entry may fail for unexplained reasons, or a successful transition (VM entry or VM exit) may load processor state with unexpected values.

It is not necessary to initialize fields that the logical processor will not use. (For example, it is not necessary to initialize the MSR-bitmap address if the “use MSR bitmaps” VM-execution control is 0.)

A processor maintains some VMCS information that cannot be modified with the VMWRITE instruction; this includes a VMCS’s launch state (see Section 25.1). Such information may be stored in the VMCS data portion of a VMCS region. Because the format of this information is implementation-specific, there is no way for software to know, when it first allocates a region of memory for use as a VMCS region, how the processor will determine this information from the contents of the memory region.

In addition to its other functions, the VMCLEAR instruction initializes any implementation-specific information in the VMCS region referenced by its operand. To avoid the uncertainties of implementation-specific behavior, software should execute VMCLEAR on a VMCS region before making the corresponding VMCS active with VMPTRLD for the first time. (Figure 25-1 illustrates how execution of VMCLEAR puts a VMCS into a well-defined state.)

The following software usage is consistent with these limitations:

- VMCLEAR should be executed for a VMCS before it is used for VM entry for the first time.
- VMLAUNCH should be used for the first VM entry using a VMCS after VMCLEAR has been executed for that VMCS.
- VMRESUME should be used for any subsequent VM entry using a VMCS (until the next execution of VMCLEAR for the VMCS).

It is expected that, in general, VMRESUME will have lower latency than VMLAUNCH. Since “migrating” a VMCS from one logical processor to another requires use of VMCLEAR (see Section 25.11.1), which sets the launch state of the VMCS to “clear”, such migration requires the next VM entry to be performed using VMLAUNCH. Software developers can avoid the performance cost of increased VM-entry latency by avoiding unnecessary migration of a VMCS from one logical processor to another.

### 25.11.4 Software Access to Related Structures

In addition to data in the VMCS region itself, VMX non-root operation can be controlled by data structures that are referenced by pointers in a VMCS (for example, the I/O bitmaps). While the pointers to these data structures are parts of the VMCS, the data structures themselves are not. They are not accessible using VMREAD and VMWRITE but by ordinary memory writes.

Software should ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 25.11.1). Exceptions are made for the following data structures (subject to detailed discussion in the sections indicated): EPT paging structures and the data structures used to locate SPP vectors (Section 29.4.3); the virtual-APIC page (Section 30.1); the posted interrupt descriptor (Section 30.6); and the virtualization-exception information area (Section 26.5.7.2).

### 25.11.5 VMXON Region

Before executing VMXON, software allocates a region of memory (called the VMXON region)\(^1\) that the logical processor uses to support VMX operation. The physical address of this region (the VMXON pointer) is provided in an operand to VMXON. The VMXON pointer is subject to the limitations that apply to VMCS pointers:

---

1. The amount of memory required for the VMXON region is the same as that required for a VMCS region. This size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).
• The VMXON pointer must be 4-KByte aligned (bits 11:0 must be zero).
• The VMXON pointer must not set any bits beyond the processor’s physical-address width.\textsuperscript{1,2}

Before executing VMXON, software should write the VMCS revision identifier (see Section 25.2) to the VMXON region. (Specifically, it should write the 31-bit VMCS revision identifier to bits 30:0 of the first 4 bytes of the VMXON region; bit 31 should be cleared to 0.) It need not initialize the VMXON region in any other way. Software should use a separate region for each logical processor and should not access or modify the VMXON region of a logical processor between execution of VMXON and VMXOFF on that logical processor. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 25.11.1).

\textsuperscript{1} Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

\textsuperscript{2} If IA32_VMX_BASIC[48] is read as 1, the VMXON pointer must not set any bits in the range 63:32; see Appendix A.1.
In a virtualized environment using VMX, the guest software stack typically runs on a logical processor in VMX non-root operation. This mode of operation is similar to that of ordinary processor operation outside of the virtualized environment. This chapter describes the differences between VMX non-root operation and ordinary processor operation with special attention to causes of VM exits (which bring a logical processor from VMX non-root operation to root operation). The differences between VMX non-root operation and ordinary processor operation are described in the following sections:

- Section 26.1, "Instructions That Cause VM Exits."
- Section 26.2, "Other Causes of VM Exits."
- Section 26.3, "Changes to Instruction Behavior in VMX Non-Root Operation."
- Section 26.4, "Other Changes in VMX Non-Root Operation."
- Section 26.5, "Features Specific to VMX Non-Root Operation."
- Section 26.6, "Unrestricted Guests."

Chapter 27, "VM Entries," describes the data control structures that govern VMX non-root operation. Chapter 27, "VM Entries," describes the operation of VM entries by which the processor transitions from VMX root operation to VMX non-root operation. Chapter 26, "VMX Non-Root Operation," describes the operation of VM exits by which the processor transitions from VMX non-root operation to VMX root operation.

Chapter 29, "VMX Support for Address Translation," describes two features that support address translation in VMX non-root operation. Chapter 30, "APIC Virtualization and Virtual Interrupts," describes features that support virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC) in VMX non-root operation.

### 26.1 Instructions That Cause VM Exits

Certain instructions may cause VM exits if executed in VMX non-root operation. Unless otherwise specified, such VM exits are "fault-like," meaning that the instruction causing the VM exit does not execute and no processor state is updated by the instruction. Section 28.1 details architectural state in the context of a VM exit.

Section 26.1.1 defines the prioritization between faults and VM exits for instructions subject to both. Section 26.1.2 identifies instructions that cause VM exits whenever they are executed in VMX non-root operation (and thus can never be executed in VMX non-root operation). Section 26.1.3 identifies instructions that cause VM exits depending on the settings of certain VM-execution control fields (see Section 25.6).

#### 26.1.1 Relative Priority of Faults and VM Exits

The following principles describe the ordering between existing faults and VM exits:

- Certain exceptions have priority over VM exits. These include invalid-opcode exceptions, faults based on privilege level, and general-protection exceptions that are based on checking I/O permission bits in the task-state segment (TSS). For example, execution of RDMSR with CPL = 3 generates a general-protection exception and not a VM exit.
- Faults incurred while fetching instruction operands have priority over VM exits that are conditioned based on the contents of those operands (see LMSW in Section 26.1.3).
- VM exits caused by execution of the INS and OUTS instructions (resulting either because the “unconditional I/O exiting” VM-execution control is 1 or because the “use I/O bitmaps control is 1”) have priority over the following faults:

1. These include faults generated by attempts to execute, in virtual-8086 mode, privileged instructions that are not recognized in that mode.
2. MOV DR is an exception to this rule; see Section 26.1.3.
VMX NON-ROOT OPERATION

- A general-protection fault due to the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) being unusable
- A general-protection fault due to an offset beyond the limit of the relevant segment
- An alignment-check exception

Fault-like VM exits have priority over exceptions other than those mentioned above. For example, RDMSR of a non-existent MSR with CPL = 0 generates a VM exit and not a general-protection exception.

When Section 26.1.2 or Section 26.1.3 (below) identify an instruction execution that may lead to a VM exit, it is assumed that the instruction does not incur a fault that takes priority over a VM exit.

### 26.1.2 Instructions That Cause VM Exits Unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, GETSEC,\(^1\) INVD, and XSETBV. This is also true of instructions introduced with VMX, which include: INVEPT, INVVPID, VMCALL,\(^2\) VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON.

### 26.1.3 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause “fault-like” VM exits based on the conditions described:\(^3\)

- **CLTS.** The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.

- **ENCLS.** The ENCLS instruction causes a VM exit if the “enable ENCLS exiting” VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLS-exiting bitmap is 1 (see Section 25.6.16).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLS-exiting bitmap is 1.

- **ENCLV.** The ENCLV instruction causes a VM exit if the “enable ENCLV exiting” VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLV-exiting bitmap is 1 (see Section 25.6.17).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLV-exiting bitmap is 1.

- **ENQCMD, ENQCMDs.** The behavior of each of these instructions is determined by the setting of the “PASID translation” VM-execution control. If that control is 0, the instruction executes normally. If the control is 1, instruction behavior is modified and may cause a VM exit. See Section 26.5.8.

- **HLT.** The HLT instruction causes a VM exit if the “HLT exiting” VM-execution control is 1.

- **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.** The behavior of each of these instructions is determined by the settings of the “unconditional I/O exiting” and “use I/O bitmaps” VM-execution controls:
  - If both controls are 0, the instruction executes normally.

---

1. An execution of GETSEC in VMX non-root operation causes a VM exit if CR4.SMXE[Bit 14] = 1 regardless of the value of CPL or RAX. An execution of GETSEC causes an invalid-opcode exception (#UD) if CR4.SMXE[Bit 14] = 0.
2. Under the dual-monitor treatment of SMIs and SMM, executions of VMCALL cause SMM VM exits in VMX root operation outside SMM. See Section 32.15.2.
3. Items in this section may refer to secondary processor-based VM-execution controls and tertiary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the secondary processor-based VM-execution controls were all 0; similarly, if bit 17 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the tertiary processor-based VM-execution controls were all 0. See Section 25.6.2.
— If the “unconditional I/O exiting” VM-execution control is 1 and the “use I/O bitmaps” VM-execution control is 0, the instruction causes a VM exit.

— If the “use I/O bitmaps” VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap (see Section 25.6.4). If an I/O operation “wraps around” the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the “unconditional I/O exiting” VM-execution control is ignored if the “use I/O bitmaps” VM-execution control is 1).

See Section 26.1.1 for information regarding the priority of VM exits relative to faults that may be caused by the INS and OUTS instructions.

• **INVLPG.** The INVLPG instruction causes a VM exit if the “INVLPG exiting” VM-execution control is 1.

• **INVPCID.** The INVPCID instruction causes a VM exit if the “INVLPG exiting” and “enable INVPCID” VM-execution controls are both 1.

• **LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR.** These instructions cause VM exits if the “descriptor-table exiting” VM-execution controls are both 1.

• **LMSW.** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. LMSW never clears bit 0 of CR0 (CR0.PE); thus, LMSW causes a VM exit if either of the following are true:
  — The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/host mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  — For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/host mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.

• **LOADIWKEY.** The LOADIWKEY instruction causes a VM exit if the “LOADIWKEY exiting” VM-execution control is 1.

• **MONITOR.** The MONITOR instruction causes a VM exit if the “MONITOR exiting” VM-execution control is 1.

• **MOV from CR3.** The MOV from CR3 instruction causes a VM exit if the “CR3-store exiting” VM-execution control is 1. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.

• **MOV from CR8.** The MOV from CR8 instruction causes a VM exit if the “CR8-store exiting” VM-execution control is 1.

• **MOV to CR0.** The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)

• **MOV to CR3.** The MOV to CR3 instruction causes a VM exit unless the “CR3-load exiting” VM-execution control is 0 or the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Only the first \( n \) CR3-target values are considered, where \( n \) is the CR3-target count. If the “CR3-load exiting” VM-execution control is 1 and the CR3-target count is 0, MOV to CR3 always causes a VM exit.

  The first processors to support the virtual-machine extensions supported only the 1-setting of the “CR3-load exiting” VM-execution control. These processors always consult the CR3-target controls to determine whether an execution of MOV to CR3 causes a VM exit.

• **MOV to CR4.** The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.

• **MOV to CR8.** The MOV to CR8 instruction causes a VM exit if the “CR8-load exiting” VM-execution control is 1.

• **MOV DR.** The MOV DR instruction causes a VM exit if the “MOV-DR exiting” VM-execution control is 1. Such VM exits represent an exception to the principles identified in Section 26.1.1 in that they take priority over the following: general-protection exceptions based on privilege level; and invalid-opcode exceptions that occur because CR4.DE=1 and the instruction specified access to DR4 or DR5.

• **MWAIT.** The MWAIT instruction causes a VM exit if the “MWAIT exiting” VM-execution control is 1. If this control is 0, the behavior of the MWAIT instruction may be modified (see Section 26.3).

• **PAUSE.** The behavior of each of this instruction depends on CPL and the settings of the “PAUSE exiting” and “PAUSE-loop exiting” VM-execution controls:
  — CPL = 0.
• If the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls are both 0, the PAUSE instruction executes normally.

• If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit (the "PAUSE-loop exiting" VM-execution control is ignored if CPL = 0 and the "PAUSE exiting" VM-execution control is 1).

• If the "PAUSE exiting" VM-execution control is 0 and the "PAUSE-loop exiting" VM-execution control is 1, the following treatment applies.

  The processor determines the amount of time between this execution of PAUSE and the previous execution of PAUSE at CPL 0. If this amount of time exceeds the value of the VM-execution control field PLE_Gap, the processor considers this execution to be the first execution of PAUSE in a loop. (It also does so for the first execution of PAUSE at CPL 0 after VM entry.)

  Otherwise, the processor determines the amount of time since the most recent execution of PAUSE that was considered to be the first in a loop. If this amount of time exceeds the value of the VM-execution control field PLE_Window, a VM exit occurs.

  For purposes of these computations, time is measured based on a counter that runs at the same rate as the timestamp counter (TSC).

  — CPL > 0.

  • If the "PAUSE exiting" VM-execution control is 0, the PAUSE instruction executes normally.
  
  • If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit.

  The "PAUSE-loop exiting" VM-execution control is ignored if CPL > 0.

• PCONFIG. The PCONFIG instruction causes a VM exit if the "enable PCONFIG" VM-execution control is 1 and one of the following is true:

  — The value of EAX is less than 63 and the corresponding bit in the PCONFIG-exiting bitmap is 1 (see Section 25.6.18).
  
  — The value of EAX is greater than or equal to 63 and bit 63 in the PCONFIG-exiting bitmap is 1.

  If the "enable PCONFIG" VM-execution control is 1 and neither of the previous items hold, the PCONFIG instruction executes normally.

• RDMSR. The RDMSR instruction causes a VM exit if any of the following are true:

  — The "use MSR bitmaps" VM-execution control is 0.
  
  — The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  
  — The value of ECX is in the range 00000000H – 00001FFFH and bit n in read bitmap for low MSRs is 1, where

  n is the value of ECX.

  — The value of ECX is in the range C0000000H – C0001FFFH and bit n in read bitmap for high MSRs is 1, where

  n is the value of ECX & 00001FFFH.

  See Section 25.6.9 for details regarding how these bitmaps are identified.

• RDPMC. The RDPMC instruction causes a VM exit if the "RDPMC exiting" VM-execution control is 1.

• RDRAND. The RDRAND instruction causes a VM exit if the "RDRAND exiting" VM-execution control is 1.

• RDSEED. The RDSEED instruction causes a VM exit if the "RDSEED exiting" VM-execution control is 1.

• RDTSC. The RDTSC instruction causes a VM exit if the "RDTSC exiting" VM-execution control is 1.

• RDTSCP. The RDTSCP instruction causes a VM exit if the "RDTSC exiting" and "enable RDTSCP" VM-execution controls are both 1.

• RSM. The RSM instruction causes a VM exit if executed in system-management mode (SMM).\(^ 1 \)

• TPAUSE. The TPAUSE instruction causes a VM exit if the "RDTSC exiting" and "enable user wait and pause" VM-execution controls are both 1.

---

1. Execution of the RSM instruction outside SMM causes an invalid-opcode exception regardless of whether the processor is in VMX operation. It also does so in VMX root operation in SMM; see Section 32.15.3.
• **UMWAIT.** The UMWAIT instruction causes a VM exit if the “RDTSC exiting” and “enable user wait and pause” VM-execution controls are both 1.

• **VMREAD.** The VMREAD instruction causes a VM exit if any of the following are true:
  — The “VMCS shadowing” VM-execution control is 0.
  — Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  — Bit $n$ in VMREAD bitmap is 1, where $n$ is the value of bits 14:0 of the register source operand. See Section 25.6.15 for details regarding how the VMREAD bitmap is identified.

If the VMREAD instruction does not cause a VM exit, it reads from the VMCS referenced by the VMCS link pointer. See Chapter 31, “VMREAD—Read Field from Virtual-Machine Control Structure” for details of the operation of the VMREAD instruction.

• **VMWRITE.** The VMWRITE instruction causes a VM exit if any of the following are true:
  — The “VMCS shadowing” VM-execution control is 0.
  — Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  — Bit $n$ in VMWRITE bitmap is 1, where $n$ is the value of bits 14:0 of the register source operand. See Section 25.6.15 for details regarding how the VMWRITE bitmap is identified.

If the VMWRITE instruction does not cause a VM exit, it writes to the VMCS referenced by the VMCS link pointer. See Chapter 31, “VMWRITE—Write Field to Virtual-Machine Control Structure” for details of the operation of the VMWRITE instruction.

• **WBINVD.** The WBINVD instruction causes a VM exit if the “WBINVD exiting” VM-execution control is 1.

• **WBNOINVD.** The WBNOINVD instruction causes a VM exit if the “WBINVD exiting” VM-execution control is 1.

• **WRMSR.** The WRMSR instruction causes a VM exit if any of the following are true:
  — The “use MSR bitmaps” VM-execution control is 0.
  — The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  — The value of ECX is in the range 00000000H – 00001FFFH and bit $n$ in write bitmap for low MSRs is 1, where $n$ is the value of ECX.
  — The value of ECX is in the range C0000000H – C0001FFFH and bit $n$ in write bitmap for high MSRs is 1, where $n$ is the value of ECX & 00001FFFH.

See Section 25.6.9 for details regarding how these bitmaps are identified.

• **XRSTORS.** The XRSTORS instruction causes a VM exit if the “enable XSAVES/XRSTORS” VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 25.6.21).

• **XSAVES.** The XSAVES instruction causes a VM exit if the “enable XSAVES/XRSTORS” VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 25.6.21).

### 26.2 OTHER CAUSES OF VM EXITS

In addition to VM exits caused by instruction execution, the following events can cause VM exits:

• **Exceptions.** Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 25.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT1, INT3, INTO, BOUND, UDO, UD1, and UD2.\(^1\)

---

\(^1\) Items in this section may refer to secondary processor-based VM-execution controls and tertiary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the secondary processor-based VM-execution controls were all 0; similarly, if bit 17 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the tertiary processor-based VM-execution controls were all 0. See Section 25.6.2.
Page faults (exceptions with vector 14) are specially treated. When a page fault occurs, a processor consults (1) bit 14 of the exception bitmap; (2) the error code produced with the page fault [PFEC]; (3) the page-fault error-code mask field [PFEC_MASK]; and (4) the page-fault error-code match field [PFEC_MATCH]. It checks if \( \text{PFEC} \& \text{PFEC\_MASK} = \text{PFEC\_MATCH} \). If there is equality, the specification of bit 14 in the exception bitmap is followed (for example, a VM exit occurs if that bit is set). If there is inequality, the meaning of that bit is reversed (for example, a VM exit occurs if that bit is clear).

Thus, if software desires VM exits on all page faults, it can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If software desires VM exits on no page faults, it can set bit 14 in the exception bitmap to 1, the page-fault error-code mask field to 00000000H, and the page-fault error-code match field to FFFFFFFFH.

- **Triple fault.** A VM exit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap. This applies to the case in which the double-fault exception was generated within VMX non-root operation, the case in which the double-fault exception was generated during event injection by VM entry, and to the case in which VM entry is injecting a double-fault exception.

- **External interrupts.** An external interrupt causes a VM exit if the "external-interrupt exiting" VM-execution control is 1. (See Section 26.6 for an exception.) Otherwise, the processor handles the interrupt is normally.\(^1\) (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The processor does handle the interrupt and no VM exit occurs.)

- **Non-maskable interrupts (NMIs).** An NMI causes a VM exit if the “NMI exiting” VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)

- **INIT signals.** INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)

- **Start-up IPIs (SIPIs).** SIPIs cause VM exits. If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)

- **System-management interrupts (SMIs).** If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits. See Section 32.15.2.\(^2\)

- **VMX-preemption timer.** A VM exit occurs when the timer counts down to zero. See Section 26.5.1 for details of operation of the VMX-preemption timer.

Debug-trap exceptions and higher priority events take priority over VM exits caused by the VMX-preemption timer. VM exits caused by the VMX-preemption timer take priority over VM exits caused by the "NMI-window exiting" VM-execution control and lower priority events.

These VM exits wake a logical processor from the same inactive states as would a non-maskable interrupt. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

- **Bus locks.** Assertion of a bus lock (see Section 9.1.2) causes a VM exit if the "VMM bus-lock detection” VM-execution control is 1. Such a VM exit is trap-like because it is generated after execution of an instruction that asserts a bus lock. The VM exit thus does not prevent assertion of the bus lock. These VM exits take priority over system-management interrupts (SMIs), INIT signals, and lower priority events.

---

2. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT \( n \) with value 1 or 3 for \( n \).
1. Normal handling usually means delivery through the IDT, but it could also mean treatment of the interrupt as a user-interrupt notification.
2. Under the dual-monitor treatment of SMIs and SMM, SMIs also cause SMM VM exits if they occur in VMX root operation outside SMM. If the processor is using the default treatment of SMIs and SMM, SMIs are delivered as described in Section 32.14.1.
• **Instruction timeout.** If the “instruction timeout” VM-execution control is 1, a VM exit occurs if certain operations prevent the processor from reaching an instruction boundary within the amount of time specified by the instruction-timeout control VM-execution control field (see Section 25.6.25).

In addition, there are controls that cause VM exits based on the readiness of guest software to receive interrupts:

• If the “interrupt-window exiting” VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 1 and there is no blocking of events by STI or by MOV SS (see Table 25-3). Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 27.7.5).

Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.

• If the “NMI-window exiting” VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS and no blocking of events by STI (see Table 25-3). Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 27.7.6).

VM exits caused by the VMX-preemption timer and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an NMI. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

### 26.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:

• **CLTS.** Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  
  — If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 24.8), in which case CLTS causes a general-protection exception.

  — If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.

  — If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit.

• **ENQCMD, ENQCMDS.** Each of these instructions performs a 64-byte enqueue store that includes a PASID value in bits 19:0. For ENQCMD, the PASID is normally the value of IA32_PASID[19:0], while for ENQCMDS, the PASID is normally read from memory.

  The behavior of each of these instructions (and in particular the PASID value used for the enqueue store) is determined by the setting of the “PASID translation” VM-execution control:

  — If the “PASID translation” VM-execution control is 0, the instruction operates normally.

  — If the “PASID translation” VM-execution control is 1, the PASID value used for the enqueue store is determined by the PASID-translation process described in Section 26.5.8. (Note the PASID translation may result in a VM exit, in which case the enqueue store is not performed.)

---

1. Items in this section may refer to secondary processor-based VM-execution controls and tertiary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the secondary processor-based VM-execution controls were all 0; similarly, if bit 17 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the tertiary processor-based VM-execution controls were all 0. See Section 25.6.2.
VMX NON-ROOT OPERATION

An execution of ENQCMD or ENQCMDS performs PASID translation only after checking for conditions that may result in general-protection exception (the check of IA32_PASID.Valid for ENQCMD; the privilege-level check for ENQCMDS), after loading the instruction’s source operand from memory, and thus after any faults or VM exits that the loading may cause (e.g., page faults or EPT violations). PASID translation occurs before the actual enqueue store and thus before any faults or VM exits that it may cause.

• **INVPCID.** Behavior of the INVPCID instruction is determined first by the setting of the “enable INVPCID” VM-execution control:
  — If the “enable INVPCID” VM-execution control is 0, INVPCID causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
  — If the “enable INVPCID” VM-execution control is 1, treatment is based on the setting of the “INVLPG exiting” VM-execution control:
    • If the “INVLPG exiting” VM-execution control is 0, INVPCID operates normally.
    • If the “INVLPG exiting” VM-execution control is 1, INVPCID causes a VM exit.

• **IRET.** Behavior of IRET with regard to NMI blocking (see Table 25-3) is determined by the settings of the “NMI exiting” and “virtual NMIs” VM-execution controls:
  — If the “NMI exiting” VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” control must be 0; see Section 27.2.1.1.)
  — If the “NMI exiting” VM-execution control is 1, IRET does not affect blocking of NMIs. If, in addition, the “virtual NMIs” VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking.

The unblocking of NMIs or virtual NMIs specified above occurs even if IRET causes a fault.

• **LMSW.** Outside of VMX non-root operation, LMSW loads its source operand into CR0[3:0], but it does not clear CR0.PE if that bit is set. In VMX non-root operation, an execution of LMSW that does not cause a VM exit (see Section 26.1.3) leaves unmodified any bit in CR0[3:0] corresponding to a bit set in the CR0 guest/host mask. An attempt to set any other bit in CR0[3:0] to a value not supported in VMX operation (see Section 24.8) causes a general-protection exception. Attempts to clear CR0.PE are ignored without fault.

• **MOV from CR0.** The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 reads the value of the CR0 read shadow.

Depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

• **MOV from CR3.** If the “enable EPT” VM-execution control is 1 and an execution of MOV from CR3 does not cause a VM exit (see Section 26.1.3), the value loaded from CR3 is a guest-physical address; see Section 29.3.1.

• **MOV from CR4.** The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 reads the value of the CR4 read shadow.

Depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.

• **MOV from CR8.** If the MOV from CR8 instruction does not cause a VM exit (see Section 26.1.3), its behavior is modified if the “use TPR shadow” VM-execution control is 1; see Section 30.3.

• **MOV to CR0.** An execution of MOV to CR0 that does not cause a VM exit (see Section 26.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. Treatment of attempts to modify other bits in CR0 depends on the setting of the “unrestricted guest” VM-execution control:
  — If the control is 0, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 to a value not supported in VMX operation (see Section 24.8).
— If the control is 1, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 other than bit 0 (PE) or bit 31 (PG) to a value not supported in VMX operation. It remains the case, however, that MOV to CR0 causes a general-protection exception if it would result in CR0.PE = 0 and CR0.PG = 1 or if it would result in CR0.PG = 1, CR4.PAE = 0, and IA32_EFER.LME = 1.

**MOV to CR3.** If the “enable EPT” VM-execution control is 1 and an execution of MOV to CR3 does not cause a VM exit (see Section 26.1.3), the value loaded into CR3 is treated as a guest-physical address; see Section 29.3.1.

— If PAE paging is not being used, the instruction does not use the guest-physical address to access memory and it does not cause it to be translated through EPT.1

— If PAE paging is being used, the instruction translates the guest-physical address through EPT and uses the result to load the four (4) page-directory-pointer-table entries (PDPTEs). The instruction does not use the guest-physical addresses the PDPTEs to access memory and it does not cause them to be translated through EPT.

**MOV to CR4.** An execution of MOV to CR4 that does not cause a VM exit (see Section 26.1.3) leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit in CR4 (not corresponding to a bit set in the CR4 guest/host mask) to a value not supported in VMX operation (see Section 24.8).

**MOV to CR8.** If the MOV to CR8 instruction does not cause a VM exit (see Section 26.1.3), its behavior is modified if the “use TPR shadow” VM-execution control is 1; see Section 30.3.

**MWAIT.** Behavior of the MWAIT instruction (which always causes an invalid-opcode exception—#UD—if CPL > 0) is determined by the setting of the “MWAIT exiting” VM-execution control:

— If the “MWAIT exiting” VM-execution control is 1, MWAIT causes a VM exit.

— If the “MWAIT exiting” VM-execution control is 0, MWAIT operates normally if one of the following are true: (1) ECX[0] is 0; (2) RFLAGS.IF = 1; or both of the following are true: (a) the “interrupt-window exiting” VM-execution control is 0; and (b) the logical processor has not recognized a pending virtual interrupt (see Section 29.2.1).

— If the “MWAIT exiting” VM-execution control is 0, ECX[0] = 1, and RFLAGS.IF = 0, MWAIT does not cause the processor to enter an implementation-dependent optimized state if either the “interrupt-window exiting” VM-execution control is 1 or the logical processor has recognized a pending virtual interrupt; instead, control passes to the instruction following the MWAIT instruction.

**PCONFIG.** Behavior of the PCONFIG instruction is determined by the setting of the “enable PCONFIG” VM-execution control:

— If the “enable PCONFIG” VM-execution control is 0, PCONFIG causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.

— If the “enable PCONFIG” VM-execution control is 1, PCONFIG may cause a VM exit as specified in Section 26.1.3; if it does not cause such a VM exit, it operates normally.

**RDMSR.** Section 26.1.3 identifies when executions of the RDMSR instruction cause VM exits. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction’s behavior may be modified for certain values of ECX:

— If ECX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the instruction is determined by the setting of the “use TSC offsetting” VM-execution control:

  • If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.

  • If the control is 1, the value returned is determined by the setting of the “use TSC scaling” VM-execution control:

    — If the control is 0, RDMSR loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.

---

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
VMX NON-ROOT OPERATION

— If the control is 1, RDMSR first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

The 1-setting of the "use TSC-offsetting" VM-execution control does not affect executions of RDMSR if ECX contains 6E0H (indicating the IA32_TSC_DEADLINE MSR). Such executions return the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.

— If ECX contains 48H (indicating the IA32_SPEC_CTRL MSR), the value returned by the instruction is determined by the setting of the "virtualize IA32_SPEC_CTRL" VM-execution control:

• If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_SPEC_CTRL MSR.
• If the control is 1, the value returned is that of the IA32_SPEC_CTRL shadow field in the VMCS.

— If ECX is in the range 800H–8FFH (indicating an APIC MSR), instruction behavior may be modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 30.5.

• RDPID. Behavior of the RDPID instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:

— If the "enable RDTSCP" VM-execution control is 0, RDPID causes an invalid-opcode exception (#UD).
— If the "enable RDTSCP" VM-execution control is 1, RDPID operates normally.

• RDTSC. Behavior of the RDTSC instruction is determined by the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:

— If both controls are 0, RDTSC operates normally.
— If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

• If the control is 0, RDTSC loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
• If the control is 1, RDTSC first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

— If the "RDTSC exiting" VM-execution control is 1, RDTSC causes a VM exit.

• RDTSCP. Behavior of the RDTSCP instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:

— If the "enable RDTSCP" VM-execution control is 0, RDTSCP causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.
— If the "enable RDTSCP" VM-execution control is 1, treatment is based on the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:

• If both controls are 0, RDTSCP operates normally.
• If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

— If the control is 0, RDTSCP loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
— If the control is 1, RDTSCP first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

In either case, RDTSCP also loads ECX with the value of bits 31:0 of the IA32_TSC_AUX MSR.

• SMSW. The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the
value of the corresponding bit in CR0. For each position corresponding to a bit in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, SMSW reads normally from CR0; if every bit is set in the CR0 guest/host mask, SMSW returns the value of the CR0 read shadow.

Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

• **TPAUSE.** Behavior of the TPAUSE instruction is determined first by the setting of the "enable user wait and pause" VM-execution control:
  
  — If the "enable user wait and pause" VM-execution control is 0, TPAUSE causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  
  — If the "enable user wait and pause" VM-execution control is 1, treatment is based on the setting of the "RDTSC exiting" VM-execution control:
    
    • If the "RDTSC exiting" VM-execution control is 0, the instruction delays for an amount of time called here the **physical delay.** The physical delay is first computed by determining the virtual delay (the time to delay relative to the guest's timestamp counter).
      
      If IA32_UMWAIT_CONTROL[31:2] is zero, the virtual delay is the value in EDX:EAX minus the value that RDTSC would return (see above); if IA32_UMWAIT_CONTROL[31:2] is not zero, the virtual delay is the minimum of that difference and AND(IA32_UMWAIT_CONTROL,FFFFFFFCH).
      
      The physical delay depends upon the settings of the "use TSC offsetting" and "use TSC scaling" VM-execution controls:
      
      — If either control is 0, the physical delay is the virtual delay.
      
      — If both controls are 1, the virtual delay is multiplied by $2^{48}$ (using a shift) to produce a 128-bit integer. That product is then divided by the TSC multiplier to produce a 64-bit integer. The physical delay is that quotient.
      
    • If the "RDTSC exiting" VM-execution control is 1, TPAUSE causes a VM exit.

• **UMONITOR.** Behavior of the UMONITOR instruction is determined by the setting of the "enable user wait and pause" VM-execution control:
  
  — If the "enable user wait and pause" VM-execution control is 0, UMONITOR causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  
  — If the "enable user wait and pause" VM-execution control is 1, UMONITOR operates normally.

• **UMWAIT.** Behavior of the UMWAIT instruction is determined first by the setting of the "enable user wait and pause" VM-execution control:
  
  — If the "enable user wait and pause" VM-execution control is 0, UMWAIT causes an invalid-opcode exception (#UD). This exception takes priority over any exception the instruction may incur.
  
  — If the "enable user wait and pause" VM-execution control is 1, treatment is based on the setting of the "RDTSC exiting" VM-execution control:
    
    • If the "RDTSC exiting" VM-execution control is 0, and if the instruction causes a delay, the amount of time delayed is called here the **physical delay.** The physical delay is first computed by determining the virtual delay (the time to delay relative to the guest's timestamp counter).
      
      If IA32_UMWAIT_CONTROL[31:2] is zero, the virtual delay is the value in EDX:EAX minus the value that RDTSC would return (see above); if IA32_UMWAIT_CONTROL[31:2] is not zero, the virtual delay is the minimum of that difference and AND(IA32_UMWAIT_CONTROL,FFFFFFFCH).
      
      The physical delay depends upon the settings of the "use TSC offsetting" and "use TSC scaling" VM-execution controls:
      
      — If either control is 0, the physical delay is the virtual delay.
— If both controls are 1, the virtual delay is multiplied by $2^{48}$ (using a shift) to produce a 128-bit integer. That product is then divided by the TSC multiplier to produce a 64-bit integer. The physical delay is that quotient.

• If the "RDTSCE exiting" VM-execution control is 1, UMWAIT causes a VM exit.

• **WRMSR.** Section 26.1.3 identifies when executions of the WRMSR instruction cause VM exits. If such an execution neither a fault due to CPL > 0 nor a VM exit, the instruction’s behavior may be modified for certain values of ECX:

  — If ECX contains 48H (indicating the IA32_SPEC_CTRL MSR), instruction behavior depends on the setting of the "virtualize IA32_SPEC_CTRL" VM-execution control:
  
  • If the control is 0, WRMSR operates normally, loading the IA32_SPEC_CTRL MSR with the value in EAX:EDX.
  
  • If the control is 1, instruction will attempt to write the IA32_SPEC_CTRL MSR using the instruction’s source operand, but it will attempt to modify only those bits in positions corresponding to bits cleared in the IA32_SPEC_CTRL mask field in the VMCS.

  Specifically, the instruction attempts to write the MSR with the following value:

  \[
  (MSR_{VAL} \& ISC\_MASK) \text{ OR } (SRC \& NOT ISC\_MASK),
  \]

  where MSR_{VAL} is the original value of the MSR, ISC\_MASK is the IA32_SPEC_CTRL mask, and SRC is the instruction’s source operand.

  Any fault that would result from writing that value to the MSR (e.g., due to a reserved-bit violation) occurs normally. Otherwise, the value is written to the MSR.

  Such a write to the MSR will have any side effects that would occur normally had the MSR been written with the value indicated above (including any side effects that may result from writing unchanged values to the masked bits).

  If the write completes without a fault, the unmodified value of the source operand is written to the IA32_SPEC_CTRL shadow field in the VMCS.

  — If ECX contains 79H (indicating IA32_BIOS_UPDT TRIG MSR), no microcode update is loaded, and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.

  — On processors that support Intel PT but which do not allow it to be used in VMX operation, if ECX contains 570H (indicating the IA32_RTIT_CTL MSR), the instruction causes a general-protection exception.\(^1\)

  — If ECX contains 808H (indicating the TPR MSR), 80BH (the EOI MSR), 830H (the ICR MSR), or 83FH (the self-IPI MSR), instruction behavior may be modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 30.5.

• **XRSTORS.** Behavior of the XRSTORS instruction is determined first by the setting of the "enable XSAVE/XRSTORS" VM-execution control:

  — If the "enable XSAVE/XRSTORS" VM-execution control is 0, XRSTORS causes an invalid-opcode exception (#UD).

  — If the "enable XSAVE/XRSTORS" VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 25.6.21):

    • XRSTORS causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.

    • Otherwise, XRSTORS operates normally.

• **XSAVE.** Behavior of the XSAVE instruction is determined first by the setting of the "enable XSAVE/XRSTORS" VM-execution control:

  — If the "enable XSAVE/XRSTORS" VM-execution control is 0, XSAVE causes an invalid-opcode exception (#UD).

---

1. Software should read the VMX capability MSR IA32_VMX_MISC to determine whether the processor allows Intel PT to be used in VMX operation (see Appendix A.6).
— If the “enable XSAVE/XRSTORS” VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 25.6.21):
  • XSAVE causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
  • Otherwise, XSAVE operates normally.

### 26.4 OTHER CHANGES IN VMX NON-ROOT OPERATION

Treatments of event blocking, task switches, and certain shadow-stack updates may differ in VMX non-root operation as described in the following sections.

#### 26.4.1 Event Blocking

Event blocking is modified in VMX non-root operation as follows:

- If the “external-interrupt exiting” VM-execution control is 1, RFLAGS.IF does not control the blocking of external interrupts. In this case, an external interrupt that is not blocked for other reasons causes a VM exit (even if RFLAGS.IF = 0).
- If the “external-interrupt exiting” VM-execution control is 1, external interrupts may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).
- If the “NMI exiting” VM-execution control is 1, non-maskable interrupts (NMIs) may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).

#### 26.4.2 Treatment of Task Switches

Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. However, the following checks are performed (in the order indicated), possibly resulting in a fault, before there is any possibility of a VM exit due to task switch:

1. If a task gate is being used, appropriate checks are made on its P bit and on the proper values of the relevant privilege fields. The following cases detail the privilege checks performed:
   a. If CALL, INT n, INT1, INT3, INTO, or JMP accesses a task gate in IA-32e mode, a general-protection exception occurs.
   b. If CALL, INT n, INT3, INTO, or JMP accesses a task gate outside IA-32e mode, privilege-levels checks are performed on the task gate but, if they pass, privilege levels are not checked on the referenced task-state segment (TSS) descriptor.
   c. If CALL or JMP accesses a TSS descriptor directly in IA-32e mode, a general-protection exception occurs.
   d. If CALL or JMP accesses a TSS descriptor directly outside IA-32e mode, privilege levels are checked on the TSS descriptor.
   e. If a non-maskable interrupt (NMI), an exception, or an external interrupt accesses a task gate in the IDT in IA-32e mode, a general-protection exception occurs.
   f. If a non-maskable interrupt (NMI), an exception other than breakpoint exceptions (#BP) and overflow exceptions (#OF), or an external interrupt accesses a task gate in the IDT outside IA-32e mode, no privilege checks are performed.
   g. If IRET is executed with RFLAGS.NT = 1 in IA-32e mode, a general-protection exception occurs.
   h. If IRET is executed with RFLAGS.NT = 1 outside IA-32e mode, a TSS descriptor is accessed directly and no privilege checks are made.

2. Checks are made on the new TSS selector (for example, that is within GDT limits).
3. The new TSS descriptor is read. (A page fault results if a relevant GDT page is not present).
4. The TSS descriptor is checked for proper values of type (depends on type of task switch), P bit, S bit, and limit.
Only if checks 1–4 all pass (do not generate faults) might a VM exit occur. However, the ordering between a VM exit due to a task switch and a page fault resulting from accessing the old TSS or the new TSS is implementation-specific. Some processors may generate a page fault (instead of a VM exit due to a task switch) if accessing either TSS would cause a page fault. Other processors may generate a VM exit due to a task switch even if accessing either TSS would cause a page fault.

If an attempt at a task switch through a task gate in the IDT causes an exception (before generating a VM exit due to the task switch) and that exception causes a VM exit, information about the event whose delivery that accessed the task gate is recorded in the IDT-vectoring information fields and information about the exception that caused the VM exit is recorded in the VM-exit interruption-information fields. See Section 28.2. The fact that a task gate was being accessed is not recorded in the VMCS.

If an attempt at a task switch through a task gate in the IDT causes VM exit due to the task switch, information about the event whose delivery accessed the task gate is recorded in the IDT-vectoring fields of the VMCS. Since the cause of such a VM exit is a task switch and not an interruption, the valid bit for the VM-exit interruption information field is 0. See Section 28.2.

26.4.3 Shadow-Stack Updates

As noted in Section 17.2.3, “Supervisor Shadow Stack Token,” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, a switch of shadow stack may occur as part of IDT event delivery or an execution of far CALL that changes the CPL, or as part of IDT event delivery that uses the interrupt stack table (IST).

As part of the shadow-stack switch, the processor gains exclusive access to the new stack through manipulation of the supervisor shadow stack token located at the base of the new shadow stack. The processor reads the token and, among other things, confirms that bit 0 of the token (its busy bit) is 0. If the busy bit is already 1, the transition (event delivery or far CALL) cause a general-protection fault and does not complete. If the busy bit is 0, the transition sets the busy bit by writing to the token in memory. (The update is atomic with the original read of the token.)

If the transition commenced with CPL < 3, it will follow the token update by pushing three items on the new shadow stack (for the old values of the CS selector, instruction pointer, and shadow-stack pointer). As noted in Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, if any of the pushes causes a VM exit, the processor will revert to the old shadow stack and the busy bit in the new shadow stack's token remains set. The new shadow stack is said to be prematurely busy.

If the “prematurely busy shadow stack” VM-exit control is 1, a VM exit that results in a shadow stack becoming prematurely busy will indicate that fact through information saved in the VMCS. See Section 28.2.1.

26.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION

Some VM-execution controls support features that are specific to VMX non-root operation. These are the VMX-preemption timer (Section 26.5.1) and the monitor trap flag (Section 26.5.2), translation of guest-physical addresses (Section 26.5.3 and Section 26.5.4), APIC virtualization (Section 26.5.5), VM functions (Section 26.5.6), and virtualization exceptions (Section 26.5.7).

26.5.1 VMX-Preemption Timer

If the last VM entry was performed with the 1-setting of “activate VMX-preemption timer” VM-execution control, the VMX-preemption timer counts down (from the value loaded by VM entry; see Section 27.7.4) in VMX non-root operation. When the timer counts down to zero, it stops counting down and a VM exit occurs (see Section 26.2).

The VMX-preemption timer counts down at rate proportional to that of the timestamp counter (TSC). Specifically, the timer counts down by 1 every time bit X in the TSC changes due to a TSC increment. The value of X is in the range 0–31 and can be determined by consulting the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

The VMX-preemption timer operates in the C-states C0, C1, and C2; it also operates in the shutdown and wait-for-SIPI states. If the timer counts down to zero in any state other than the wait-for SIPI state, the logical processor...
transitions to the C0 C-state and causes a VM exit; the timer does not cause a VM exit if it counts down to zero in the wait-for-SIPI state. The timer is not decremented in C-states deeper than C2.

Treatment of the timer in the case of system management interrupts (SMIs) and system-management mode (SMM) depends on whether the treatment of SMIs and SMM:

- If the default treatment of SMIs and SMM (see Section 32.14) is active, the VMX-preemption timer counts across an SMI to VMX non-root operation, subsequent execution in SMM, and the return from SMM via the RSM instruction. However, the timer can cause a VM exit only from VMX non-root operation. If the timer expires during SMI, in SMM, or during RSM, a timer-induced VM exit occurs immediately after RSM with its normal priority unless it is blocked based on activity state (Section 26.2).
- If the dual-monitor treatment of SMIs and SMM (see Section 32.15) is active, transitions into and out of SMM are VM exits and VM entries, respectively. The treatment of the VMX-preemption timer by those transitions is mostly the same as for ordinary VM exits and VM entries; Section 32.15.2 and Section 32.15.4 detail some differences.

### 26.5.2 Monitor Trap Flag

The **monitor trap flag** is a debugging feature that causes VM exits to occur on certain instruction boundaries in VMX non-root operation. Such VM exits are called **MTF VM exits**. An MTF VM exit may occur on an instruction boundary in VMX non-root operation as follows:

- If the “monitor trap flag” VM-execution control is 1 and VM entry is injecting a vectored event (see Section 27.6.1), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry.
- If VM entry is injecting a pending MTF VM exit (see Section 27.6.2), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry. This is the case even if the “monitor trap flag” VM-execution control is 0.
- If the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and a pending event (e.g., debug exception or interrupt) is delivered before an instruction can execute, an MTF VM exit is pending on the instruction boundary following delivery of the event (or any nested exception).
- Suppose that the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is a REP-prefixed string instruction:
  - If the first iteration of the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).
  - If the first iteration of the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary after that iteration.
- Suppose that the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is the XBEGIN instruction. In this case, an MTF VM exit is pending at the fallback instruction address of the XBEGIN instruction. This behavior applies regardless of whether advanced debugging of RTM transactional regions has been enabled (see Section 16.3.7, “RTM-Enabled Debugger Support,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- Suppose that the “monitor trap flag” VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is neither a REP-prefixed string instruction or the XBEGIN instruction:
  - If the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).
  - If the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary following execution of that instruction. If the instruction is INT1, INT3, or INTO, this boundary follows delivery of any software exception. If the instruction is INT n, this boundary follows delivery of a software interrupt. If the instruction is HLT, the MTF VM exit will be from the HLT activity state.

No MTF VM exit occurs if another VM exit occurs before reaching the instruction boundary on which an MTF VM exit would be pending (e.g., due to an exception or triple fault).

1. This item includes the cases of an invalid opcode exception—#UD—generated by the UD0, UD1, and UD2 instructions and a BOUND-range exceeded exception—#BR—generated by the BOUND instruction.
An MTF VM exit occurs on the instruction boundary on which it is pending unless a higher priority event takes precedence or the MTF VM exit is blocked due to the activity state:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over MTF VM exits. MTF VM exits take priority over debug-trap exceptions and lower priority events.
- No MTF VM exit occurs if the processor is in either the shutdown activity state or wait-for-SIPI activity state. If a non-maskable interrupt subsequently takes the logical processor out of the shutdown activity state without causing a VM exit, an MTF VM exit is pending after delivery of that interrupt.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 40.2 for details.

### 26.5.3 Translation of Guest-Physical Addresses Using EPT

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain physical addresses are treated as guest-physical addresses and are not used to access memory directly. Instead, guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.

Details of the EPT mechanism are given in Section 29.3.

### 26.5.4 Translation of Guest-Physical Addresses Used by Intel Processor Trace

As described in Chapter 33, Intel® Processor Trace (IntelPT) captures information about software execution using dedicated hardware facilities.

Intel PT can be configured so that the trace output is written to memory using physical addresses. For example, when the ToPA (table of physical addresses) output mechanism is used, the IA32_RTIT_OUTPUT_BASE MSR contains the physical address of the base of the current ToPA. Each entry in that table contains the physical address of an output region in memory. When an output region becomes full, the ToPA output mechanism directs subsequent trace output to the next output region as indicated in the ToPA.

When the “Intel PT uses guest physical addresses” VM-execution control is 1, the logical processor treats the addresses used by Intel PT (the output addresses as well as those used to discover the output addresses) as guest-physical addresses, translating to physical addresses using EPT before trace output is written to memory.

Translating these addresses through EPT implies that the trace-output mechanism may cause EPT violations and VM exits; details are provided in Section 26.5.4.1. Section 26.5.4.2 describes a mechanism that ensures that these VM exits do not cause loss of trace data.

#### 26.5.4.1 Guest-Physical Address Translation for Intel PT: Details

When the “Intel PT uses guest physical addresses” VM-execution control is 1, the addresses used by Intel PT are treated as guest-physical addresses and translated using EPT. These addresses include the addresses of the output regions as well as the addresses of the ToPA entries that contain the output-region addresses.

Translation of accesses by the trace-output process may result in EPT violations or EPT misconfigurations (Section 29.3.3), resulting in VM exits. EPT violations resulting for the trace-output process always cause VM exits and are never converted to virtualization exceptions (Section 26.5.7.1).

If no EPT violation or EPT misconfiguration occurs and if page-modification logging (Section 29.3.6) is enabled, the address of an output region may be added to the page-modification log. If the log is full, a page-modification log-full event occurs, resulting in a VM exit.

If the “virtualize APIC accesses” VM-execution control is 1, a guest-physical address used by the trace-output process may be translated to an address on the APIC-access page. In this case, the access by the trace-output process causes an APIC-access VM exit as discussed in Section 30.4.6.1.
26.5.4.2 Trace-Address Pre-Translation (TAPT)

Because it buffers trace data produced by Intel PT before it is written to memory, the processor ensures that buffered data is not lost when a VM exit disables Intel PT. Specifically, the processor ensures that there is sufficient space left in the current output page for the buffered data. If this were not done, buffered trace data could be lost and the resulting trace corrupted.

To prevent the loss of buffered trace data, the processor uses a mechanism called trace-address pre-translation (TAPT). With TAPT, the processor translates using EPT the guest-physical address of the current output region before that address would be used to write buffered trace data to memory.

Because of TAPT, no translation (and thus no EPT violation) occurs at the time output is written to memory; the writes to memory use translations that were cached as part of TAPT. (The details given in Section 26.5.4.1 apply to TAPT.) TAPT ensures that, if a write to the output region would cause an EPT violation, the resulting VM exit is delivered at the time of TAPT, before the region would be used. This allows software to resolve the EPT violation at that time and ensures that, when it is necessary to write buffered trace data to memory, that data will not be lost due to an EPT violation.

TAPT (and resulting VM exits) may occur at any of the following times:

- When software in VMX non-root operation enables tracing by loading the IA32_RTIT_CTL MSR to set the TraceEn bit, using the WRMSR instruction or the XRSTORS instruction. Any VM exit resulting from TAPT in this case is trap-like: the WRMSR or XRSTORS completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).
- At an instruction boundary when one output region becomes full and Intel PT transitions to the next output region.
  VM exits resulting from TAPT in this case take priority over any pending debug exceptions. Such a VM exit will save information about such exceptions in the guest-state area of the VMCS.
- As part of a VM entry that enables Intel PT. See Section 27.5 for details.

TAPT may translate not only the guest-physical address of the current output region but those of subsequent output regions as well. (Doing so may provide better protection of trace data.) This implies that any VM exits resulting from TAPT may result from the translation of output-region addresses other than that of the current output region.

26.5.5 APIC Virtualization

APIC virtualization is a collection of features that can be used to support the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC). When APIC virtualization is enabled, the processor emulates many accesses to the APIC, tracks the state of the virtual APIC, and delivers virtual interrupts — all in VMX non-root operation without a VM exit.

Details of the APIC virtualization are given in Chapter 30.

26.5.6 VM Functions

A VM function is an operation provided by the processor that can be invoked from VMX non-root operation without a VM exit. VM functions are enabled and configured by the settings of different fields in the VMCS. Software in VMX non-root operation invokes a VM function with the VMFUNC instruction; the value of EAX selects the specific VM function being invoked.

Section 26.5.6.1 explains how VM functions are enabled. Section 26.5.6.2 specifies the behavior of the VMFUNC instruction. Section 26.5.6.3 describes a specific VM function called EPTP switching.

26.5.6.1 Enabling VM Functions

Software enables VM functions generally by setting the “enable VM functions” VM-execution control. A specific VM function is enabled by setting the corresponding VM-function control.
Suppose, for example, that software wants to enable EPTP switching (VM function 0; see Section 25.6.14). To do so, it must set the “activate secondary controls” VM-execution control (bit 31 of the primary processor-based VM-execution controls), the “enable VM functions” VM-execution control (bit 13 of the secondary processor-based VM-execution controls) and the “EPTP switching” VM-function control (bit 0 of the VM-function controls).

### 26.5.6.2 General Operation of the VMFUNC Instruction

The VMFUNC instruction causes an invalid-opcode exception (UD) if the “enable VM functions” VM-execution controls is 01 or the value of EAX is greater than 63 (only VM functions 0–63 can be enable). Otherwise, the instruction causes a VM exit if the bit at position EAX is 0 in the VM-function controls (the selected VM function is not enabled). If such a VM exit occurs, the basic exit reason used is 59 (3BH), indicating “VMFUNC”, and the length of the VMFUNC instruction is saved into the VM-exit instruction-length field. If the instruction causes neither an invalid-opcode exception nor a VM exit due to a disabled VM function, it performs the functionality of the VM function specified by the value in EAX.

Individual VM functions may perform additional fault checking (e.g., one might cause a general-protection exception if CPL > 0). In addition, specific VM functions may include checks that might result in a VM exit. If such a VM exit occurs, VM-exit information is saved as described in the previous paragraph. The specification of a VM function may indicate that additional VM-exit information is provided.

The specific behavior of the EPTP-switching VM function (including checks that result in VM exits) is given in Section 26.5.6.3.

### 26.5.6.3 EPTP Switching

EPTP switching is VM function 0. This VM function allows software in VMX non-root operation to load a new value for the EPT pointer (EPTP), thereby establishing a different EPT paging-structure hierarchy (see Section 29.3 for details of the operation of EPT). Software is limited to selecting from a list of potential EPTP values configured in advance by software in VMX root operation.

Specifically, the value of ECX is used to select an entry from the EPTP list, the 4-KByte structure referenced by the EPTP-list address (see Section 25.6.14; because this structure contains 512 8-Byte entries, VMFUNC causes a VM exit if ECX ≥ 512). The EPTP value in the selected entry is evaluated to determine whether it is valid for EPTP switching: a value is valid if (1) it is the same as the current EPTP value in bits 5:3 (these bits specify the EPT page-walk length); and (2) it would not cause VM entry to fail (see Section 27.2.1.1). If the value is invalid, a VM exit occurs. Otherwise, the value is stored in the EPTP field of the current VMCS and is used for subsequent accesses using guest-physical addresses. The following pseudocode provides details:

```
IF ECX ≥ 512
    THEN VM exit;
ELSE
    tent_EPTP := 8 bytes from EPTP-list address + 8 * ECX;
    IF tent_EPTP is not a valid EPTP value (would cause VM entry to fail if in EPTP or change EPT page-walk length)
        THEN VM exit;
    ELSE
        write tent_EPTP to the EPTP field in the current VMCS;
        use tent_EPTP as the new EPTP value for address translation;
        IF processor supports the 1-setting of the “EPT-violation #VE” VM-execution control
            THEN
                write ECX[15:0] to EPTP-index field in current VMCS;
                use ECX[15:0] as EPTP index for subsequent EPT-violation virtualization exceptions (see Section 26.5.7.2);
            FI;
        FI;
FI;
```

Execution of the EPTP-switching VM function does not modify the state of any registers; no flags are modified.

1. “Enable VM functions” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “enable VM functions” VM-execution control were 0. See Section 25.6.2.
If the “Intel PT uses guest physical addresses” VM-execution control is 1 and IA32_RTIT_CTL.TraceEn = 1, any execution of the EPTP-switching VM function causes a VM exit.\(^1\)

As noted in Section 26.5.6.2, an execution of the EPTP-switching VM function that causes a VM exit (as specified above), uses the basic exit reason 59, indicating “VMFUNC”. The length of the VMFUNC instruction is saved into the VM-exit instruction-length field. No additional VM-exit information is provided.

An execution of VMFUNC loads EPTP from the EPTP list (and thus does not cause a fault or VM exit) is called an **EPTP-switching VMFUNC**. After an EPTP-switching VMFUNC, control passes to the next instruction. The logical processor starts creating and using guest-physical and combined mappings associated with the new value of bits 51:12 of EPTP; the combined mappings created and used are associated with the current VPID and PCID (these are not changed by VMFUNC).\(^2\) If the “enable VPID” VM-execution control is 0, an EPTP-switching VMFUNC invalidates combined mappings associated with VPID 0000H (for all PCIDs and for all EPTRTA values, where EPTRTA is the value of bits 51:12 of EPTP).

Because an EPTP-switching VMFUNC may change the translation of guest-physical addresses, it may affect use of the guest-physical address in CR3. The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration due to the translation of that guest-physical address through the new EPT paging structures. The following items provide details that apply if CR0.PG = 1:

- If 32-bit paging or 4-level paging\(^3\) is in use (either CR4.PAE = 0 or IA32_EFER.LMA = 1), the next memory access with a linear address uses the translation of the guest-physical address in CR3 through the new EPT paging structures. As a result, this access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.
- If PAE paging is in use (CR4.PAE = 1 and IA32_EFER.LMA = 0), an EPTP-switching VMFUNC **does not** load the four page-directory-pointer-table entries (PDPTEs) from the guest-physical address in CR3. The logical processor continues to use the four guest-physical addresses already present in the PDPTEs. The guest-physical address in CR3 is not translated through the new EPT paging structures (until some operation that would load the PDPTEs). The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during the translation of a guest-physical address in any of the PDPTEs. A subsequent memory access with a linear address uses the translation of the guest-physical address in the appropriate PDPTE through the new EPT paging structures. As a result, such an access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.

If an EPTP-switching VMFUNC establishes an EPTP value that enables accessed and dirty flags for EPT (by setting bit 6), subsequent memory accesses may fail to set those flags as specified if there has been no appropriate execution of INVEPT since the last use of an EPTP value that does not enable accessed and dirty flags for EPT (because bit 6 is clear) and that is identical to the new value on bits 51:12.

If the processor supports the 1-setting of the “EPT-violation #VE” VM-execution control, an EPTP-switching VMFUNC loads the value in ECX[15:0] into to EPTP-index field in current VMCS. Subsequent EPT-violation virtualization exceptions will save this value into the virtualization-exception information area (see Section 26.5.7.2).

### 26.5.7 Virtualization Exceptions

A **virtualization exception** is a new processor exception. It uses vector 20 and is abbreviated #VE.

A virtualization exception can occur only in VMX non-root operation. Virtualization exceptions occur only with certain settings of certain VM-execution controls. Generally, these settings imply that certain conditions that would normally cause VM exits instead cause virtualization exceptions.

In particular, the 1-setting of the “EPT-violation #VE” VM-execution control causes some EPT violations to generate virtualization exceptions instead of VM exits. Section 26.5.7.1 provides the details of how the processor determines whether an EPT violation causes a virtualization exception or a VM exit.

---

1. Such a VM exit ensures the proper recording of trace data that might otherwise be lost during the change of EPT paging-structure hierarchy. Software handling the VM exit can change emulate the VM function and then resume the guest.
2. If the “enable VPID” VM-execution control is 0, the current VPID is 0000H; if CR4.PCIDE = 0, the current PCID is 000H.
3. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.
When the processor encounters a virtualization exception, it saves information about the exception to the virtualization-exception information area; see Section 26.5.7.2.

After saving virtualization-exception information, the processor delivers a virtualization exception as it would any other exception; see Section 26.5.7.3 for details.

### 26.5.7.1 Convertible EPT Violations

If the "EPT-violation #VE" VM-execution control is 0 (e.g., on processors that do not support this feature), EPT violations always cause VM exits. If instead the control is 1, certain EPT violations may be converted to cause virtualization exceptions instead; such EPT violations are **convertible**.

The values of certain EPT paging-structure entries determine which EPT violations are convertible. Specifically, bit 63 of certain EPT paging-structure entries may be defined to mean **suppress #VE**:

- If bits 2:0 of an EPT paging-structure entry are all 0, the entry is not **present**.\(^1\) If the processor encounters such an entry while translating a guest-physical address, it causes an EPT violation. The EPT violation is convertible if and only if bit 63 of the entry is 0.
- If an EPT paging-structure entry is present, the following cases apply:
  - If the value of the EPT paging-structure entry is not supported, the entry is **misconfigured**. If the processor encounters such an entry while translating a guest-physical address, it causes an EPT misconfiguration (not an EPT violation). EPT misconfigurations always cause VM exits.
  - If the value of the EPT paging-structure entry is supported, the following cases apply:
    - If bit 7 of the entry is 1, or if the entry is an EPT PTE, the entry maps a page. If the processor uses such an entry to translate a guest-physical address, and if an access to that address causes an EPT violation, the EPT violation is convertible if and only if bit 63 of the entry is 0.
    - If bit 7 of the entry is 0 and the entry is not an EPT PTE, the entry references another EPT paging structure. The processor does not use the value of bit 63 of the entry to determine whether any subsequent EPT violation is convertible.

If an access to a guest-physical address causes an EPT violation, bit 63 of exactly one of the EPT paging-structure entries used to translate that address is used to determine whether the EPT violation is convertible: either a entry that is not present (if the guest-physical address does not translate to a physical address) or an entry that maps a page (if it does).

A convertible EPT violation instead causes a virtualization exception if the following all hold:

- CR0.PE = 1;
- the logical processor is not in the process of delivering an event through the IDT;
- the EPT violation did not cause a shadow stack to become prematurely busy (see Section 26.4.3);
- the EPT violation does not result from the output process of Intel Processor Trace (Section 26.5.4); and
- the 32 bits at offset 4 in the virtualization-exception information area are all 0.

Delivery of virtualization exceptions writes the value FFFFFFFFH to offset 4 in the virtualization-exception information area (see Section 26.5.7.2). Thus, once a virtualization exception occurs, another can occur only if software clears this field.

### 26.5.7.2 Virtualization-Exception Information

Virtualization exceptions save data into the virtualization-exception information area (see Section 25.6.20). Table 26-1 enumerates the data saved and the format of the area.

A VMM may allow guest software to access the virtualization-exception information area. If it does, the guest software may modify that memory (e.g., to clear the 32-bit value at offset 4; see Section 26.5.7.1). (This is an exception to the general requirement given in Section 25.11.4.)

\(^1\) If the "mode-based execute control for EPT" VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 or bit 10 is 1.
26.5.7.3 Delivery of Virtualization Exceptions

After saving virtualization-exception information, the processor treats a virtualization exception as it does other exceptions:

- If bit 20 (#VE) is 1 in the exception bitmap in the VMCS, a virtualization exception causes a VM exit (see below). If the bit is 0, the virtualization exception is delivered using gate descriptor 20 in the IDT.
- Virtualization exceptions produce no error code. Delivery of a virtualization exception pushes no error code on the stack.
- With respect to double faults, virtualization exceptions have the same severity as page faults. If delivery of a virtualization exception encounters a nested fault that is either contributory or a page fault, a double fault (#DF) is generated. See Chapter 6, “Interrupt 8—Double Fault Exception (#DF)” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

It is not possible for a virtualization exception to be encountered while delivering another exception (see Section 26.5.7.1).

If a virtualization exception causes a VM exit directly (because bit 20 is 1 in the exception bitmap), information about the exception is saved normally in the VM-exit interruption information field in the VMCS (see Section 28.2.2). Specifically, the event is reported as a hardware exception with vector 20 and no error code. Bit 12 of the field (NMI unblocking due to IRET) is set normally.

If a virtualization exception causes a VM exit indirectly (because bit 20 is 0 in the exception bitmap and delivery of the exception generates an event that causes a VM exit), information about the exception is saved normally in the IDT-vectoring information field in the VMCS (see Section 28.2.4). Specifically, the event is reported as a hardware exception with vector 20 and no error code.

26.5.8 PASID Translation

The ENQCMD and ENQCMDS instructions each performs a 64-byte enqueue store that includes a 20-bit PASID value in bits 19:0. For ENQCMD, the PASID is normally the value of IA32_PASID[19:0], while for ENQCMDS, the PASID is normally read from memory.

If the "PASID translation" VM-execution control is 1, the PASID value identified in the previous paragraph is treated as a guest PASID. PASID translation converts this guest PASID to a 20-bit host PASID. After this translation, the enqueue store is performed, using the host PASID in place of the guest PASID.

PASID translation is implemented by two hierarchies of data structures (PASID-translation hierarchies) configured by a VMM. Guest PASIDs 00000H to 7FFFFH are translated through the low PASID-translation hierarchy, while guest PASIDs 80000 to FFFFFH are translated through the high PASID-translation hierarchy.
The root of each PASID-translation hierarchy is a 4-KByte **PASID directory**. The low PASID directory is located at the low PASID directory address, and the high PASID directory is located at the high PASID directory address (these physical addresses are VM-execution control fields in the VMCS). A PASID directory comprises 512 8-byte entries, each of which has the following format:

- Bit 0 is the entry’s present bit. The entry is used only if this bit is 1.
- Bits 11:1 are reserved and must be 0.
- Bits M–1:12 specify the 4-KByte aligned address of a PASID table (see below), where M is the processor’s physical-address width.
- Bits 63:M are reserved and must be 0.

A PASID-translation hierarchy also includes up to 512 4-KByte **PASID tables**; each of these is referenced by a PASID directory entry (see above). A PASID table comprises 1024 4-byte entries, each of which has the following format:

- Bits 19:0 are the host PASID specified by the entry.
- Bits 30:20 are reserved and must be 0.
- Bits 31 is the entry’s valid bit. The entry is used only if this bit is 1.

When PASID translation is enabled, the guest PASID determined by the instruction (see above) is converted to a host PASID using the following process:

- If bit 19 of guest PASID is clear, the low PASID directory is used; otherwise, the high PASID directory is used.
- Bits 18:10 of the guest PASID select an entry from the PASID directory. A VM exit occurs if the entry’s present bit is clear or if any reserved bit is set. Otherwise, bits M:0 of the entry (with bit 0 cleared) contain the physical address of a PASID table.
- Bits 9:0 of the guest PASID select an entry from the PASID table. A VM exit occurs if the entry’s valid bit is clear or if any reserved bit is set. Otherwise, bits 19:0 of the entry are the host PASID.

If PASID translation results in a VM exit (due to a present or valid bit being clear, or a reserved bit being set), the instruction does not complete and no enqueue store is performed.

### 26.6 UNRESTRICTED GUESTS

The first processors to support VMX operation require CR0.PE and CR0.PG to be 1 in VMX operation (see Section 24.8). This restriction implies that guest software cannot be run in unpaged protected mode or in real-address mode. Later processors support a VM-execution control called “unrestricted guest”. If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation. Such processors allow guest software to run in unpaged protected mode or in real-address mode. The following items describe the behavior of such software:

- The MOV CR0 instructions does not cause a general-protection exception simply because it would set either CR0.PE and CR0.PG to 0. See Section 26.3 for details.
- A logical processor treats the values of CR0.PE and CR0.PG in VMX non-root operation just as it does outside VMX operation. Thus, if CR0.PE = 0, the processor operates as it does normally in real-address mode (for example, it uses the 16-bit interrupt table to deliver interrupts and exceptions). If CR0.PG = 0, the processor operates as it does normally when paging is disabled.  
- Processor operation is modified by the fact that the processor is in VMX non-root operation and by the settings of the VM-execution controls just as it is in protected mode or when paging is enabled. Instructions, interrupts, and exceptions that cause VM exits in protected mode or when paging is enabled also do so in real-address mode or when paging is disabled. The following examples should be noted:
  - If CR0.PG = 0, page faults do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, invalid-TSS exceptions do not occur and thus cannot cause VM exits.

---

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the “unrestricted guest” VM-execution control were 0. See Section 25.6.2.
— If CR0.PE = 0, the following instructions cause invalid-opcode exceptions and do not cause VM exits: INVEPT, INVVPID, LLDT, LTR, SLDT, STR, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTREST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.

- If CR0.PG = 0, each linear address is passed directly to the EPT mechanism for translation to a physical address. The guest memory type passed on to the EPT mechanism is WB (writeback).

---

1. As noted in Section 27.2.1.1, the "enable EPT" VM-execution control must be 1 if the "unrestricted guest" VM-execution control is 1.
Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. VMLAUNCH can be used only with a VMCS whose launch state is clear and VMRESUME can be used only with a VMCS whose the launch state is launched. VMLAUNCH should be used for the first VM entry after VMCLEAR; VMRESUME should be used for subsequent VM entries with the same VMCS.

Each VM entry performs the following steps in the order indicated:

1. Basic checks are performed to ensure that VM entry can commence (Section 27.1).
2. The control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation and that the VMCS is correctly configured to support the next VM exit (Section 27.2).
3. The following may be performed in parallel or in any order (Section 27.3):
   - The guest-state area of the VMCS is checked to ensure that, after the VM entry completes, the state of the logical processor is consistent with IA-32 and Intel 64 architectures.
   - Processor state is loaded from the guest-state area and based on controls in the VMCS.
   - Address-range monitoring is cleared.
4. MSRs are loaded from the VM-entry MSR-load area (Section 27.4).
5. If VMLAUNCH is being executed, the launch state of the VMCS is set to “launched.”
6. If the “Intel PT uses guest physical addresses” VM-execution control is 1, trace-address pre-translation (TAPT) may occur (see Section 26.5.4 and Section 27.5).
7. An event may be injected in the guest context (Section 27.6).

Steps 1–4 above perform checks that may cause VM entry to fail. Such failures occur in one of the following three ways:

- Some of the checks in Section 27.1 may generate ordinary faults (for example, an invalid-opcode exception). Such faults are delivered normally.
- Some of the checks in Section 27.1 and all the checks in Section 27.2 cause control to pass to the instruction following the VM-entry instruction. The failure is indicated by setting RFLAGS.ZF or RFLAGS.CF (if there is no current VMCS). If there is a current VMCS, an error number indicating the cause of the failure is stored in the VM-instruction error field. See Chapter 31 for the error numbers.
- The checks in Section 27.3 and Section 27.4 cause processor state to be loaded from the host-state area of the VMCS (as would be done on a VM exit). Information about the failure is stored in the VM-exit information fields. See Section 27.8 for details.

EFLAGS.TF = 1 causes a VM-entry instruction to generate a single-step debug exception only if failure of one of the checks in Section 27.1 and Section 27.2 causes control to pass to the following instruction. A VM-entry does not generate a single-step debug exception in any of the following cases: (1) the instruction generates a fault; (2) failure of one of the checks in Section 27.3 or in loading MSRs causes processor state to be loaded from the host-state area of the VMCS; or (3) the instruction passes all checks in Section 27.1, Section 27.2, and Section 27.3 and there is no failure in loading MSRs.

Section 32.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, code running in SMM returns using VMentries instead of the RSM instruction. A VM entry returns from SMM if it is executed in SMM and the “entry to SMM” VM-entry control is 0. VM entries that return from SMM differ from ordinary VM entries in ways that are detailed in Section 32.15.4.

---

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For IA-32 processors, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
27.1 BASIC VM-ENTRY CHECKS

Before a VM entry commences, the current state of the logical processor is checked in the following order:

1. If the logical processor is in virtual-8086 mode or compatibility mode, an invalid-opcode exception is generated.
2. If the current privilege level (CPL) is not zero, a general-protection exception is generated.
3. If there is no current VMCS, RFLAGS.CF is set to 1 and control passes to the next instruction.
4. If there is a current VMCS but the current VMCS is a shadow VMCS (see Section 25.10), RFLAGS.CF is set to 1 and control passes to the next instruction.
5. If there is a current VMCS that is not a shadow VMCS, the following conditions are evaluated in order; any of these cause VM entry to fail:
   a. If there is MOV-SS blocking (see Table 25-3).
   b. If the VM entry is invoked by VMLAUNCH and the VMCS launch state is not clear.
   c. If the VM entry is invoked by VMRESUME and the VMCS launch state is not launched.

If any of these checks fail, RFLAGS.ZF is set to 1 and control passes to the next instruction. An error number indicating the cause of the failure is stored in the VM-instruction error field. See Chapter 31 for the error numbers.

27.2 CHECKS ON VMX CONTROLS AND HOST-STATE AREA

If the checks in Section 27.1 do not cause VM entry to fail, the control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation, that the VMCS is correctly configured to support the next VM exit, and that, after the next VM exit, the processor's state is consistent with the Intel 64 and IA-32 architectures.

VM entry fails if any of these checks fail. When such failures occur, control is passed to the next instruction, RFLAGS.ZF is set to 1 to indicate the failure, and the VM-instruction error field is loaded with an error number that indicates whether the failure was due to the controls or the host-state area (see Chapter 31).

These checks may be performed in any order. Thus, an indication by error number of one cause (for example, host state) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same VMCS. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The checks on the controls and the host-state area are presented in Section 27.2.1 through Section 27.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

27.2.1 Checks on VMX Controls

This section identifies VM-entry checks on the VMX control fields.

27.2.1.1 VM-Execution Control Fields

VM entries perform the following checks on the VM-execution control fields:  

- Reserved bits in the pin-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.3.1).

1. If the "activate secondary controls" primary processor-based VM-execution control is 0, VM entry operates as if each secondary processor-based VM-execution control were 0. Similarly, if the "activate tertiary controls" primary processor-based VM-execution control is 0, VM entry operates as if each tertiary processor-based VM-execution control were 0. See Section 25.6.2.
Reserved bits in the primary processor-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.3.2).

If the “activate secondary controls” primary processor-based VM-execution control is 1, reserved bits in the secondary processor-based VM-execution controls must be cleared. Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.3.3).

If the “activate secondary controls” primary processor-based VM-execution control is 0 (or if the processor does not support the 1-setting of that control), no checks are performed on the secondary processor-based VM-execution controls. The logical processor operates as if all the secondary processor-based VM-execution controls were 0.

If the “activate tertiary controls” primary processor-based VM-execution control is 1, reserved bits in the tertiary processor-based VM-execution controls must be cleared. Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.3.4).

If the “activate tertiary controls” primary processor-based VM-execution control is 0 (or if the processor does not support the 1-setting of that control), no checks are performed on the tertiary processor-based VM-execution controls. The logical processor operates as if all the tertiary processor-based VM-execution controls were 0.

The CR3-target count must not be greater than 4. Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC to determine the number of values supported (see Appendix A.6).

If the “use I/O bitmaps” VM-execution control is 1, bits 11:0 of each I/O-bitmap address must be 0. Neither address should set any bits beyond the processor’s physical-address width.\(^1\),\(^2\)

If the “use MSR bitmaps” VM-execution control is 1, bits 11:0 of the MSR-bitmap address must be 0. The address should not set any bits beyond the processor’s physical-address width.\(^3\)

If the “use TPR shadow” VM-execution control is 1, the virtual-APIC address must satisfy the following checks:

- Bits 11:0 of the address must be 0.
- The address should not set any bits beyond the processor’s physical-address width.\(^4\)

If all of the above checks are satisfied and the “use TPR shadow” VM-execution control is 1, bytes 3:1 of VTPR (see Section 30.1.1) may be cleared (behavior may be implementation-specific).

The clearing of these bytes may occur even if the VM entry fails. This is true either if the failure causes control to pass to the instruction following the VM-entry instruction or if it causes processor state to be loaded from the host-state area of the VMCS.

If the “use TPR shadow” VM-execution control is 1 and the “virtual-interrupt delivery” VM-execution control is 0, bits 31:4 of the TPR threshold VM-execution control field must be 0.

The following check is performed if the “use TPR shadow” VM-execution control is 1 and the “virtualize APIC accesses” and “virtual-interrupt delivery” VM-execution controls are both 0: the value of bits 3:0 of the TPR threshold VM-execution control field should not be greater than the value of bits 7:4 of VTPR (see Section 30.1.1).

If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” VM-execution control must be 0.

If the “virtual NMIs” VM-execution control is 0, the “NMI-window exiting” VM-execution control must be 0.

If the “virtualize APIC-accesses” VM-execution control is 1, the APIC-access address must satisfy the following checks:

- Bits 11:0 of the address must be 0.
- The address should not set any bits beyond the processor’s physical-address width.\(^5\)

---

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. If IA32_VMX_BASIC[48] is read as 1, these addresses must not set any bits in the range 63:32; see Appendix A.1.
3. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.
4. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.
5. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.
• If the “use TPR shadow” VM-execution control is 0, the following VM-execution controls must also be 0: “virtualize x2APIC mode”, “APIC-register virtualization”, “virtual-interrupt delivery”, and “IPI virtualization”.

• If the “virtualize x2APIC mode” VM-execution control is 1, the “virtualize APIC accesses” VM-execution control must be 0.

• If the “virtual-interrupt delivery” VM-execution control is 1, the “external-interrupt exiting” VM-execution control must be 1.

• If the “process posted interrupts” VM-execution control is 1, the following must be true:
  — The “virtual-interrupt delivery” VM-execution control is 1.
  — The “acknowledge interrupt on exit” VM-exit control is 1.
  — The posted-interrupt notification vector has a value in the range 0–255 (bits 15:8 are all 0).
  — Bits 5:0 of the posted-interrupt descriptor address are all 0.
  — The posted-interrupt descriptor address does not set any bits beyond the processor’s physical-address width.1

• If the “IPI virtualization” VM-execution control is 1, the following must be true:
  — Bits 2:0 of the PID-pointer table address are all 0.
  — The PID-pointer table address does not set any bits beyond the processor’s physical-address width.
  — The address of the last entry in the PID-pointer table does not set any bits beyond the processor’s physical-address width. (This address is the PID-pointer table address plus 8 times the last PID-pointer index.)

• If the “enable VPID” VM-execution control is 1, the value of the VPID VM-execution control field must not be 0000H.

• If the “enable EPT” VM-execution control is 1, the EPTP VM-execution control field (see Table 25-9 in Section 25.6.11) must satisfy the following checks:
  — The EPT memory type (bits 2:0) must be a value supported by the processor as indicated in the IA32_VMX_EPT_VPID_CAP MSR (see Appendix A.10).
  — Bits 5:3 must contain a value 1 less than an EPT page-walk length supported by the processor as indicated in the IA32_VMX_EPT_VPID_CAP MSR (see Section 29.3.2 and Appendix A.10).
  — Bit 6 (enable bit for accessed and dirty flags for EPT) must be 0 if bit 21 of the IA32_VMX_EPT_VPID_CAP MSR (see Appendix A.10) is read as 0, indicating that the processor does not support accessed and dirty flags for EPT.
  — Reserved bits 11:7 and 63:N (where N is the processor’s physical-address width) must all be 0.

• The “enable EPT” VM-execution control must be 1 if any of the following VM-execution controls is 1: “enable PML,” “unrestricted guest,” “mode-based execute control for EPT,” “sub-page write permissions for EPT,” “Intel PT uses guest physical addresses,” “enable HLAT,” “EPT paging-write control,” or “guest-paging verification.”

• If the “enable PML” VM-execution control is 1, the PML address must satisfy the following checks:
  — Bits 11:0 of the address must be 0.
  — The address should not set any bits beyond the processor’s physical-address width.

• If the “sub-page write permissions for EPT” VM-execution control is 1, the SPPTP VM-execution control field (see Table 25-11 in Section 25.6.22) must satisfy the following checks:
  — Bits 11:0 of the address must be 0.
  — The address should not set any bits beyond the processor’s physical-address width.

• If the “enable VM functions” processor-based VM-execution control is 1, reserved bits in the VM-function controls must be clear. Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.11). In addition, the following check is performed based on the setting of bits in the VM-function controls (see Section 25.6.14):

1. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix A.1.
— If “EPTP switching” VM-function control is 1, the “enable EPT” VM-execution control must also be 1. In addition, the EPTP-list address must satisfy the following checks:
  • Bits 11:0 of the address must be 0.
  • The address must not set any bits beyond the processor’s physical-address width.

If the “enable VM functions” processor-based VM-execution control is 0, no checks are performed on the VM-function controls.

• If the “VMCS shadowing” VM-execution control is 1, the VMREAD-bitmap and VMWRITE-bitmap addresses must each satisfy the following checks:
  — Bits 11:0 of the address must be 0.
  — The address must not set any bits beyond the processor’s physical-address width.

• If the “EPT-violation #VE” VM-execution control is 1, the virtualization-exception information address must satisfy the following checks:
  — Bits 11:0 of the address must be 0.
  — The address must not set any bits beyond the processor’s physical-address width.

• If the logical processor is operating with Intel PT enabled (if IA32_RTIT_CTL.TraceEn = 1) at the time of VM entry, the “load IA32_RTIT_CTL” VM-entry control must be 0.

• If the “Intel PT uses guest physical addresses” VM-execution control is 1, the “load IA32_RTIT_CTL” VM-entry control and the “clear IA32_RTIT_CTL” VM-exit control must both be 1.

• If the “use TSC scaling” VM-execution control is 1, the TSC-multiplier must not be zero.

• If the “enable HLAT” VM-execution control is 1, the following bits in the HLATP VM-execution control field (see Table 25-12 in Section 25.6.23) must be zero: bits 2:0, bits 11:5, and bits beyond the processor’s physical-address width.

• If the “PASID translation” VM-execution control is 1, the low PASID directory address and the high PASID directory address must each satisfy the following checks:
  — Bits 11:0 of the address must be 0.
  — The address must not set any bits beyond the processor’s physical-address width.

27.2.1.2  VM-Exit Control Fields

VM entries perform the following checks on the VM-exit control fields.

• Reserved bits in the primary VM-exit controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.4.1).

• If the “activate secondary controls” primary VM-exit control is 1, reserved bits in the secondary VM-exit controls must be cleared. Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.4.2).

• If the “activate secondary controls” primary VM-exit control is 0 (or if the processor does not support the 1-setting of that control), no checks are performed on the secondary VM-exit controls. The logical processor operates as if all the secondary VM-exit controls were 0.

• If the “activate VMX-preemption timer” VM-execution control is 0, the “save VMX-preemption timer value” VM-exit control must also be 0.

• The following checks are performed for the VM-exit MSR-store address if the VM-exit MSR-store count field is non-zero:
  — The lower 4 bits of the VM-exit MSR-store address must be 0. The address should not set any bits beyond the processor’s physical-address width.\(^1\)

---

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
VM ENTRIES

— The address of the last byte in the VM-exit MSR-store area should not set any bits beyond the processor’s physical-address width. The address of this last byte is VM-exit MSR-store address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor’s physical-address width.)

If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

• The following checks are performed for the VM-exit MSR-load address if the VM-exit MSR-load count field is non-zero:
  — The lower 4 bits of the VM-exit MSR-load address must be 0. The address should not set any bits beyond the processor’s physical-address width.
  — The address of the last byte in the VM-exit MSR-load area should not set any bits beyond the processor’s physical-address width. The address of this last byte is VM-exit MSR-load address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor’s physical-address width.)

If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

27.2.1.3 VM-Entry Control Fields

VM entries perform the following checks on the VM-entry control fields.

• Reserved bits in the VM-entry controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix A.5).

• Fields relevant to VM-entry event injection must be set properly. These fields are the VM-entry interruption-information field (see Table 25-17 in Section 25.8.3), the VM-entry exception error code, and the VM-entry instruction length. If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the following must hold:
  — The field’s interruption type (bits 10:8) is not set to a reserved value. Value 1 is reserved on all logical processors; value 7 (other event) is reserved on logical processors that do not support the 1-setting of the "monitor trap flag" VM-execution control.
  — The field’s vector (bits 7:0) is consistent with the interruption type:
    • If the interruption type is non-maskable interrupt (NMI), the vector is 2.
    • If the interruption type is hardware exception, the vector is at most 31.
    • If the interruption type is other event, the vector is 0 (pending MTF VM exit).
  — The field’s deliver-error-code bit (bit 11) is 1 if each of the following holds: (1) the interruption type is hardware exception; (2) bit 0 (corresponding to CR0.PE) is set in the CR0 field in the guest-state area; (3) IA32_VMX_BASIC[56] is read as 0 (see Appendix A.1); and (4) the vector indicates one of the following exceptions: #DF (vector 8), #TS (10), #NP (11), #GP (12), #PF (14), or #AC (17).
  — The field’s deliver-error-code bit is 0 if any of the following holds: (1) the interruption type is not hardware exception; (2) bit 0 is clear in the CR0 field in the guest-state area; or (3) IA32_VMX_BASIC[56] is read as 0 and the vector is in one of the following ranges: 0–7, 9, 15, 16, or 18–31.
  — Reserved bits in the field (30:12) are 0.
  — If the deliver-error-code bit (bit 11) is 1, bits 31:16 of the VM-entry exception error-code field are 0.
  — If the interruption type is software interrupt, software exception, or privileged software exception, the VM-entry instruction-length field is in the range 0–15. A VM-entry instruction length of 0 is allowed only if IA32_VMX_MISC[30] is read as 1; see Appendix A.6.

• The following checks are performed for the VM-entry MSR-load address if the VM-entry MSR-load count field is non-zero:
  — The lower 4 bits of the VM-entry MSR-load address must be 0. The address should not set any bits beyond the processor’s physical-address width.\(^1\)

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
VM ENTRIES

— The address of the last byte in the VM-entry MSR-load area should not set any bits beyond the processor’s physical-address width. The address of this last byte is VM-entry MSR-load address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor’s physical-address width.)

If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix A.1.

• If the processor is not in SMM, the “entry to SMM” and “deactivate dual-monitor treatment” VM-entry controls must be 0.
• The “entry to SMM” and “deactivate dual-monitor treatment” VM-entry controls cannot both be 1.

27.2.2 Checks on Host Control Registers, MSRs, and SSP

The following checks are performed on fields in the host-state area that correspond to control registers and MSRs:

• The CR0 field must not set any bit to a value not supported in VMX operation (see Section 24.8).  
• The CR4 field must not set any bit to a value not supported in VMX operation (see Section 24.8).
• If bit 23 in the CR4 field (corresponding to CET) is 1, bit 16 in the CR0 field (WP) must also be 1.
• On processors that support Intel 64 architecture, the CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width must be 0.  
• On processors that support Intel 64 architecture, the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field must each contain a canonical address.
• If the “load IA32_PERF_GLOBAL_CTRL” VM-exit control is 1, bits reserved in the IA32_PERF_GLOBAL_CTRL MSR must be 0 in the field for that register (see Figure 20-3).
• If the “load IA32_PAT” VM-exit control is 1, the value of the field for the IA32_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).
• If the “load IA32_EFER” VM-exit control is 1, bits reserved in the IA32_EFER MSR must be 0 in the field for that register. In addition, the values of the LMA and LME bits in the field must each be that of the “host address-space size” VM-exit control.
• If the “load CET state” VM-exit control is 1, the IA32_S_CET field must not set any bits reserved in the IA32_S_CET MSR, and bit 10 (corresponding to SUPPRESS) and bit 11 (TRACKER) in the field cannot both be set.
• If the “load CET state” VM-exit control is 1, bits 1:0 must be 0 in the SSP field.
• If the “load PKRS” VM-exit control is 1, bits 63:32 must be 0 in the IA32_PKRS field.

27.2.3 Checks on Host Segment and Descriptor-Table Registers

The following checks are performed on fields in the host-state area that correspond to segment and descriptor-table registers:

• In the selector field for each of CS, SS, DS, ES, FS, GS, and TR, the RPL (bits 1:0) and the TI flag (bit 2) must be 0.
• The selector fields for CS and TR cannot be 0000H.
• The selector field for SS cannot be 0000H if the “host address-space size” VM-exit control is 0.
• On processors that support Intel 64 architecture, the base-address fields for FS, GS, GDTR, IDTR, and TR must contain canonical addresses.

1. The bits corresponding to CR0.NW (bit 29) and CR0.CD (bit 30) are never checked because the values of these bits are not changed by VM exit; see Section 28.5.1.
2. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. Bit 63 of the CR3 field in the host-state area must be 0. This is true even though, if CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.
27.2.4 Checks Related to Address-Space Size

On processors that support Intel 64 architecture, the following checks related to address-space size are performed on VMX controls and fields in the host-state area:

- If the logical processor is outside IA-32e mode (if IA32_EFER.LMA = 0) at the time of VM entry, the following must hold:
  - The "IA-32e mode guest" VM-entry control is 0.
  - The "host address-space size" VM-exit control is 0.
- If the logical processor is in IA-32e mode (if IA32_EFER.LMA = 1) at the time of VM entry, the "host address-space size" VM-exit control must be 1.
- If the "host address-space size" VM-exit control is 0, the following must hold:
  - The "IA-32e mode guest" VM-entry control is 0.
  - Bit 17 of the CR4 field (corresponding to CR4.PCIDE) is 0.
  - Bits 63:32 in the RIP field are 0.
  - If the "load CET state" VM-exit control is 1, bits 63:32 in the IA32_S_CET field and in the SSP field are 0.
- If the "host address-space size" VM-exit control is 1, the following must hold:
  - Bit 5 of the CR4 field (corresponding to CR4.PAE) is 1.
  - The RIP field contains a canonical address.
  - If the "load CET state" VM-exit control is 1, the IA32_S_CET field and the SSP field contain canonical addresses.
- If the "load CET state" VM-exit control is 1, the IA32_INTERRUPT_SSP_TABLE_ADDR field contains a canonical address.

On processors that do not support Intel 64 architecture, checks are performed to ensure that the "IA-32e mode guest" VM-entry control and the "host address-space size" VM-exit control are both 0.

27.3 CHECKING AND LOADING GUEST STATE

If all checks on the VMX controls and the host-state area pass (see Section 27.2), the following operations take place concurrently: (1) the guest-state area of the VMCS is checked to ensure that, after the VM entry completes, the state of the logical processor is consistent with IA-32 and Intel 64 architectures; (2) processor state is loaded from the guest-state area or as specified by the VM-entry control fields; and (3) address-range monitoring is cleared.

Because the checking and the loading occur concurrently, a failure may be discovered only after some state has been loaded. For this reason, the logical processor responds to such failures by loading state from the host-state area, as it would for a VM exit. See Section 27.8.

27.3.1 Checks on the Guest State Area

This section describes checks performed on fields in the guest-state area. These checks may be performed in any order. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The following subsections reference fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

27.3.1.1 Checks on Guest Control Registers, Debug Registers, and MSRs

The following checks are performed on fields in the guest-state area corresponding to control registers, debug registers, and MSRs:
The CR0 field must not set any bit to a value not supported in VMX operation (see Section 24.8). The following are exceptions:

- Bit 0 (corresponding to CR0.PE) and bit 31 (PG) are not checked if the “unrestricted guest” VM-execution control is 1.1
- Bit 29 (corresponding to CR0.NW) and bit 30 (CD) are never checked because the values of these bits are not changed by VM entry; see Section 27.3.2.1.

If bit 31 in the CR0 field (corresponding to PG) is 1, bit 0 in that field (PE) must also be 1.2

The CR4 field must not set any bit to a value not supported in VMX operation (see Section 24.8).

If bit 23 in the CR4 field (corresponding to CET) is 1, bit 16 in the CR0 field (WP) must also be 1.

If the “load debug controls” VM-entry control is 1, bits reserved in the IA32_DEBUGCTL MSR must be 0 in the field for that register. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally.

The CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width are 0.4,5

If the “load debug controls” VM-entry control is 1, bits 63:32 in the DR7 field must be 0. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally (if they supported Intel 64 architecture).

- The IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field must each contain a canonical address.
- If the “load CET state” VM-entry control is 1, the IA32_S_CET field and the IA32_INTERRUPT_SSP_TABLE_ADDR field must contain canonical addresses.

If the “load IA32_PERF_GLOBAL_CTRL” VM-entry control is 1, bits reserved in the IA32_PERF_GLOBAL_CTRL MSR must be 0 in the field for that register (see Figure 20-3).

If the “load IA32_PAT” VM-entry control is 1, the value of the field for the IA32_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).

If the “load IA32_EFER” VM-entry control is 1, the following checks are performed on the field for the IA32_EFER MSR:

- Bits reserved in the IA32_EFER MSR must be 0.
- Bit 10 (corresponding to IA32_EFER.LMA) must equal the value of the “IA-32e mode guest” VM-entry control. It must also be identical to bit 8 (LME) if bit 31 in the CR0 field (corresponding to CR0.PG) is 1.6

---

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 25.6.2.
2. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
3. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
4. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
5. Bit 63 of the CR3 field in the guest-state area must be 0. This is true even though, if CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.
6. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
• If the “load IA32_BNDCFGS” VM-entry control is 1, the following checks are performed on the field for the IA32_BNDCFGS MSR:
  — Bits reserved in the IA32_BNDCFGS MSR must be 0.
  — The linear address in bits 63:12 must be canonical.
• If the “load IA32_RTIT_CTL” VM-entry control is 1, bits reserved in the IA32_RTIT_CTL MSR must be 0 in the field for that register (see Table 33-6).
• If the “load CET state” VM-entry control is 1, the IA32_S_CET field must not set any bits reserved in the IA32_S_CET MSR, and bit 10 (corresponding to SUPPRESS) and bit 11 (TRACKER) of the field cannot both be set.
• If the “load guest IA32_LBR_CTL” VM-entry control is 1, bits reserved in the IA32_LBR_CTL MSR must be 0 in the field for that register.
• If the “load PKRS” VM-entry control is 1, bits 63:32 must be 0 in the IA32_PKRS field.
• If the “load UINV” VM-entry control is 1, bits 15:8 must be 0 in the guest UINV field.

27.3.1.2  Checks on Guest Segment Registers
This section specifies the checks on the fields for CS, SS, DS, ES, FS, GS, TR, and LDTR. The following terms are used in defining these checks:
• The guest will be virtual-8086 if the VM flag (bit 17) is 1 in the RFLAGS field in the guest-state area.
• The guest will be IA-32e mode if the “IA-32e mode guest” VM-entry control is 1. (This is possible only on processors that support Intel 64 architecture.)
• Any one of these registers is said to be usable if the unusable bit (bit 16) is 0 in the access-rights field for that register.

The following are the checks on these fields:
• Selector fields.
  — TR. The TI flag (bit 2) must be 0.
  — LDTR. If LDTR is usable, the TI flag (bit 2) must be 0.
  — SS. If the guest will not be virtual-8086 and the “unrestricted guest” VM-execution control is 0, the RPL (bits 1:0) must equal the RPL of the selector field for CS.1
• Base-address fields.
  — CS, SS, DS, ES, FS, GS. If the guest will be virtual-8086, the address must be the selector field shifted left 4 bits (multiplied by 16).
  — The following checks are performed on processors that support Intel 64 architecture:
    • TR, FS, GS. The address must be canonical.
    • LDTR. If LDTR is usable, the address must be canonical.
    • CS. Bits 63:32 of the address must be zero.
    • SS, DS, ES. If the register is usable, bits 63:32 of the address must be zero.
• Limit fields for CS, SS, DS, ES, FS, GS. If the guest will be virtual-8086, the field must be 0000FFFFH.
• Access-rights fields.
  — CS, SS, DS, ES, FS, GS.
    • If the guest will be virtual-8086, the field must be 000000F3H. This implies the following:
      — Bits 3:0 (Type) must be 3, indicating an expand-up read/write accessed data segment.
      — Bit 4 (S) must be 1.

1. “Unrestricted guest” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “unrestricted guest” VM-execution control were 0. See Section 25.6.2.
— Bits 6:5 (DPL) must be 3.
— Bit 7 (P) must be 1.
— Bits 11:8 (reserved), bit 12 (software available), bit 13 (reserved/L), bit 14 (D/B), bit 15 (G), bit 16 (unusable), and bits 31:17 (reserved) must all be 0.

• If the guest will not be virtual-8086, the different sub-fields are considered separately:
  — Bits 3:0 (Type).
    • CS. The values allowed depend on the setting of the "unrestricted guest" VM-execution control:
      — If the control is 0, the Type must be 9, 11, 13, or 15 (accessed code segment).
      — If the control is 1, the Type must be either 3 (read/write accessed expand-up data segment) or one of 9, 11, 13, and 15 (accessed code segment).
    • SS. If SS is usable, the Type must be 3 or 7 (read/write, accessed data segment).
    • DS, ES, FS, GS. The following checks apply if the register is usable:
      — Bit 0 of the Type must be 1 (accessed).
      — If bit 3 of the Type is 1 (code segment), then bit 1 of the Type must be 1 (readable).
  — Bit 4 (S). If the register is CS or if the register is usable, S must be 1.
  — Bits 6:5 (DPL).
    • CS.
      — If the Type is 3 (read/write accessed expand-up data segment), the DPL must be 0. The Type can be 3 only if the "unrestricted guest" VM-execution control is 1.
      — If the Type is 9 or 11 (non-conforming code segment), the DPL must equal the DPL in the access-rights field for SS.
      — If the Type is 13 or 15 (conforming code segment), the DPL cannot be greater than the DPL in the access-rights field for SS.
    • SS.
      — If the "unrestricted guest" VM-execution control is 0, the DPL must equal the RPL from the selector field.
      — The DPL must be 0 either if the Type in the access-rights field for CS is 3 (read/write accessed expand-up data segment) or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.\footnote{The following apply if either the "unrestricted guest" VM-execution control or bit 31 of the primary processor-based VM-execution controls is 0: (1) bit 0 in the CR0 field must be 1 if the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation; and (2) the Type in the access-rights field for CS cannot be 3.}
    • DS, ES, FS, GS. The DPL cannot be less than the RPL in the selector field if (1) the "unrestricted guest" VM-execution control is 0; (2) the register is usable; and (3) the Type in the access-rights field is in the range 0 – 11 (data segment or non-conforming code segment).
  — Bit 7 (P). If the register is CS or if the register is usable, P must be 1.
  — Bits 11:8 (reserved). If the register is CS or if the register is usable, these bits must all be 0.
  — Bit 14 (D/B). For CS, D/B must be 0 if the guest will be IA-32e mode and the L bit (bit 13) in the access-rights field is 1.
  — Bit 15 (G). The following checks apply if the register is CS or if the register is usable:
    • If any bit in the limit field in the range 11:0 is 0, G must be 0.
    • If any bit in the limit field in the range 31:20 is 1, G must be 1.
  — Bits 31:17 (reserved). If the register is CS or if the register is usable, these bits must all be 0.
— TR. The different sub-fields are considered separately:
  • Bits 3:0 (Type).
    — If the guest will not be IA-32e mode, the Type must be 3 (16-bit busy TSS) or 11 (32-bit busy TSS).
    — If the guest will be IA-32e mode, the Type must be 11 (64-bit busy TSS).
  • Bit 4 (S). S must be 0.
  • Bit 7 (P). P must be 1.
  • Bits 11:8 (reserved). These bits must all be 0.
  • Bit 15 (G).
    — If any bit in the limit field in the range 11:0 is 0, G must be 0.
    — If any bit in the limit field in the range 31:20 is 1, G must be 1.
  • Bit 16 (Unusable). The unusable bit must be 0.
  • Bits 31:17 (reserved). These bits must all be 0.
— LDTR. The following checks on the different sub-fields apply only if LDTR is usable:
  • Bits 3:0 (Type). The Type must be 2 (LDT).
  • Bit 4 (S). S must be 0.
  • Bit 7 (P). P must be 1.
  • Bits 11:8 (reserved). These bits must all be 0.
  • Bit 15 (G).
    — If any bit in the limit field in the range 11:0 is 0, G must be 0.
    — If any bit in the limit field in the range 31:20 is 1, G must be 1.
  • Bits 31:17 (reserved). These bits must all be 0.

27.3.1.3 Checks on Guest Descriptor-Table Registers
The following checks are performed on the fields for GDTR and IDTR:
• On processors that support Intel 64 architecture, the base-address fields must contain canonical addresses.
• Bits 31:16 of each limit field must be 0.

27.3.1.4 Checks on Guest RIP, RFLAGS, and SSP
The following checks are performed on fields in the guest-state area corresponding to RIP, RFLAGS, and SSP (shadow-stack pointer):
• RIP. The following checks are performed on processors that support Intel 64 architecture:
  — Bits 63:32 must be 0 if the "IA-32e mode guest" VM-entry control is 0 or if the L bit (bit 13) in the access-rights field for CS is 0.
  — If the processor supports N < 64 linear-address bits, bits 63:N must be identical if the "IA-32e mode guest" VM-entry control is 1 and the L bit in the access-rights field for CS is 1.\(^1\) (No check applies if the processor supports 64 linear-address bits.) The guest RIP value is not required to be canonical; the value of bit N-1 may differ from that of bit N.
• RFLAGS.
  — Reserved bits 63:22 (bits 31:22 on processors that do not support Intel 64 architecture), bit 15, bit 5 and bit 3 must be 0 in the field, and reserved bit 1 must be 1.

\(^1\) Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.
— The VM flag (bit 17) must be 0 either if the "IA-32e mode guest" VM-entry control is 1 or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.

— The IF flag (RFLAGS[bit 9]) must be 1 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) is external interrupt.

• SSP. The following checks are performed if the "load CET state" VM-entry control is 1
  — Bits 1:0 must be 0.
  — If the processor supports the Intel 64 architecture, bits 63:N must be identical, where N is the CPU’s maximum linear-address width. (This check does not apply if the processor supports 64 linear-address bits.) The guest SSP value is not required to be canonical; the value of bit N-1 may differ from that of bit N.

27.3.1.5 Checks on Guest Non-Register State

The following checks are performed on fields in the guest-state area corresponding to non-register state:

• Activity state.
  — The activity-state field must contain a value in the range 0 – 3, indicating an activity state supported by the implementation (see Section 25.4.2). Future processors may include support for other activity states. Software should read the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine what activity states are supported.
  — The activity-state field must not indicate the HLT state if the DPL (bits 6:5) in the access-rights field for SS is not 0.
  — The activity-state field must indicate the active state if the interruptibility-state field indicates blocking by either MOV-SS or by STI (if either bit 0 or bit 1 in that field is 1).
  — If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the interruption to be delivered (as defined by interruption type and vector) must not be one that would normally be blocked while a logical processor is in the activity state corresponding to the contents of the activity-state field. The following items enumerate the interruptions (as specified in the VM-entry interruption-information field) whose injection is allowed for the different activity states:
    • Active. Any interruption is allowed.
    • HLT. The only events allowed are the following:
      — Those with interruption type external interrupt or non-maskable interrupt (NMI).
      — Those with interruption type hardware exception and vector 1 (debug exception) or vector 18 (machine-check exception).
      — Those with interruption type other event and vector 0 (pending MTF VM exit).
      See Table 25-17 in Section 25.8.3 for details regarding the format of the VM-entry interruption-information field.
    • Shutdown. Only NMIs and machine-check exceptions are allowed.
    • Wait-for-SIPI. No interruptions are allowed.
  — The activity-state field must not indicate the wait-for-SIPI state if the "entry to SMM" VM-entry control is 1.

• Interruptibility state.
  — The reserved bits (bits 31:5) must be 0.
  — The field cannot indicate blocking by both STI and MOV SS (bits 0 and 1 cannot both be 1).
  — Bit 0 (blocking by STI) must be 0 if the IF flag (bit 9) is 0 in the RFLAGS field.

---

1. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
2. As noted in Section 25.4.1, SS.DPL corresponds to the logical processor's current privilege level (CPL).
— Bit 0 (blocking by STI) and bit 1 (blocking by MOV-SS) must both be 0 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) in that field has value 0, indicating external interrupt, or value 2, indicating non-maskable interrupt (NMI).
— Bit 2 (blocking by SMI) must be 0 if the processor is not in SMM.
— Bit 2 (blocking by SMI) must be 1 if the "entry to SMM" VM-entry control is 1.
— Bit 3 (blocking by NMI) must be 0 if the "virtual NMIs" VM-execution control is 1, the valid bit (bit 31) in the VM-entry interruption-information field is 1, and the interruption type (bits 10:8) in that field has value 2 (indicating NMI).
— If bit 4 (enclave interruption) is 1, bit 1 (blocking by MOV-SS) must be 0 and the processor must support for SGX by enumerating CPUID.(EAX=07H,ECX=0):EBX.SGX[bit 2] as 1.

NOTE
If the "virtual NMIs" VM-execution control is 0, there is no requirement that bit 3 be 0 if the valid bit in the VM-entry interruption-information field is 1 and the interruption type in that field has value 2.

• Pending debug exceptions.
  — Bits 11:4, bit 13, bit 15, and bits 63:17 (bits 31:17 on processors that do not support Intel 64 architecture) must be 0.
  — The following checks are performed if any of the following holds: (1) the interruptibility-state field indicates blocking by STI (bit 0 in that field is 1); (2) the interruptibility-state field indicates blocking by MOV SS (bit 1 in that field is 1); or (3) the activity-state field indicates HLT:
    • Bit 14 (BS) must be 1 if the TF flag (bit 8) in the RFLAGS field is 1 and the BTF flag (bit 1) in the IA32_DEBUGCTL field is 0.
    • Bit 14 (BS) must be 0 if the TF flag (bit 8) in the RFLAGS field is 0 or the BTF flag (bit 1) in the IA32_DEBUGCTL field is 1.
  — The following checks are performed if bit 16 (RTM) is 1:
    • Bits 11:0, bits 15:13, and bits 63:17 (bits 31:17 on processors that do not support Intel 64 architecture) must be 0; bit 12 must be 1.
    • The processor must support for RTM by enumerating CPUID.(EAX=07H,ECX=0):EBX[bit 11] as 1.
    • The interruptibility-state field must not indicate blocking by MOV SS (bit 1 in that field must be 0).
• VMCS link pointer. The following checks apply if the field contains a value other than FFFFFFFF_FFFFFFFFH:
  — Bits 11:0 must be 0.
  — Bits beyond the processor’s physical-address width must be 0.1,2
  — The 4 bytes located in memory referenced by the value of the field (as a physical address) must satisfy the following:
    • Bits 30:0 must contain the processor’s VMCS revision identifier (see Section 25.2).3
    • Bit 31 must contain the setting of the “VMCS shadowing” VM-execution control.4 This implies that the referenced VMCS is a shadow VMCS (see Section 25.10) if and only if the "VMCS shadowing" VM-execution control is 1.

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. If IA32_VMX_BASIC[48] is read as 1, this field must not set any bits in the range 63:32; see Appendix A.1.
3. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field. For all processors produced prior to this change, bit 31 of the VMCS revision identifier was 0.
4. “VMCS shadowing” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “VMCS shadowing” VM-execution control were 0. See Section 25.6.2.
— If the processor is not in SMM or the “entry to SMM” VM-entry control is 1, the field must not contain the current VMCS pointer.

— If the processor is in SMM and the “entry to SMM” VM-entry control is 0, the field must differ from the executive-VMCS pointer.

### 27.3.1.6 Checks on Guest Page-Directory-Pointer-Table Entries

If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0, the logical processor uses **PAE paging** (see Section 4.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). When PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs.

A VM entry is to a guest that uses PAE paging if (1) bit 31 (corresponding to CR0.PG) is set in the CR0 field in the guest-state area; (2) bit 5 (corresponding to CR4.PAE) is set in the CR4 field; and (3) the “IA-32e mode guest” VM-entry control is 0. Such a VM entry checks the validity of the PDPTEs:

- If the “enable EPT” VM-execution control is 0, VM entry checks the validity of the PDPTEs referenced by the CR3 field in the guest-state area if either (1) PAE paging was not in use before the VM entry; or (2) the value of CR3 is changing as a result of the VM entry. VM entry may check their validity even if neither (1) nor (2) hold.

- If the “enable EPT” VM-execution control is 1, VM entry checks the validity of the PDPTE fields in the guest-state area (see Section 25.4.2).

A VM entry to a guest that does not use PAE paging does not check the validity of any PDPTEs.

A VM entry that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use. If MOV to CR3 would cause a general-protection exception due to the PDPTEs that would be loaded (e.g., because a reserved bit is set), the VM entry fails.

### 27.3.2 Loading Guest State

Processor state is updated on VM entries in the following ways:

- Some state is loaded from the guest-state area.
- Some state is determined by VM-entry controls.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order and in parallel with the checking of VMCS contents (see Section 27.3.1).

The loading of guest state is detailed in Section 27.3.2.1 to Section 27.3.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

In addition to the state loading described in this section, VM entries may load MSRs from the VM-entry MSR-load area (see Section 27.4). This loading occurs only after the state loading described in this section and the checking of VMCS contents described in Section 27.3.1.

#### 27.3.2.1 Loading Guest Control Registers, Debug Registers, and MSRs

The following items describe how guest control registers, debug registers, and MSRs are loaded on VM entry:

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine the number physical-address bits supported by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

2. “Enable EPT” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “enable EPT” VM-execution control were 0. See Section 25.6.2.

3. This implies that (1) bits 11:9 in each PDPT is ignored; and (2) if bit 0 (present) is clear in one of the PDPT, bits 63:1 of that PDPT are ignored.
• CR0 is loaded from the CR0 field with the exception of the following bits, which are never modified on VM entry: ET (bit 4); reserved bits 15:6, 17, and 28:19; NW (bit 29) and CD (bit 30). The values of these bits in the CR0 field are ignored.

• CR3 and CR4 are loaded from the CR3 field and the CR4 field, respectively.

• If the “load debug controls” VM-entry control is 1, DR7 is loaded from the DR7 field with the exception that bit 12 and bits 15:14 are always 0 and bit 10 is always 1. The values of these bits in the DR7 field are ignored.

The first processors to support the virtual-machine extensions supported only the 1-setting of the “load debug controls” VM-entry control and thus always loaded DR7 from the DR7 field.

• The following describes how certain MSRs are loaded using fields in the guest-state area:
  — If the “load debug controls” VM-entry control is 1, the IA32_DEBUGCTL MSR is loaded from the IA32_DEBUGCTL field. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always loaded the IA32_DEBUGCTL MSR from the IA32_DEBUGCTL field.
  — The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since this field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
  — The IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are loaded from the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field, respectively. On processors that do not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
  — The following are performed on processors that support Intel 64 architecture:
    • The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 27.3.2.2).
    • If the “load IA32_EFER” VM-entry control is 0, bits in the IA32_EFER MSR are modified as follows:
      — IA32_EFER.LMA is loaded with the setting of the “IA-32e mode guest” VM-entry control.
      — If CR0 is being loaded so that CR0.PG = 1, IA32_EFER.LME is also loaded with the setting of the “IA-32e mode guest” VM-entry control. Otherwise, IA32_EFER.LME is unmodified.

See below for the case in which the “load IA32_EFER” VM-entry control is 1
  — If the “load IA32_PERF_GLOBAL_CTRL” VM-entry control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field.
  — If the “load IA32_PAT” VM-entry control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field.
  — If the “load IA32_EFER” VM-entry control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field.
  — If the “load IA32_BNDCFGS” VM-entry control is 1, the IA32_BNDCFGS MSR is loaded from the IA32_BNDCFGS field.
  — If the “load IA32_RTIT_CTL” VM-entry control is 1, the IA32_RTIT_CTL MSR is loaded from the IA32_RTIT_CTL field.
  — If the “load CET” VM-entry control is 1, the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are loaded from the IA32_S_CET field and the IA32_INTERRUPT_SSP_TABLE_ADDR field, respectively. On processors that do not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
  — If the “load guest IA32_LBR_CTL” VM-entry control is 1, the IA32_LBR_CTL MSR is loaded from the IA32_LBR_CTL guest state field.
  — If the “load PKRS” VM-entry control is 1, the IA32_PKRS MSR is loaded from the IA32_PKRS field.
  — If the “load UINV” VM-entry control is 1, UINV is loaded with the low 8 bits of the UINV field. UINV is represented in bits 39:32 of the IA32_UINTR_MISC MSR. The remainder of the MSR is not modified.

1. Bits 15:6, bit 17, and bit 28:19 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. Bits 15:6, bit 17, and bit 28:19 of CR0 are always 0 and CR0.ET is always 1.

2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, VM entry must be loading CR0 so that CR0.PG = 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-entry MSR-load area. See Section 27.4.

- The SMBASE register is unmodified by all VM entries except those that return from SMM.

### 27.3.2.2 Loading Guest Segment Registers and Descriptor-Table Registers

For each of CS, SS, DS, ES, FS, GS, TR, and LDTR, fields are loaded from the guest-state area as follows:

- The unusable bit is loaded from the access-rights field. This bit can never be set for TR (see Section 27.3.1.2). If it is set for one of the other registers, the following apply:
  - For each of CS, SS, DS, ES, FS, and GS, uses of the segment cause faults (general-protection exception or stack-fault exception) outside 64-bit mode, just as they would had the segment been loaded using a null selector. This bit does not cause accesses to fault in 64-bit mode.
  - If this bit is set for LDTR, uses of LDTR cause general-protection exceptions in all modes, just as they would had LDTR been loaded using a null selector.

If this bit is clear for any of CS, SS, DS, ES, FS, GS, TR, and LDTR, a null selector value does not cause a fault (general-protection exception or stack-fault exception).

- TR. The selector, base, limit, and access-rights fields are loaded.
- CS.
  - The following fields are always loaded: selector, base address, limit, and (from the access-rights field) the L, D, and G bits.
  - For the other fields, the unusable bit of the access-rights field is consulted:
    - If the unusable bit is 0, all of the access-rights field is loaded.
    - If the unusable bit is 1, the remainder of CS access rights are undefined after VM entry.
- SS, DS, ES, FS, GS, and LDTR.
  - The selector fields are loaded.
  - For the other fields, the unusable bit of the corresponding access-rights field is consulted:
    - If the unusable bit is 0, the base-address, limit, and access-rights fields are loaded.
    - If the unusable bit is 1, the base address, the segment limit, and the remainder of the access rights are undefined after VM entry with the following exceptions:
      - Bits 3:0 of the base address for SS are cleared to 0.
      - SS.DPL is always loaded from the SS access-rights field. This will be the current privilege level (CPL) after the VM entry completes.
      - SS.B is always set to 1.
      - The base addresses for FS and GS are loaded from the corresponding fields in the VMCS. On processors that support Intel 64 architecture, the values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
      - On processors that support Intel 64 architecture, the base address for LDTR is set to an undefined but canonical value.
      - On processors that support Intel 64 architecture, bits 63:32 of the base addresses for SS, DS, and ES are cleared to 0.

GDTR and IDTR are loaded using the base and limit fields.

### 27.3.2.3 Loading Guest RIP, RSP, RFLAGS, and SSP

RSP, RIP, and RFLAGS are loaded from the RSP field, the RIP field, and the RFLAGS field, respectively. If the "load CET" VM-entry control is 1, SSP (shadow-stack pointer) is loaded from the SSP field.

The following items regard the upper 32 bits of these fields on VM entries that are not to 64-bit mode:
• Bits 63:32 of RSP are undefined outside 64-bit mode. Thus, a logical processor may ignore the contents of bits 63:32 of the RSP field on VM entries that are not to 64-bit mode.
• As noted in Section 27.3.1.4, bits 63:32 of the RIP and RFLAGS fields must be 0 on VM entries that are not to 64-bit mode. (The same is true for SSP for VM entries that are not to 64-bit mode when the “load CET” VM-entry control is 1.)

27.3.2.4 Loading Page-Directory-Pointer-Table Entries

As noted in Section 27.3.1.6, the logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0. A VM entry to a guest that uses PAE paging loads the PDPTEs into internal, non-architectural registers based on the setting of the “enable EPT” VM-execution control:
• If the control is 0, the PDPTEs are loaded from the page-directory-pointer table referenced by the physical address in the value of CR3 being loaded by the VM entry (see Section 27.3.2.1). The values loaded are treated as physical addresses in VMX non-root operation.
• If the control is 1, the PDPTEs are loaded from corresponding fields in the guest-state area (see Section 25.4.2). The values loaded are treated as guest-physical addresses in VMX non-root operation.

27.3.2.5 Updating Non-Register State

Section 29.4 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM entries invalidate cached mappings:
• If the “enable VPID” VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP).
• VM entries are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the “enable VPID” VM-execution control is 1.

If the “virtual-interrupt delivery” VM-execution control is 1, VM entry loads the values of RVI and SVI from the guest interrupt-status field in the VMCS (see Section 25.4.2). After doing so, the logical processor first causes PPR virtualization (Section 30.1.3) and then evaluates pending virtual interrupts (Section 30.2.1). If a virtual interrupt is recognized, it may be delivered in VMX non-root operation immediately after VM entry (including any specified event injection) completes; see Section 27.7.5. See Section 30.2.2 for details regarding the delivery of virtual interrupts.

27.3.3 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 9.10.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. VM entries clear any address-range monitoring that may be in effect.

27.4 LOADING MSRS

VM entries may load MSRs from the VM-entry MSR-load area (see Section 25.8.2). Specifically each entry in that area (up to the number specified in the VM-entry MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.1

Processing of an entry fails in any of the following cases:
• The value of bits 31:0 is either C0000100H (the IA32_FS_BASE MSR) or C0000101 (the IA32_GS_BASE MSR).
• The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.

---

1. Because attempts to modify the value of IA32_EFER.LMA by WRMSR are ignored, attempts to modify it using the VM-entry MSR-load area are also ignored.
• The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM entry did not commence in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
• The value of bits 31:0 indicates an MSR that cannot be loaded on VM entries for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
• Bits 63:32 are not all 0.
• An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.\(^1\)

The VM entry fails if processing fails for any entry. The logical processor responds to such failures by loading state from the host-state area, as it would for a VM exit. See Section 27.8.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM entry, the logical processor will not use any translations that were cached before the transition.

### 27.5 TRACE-ADDRESS PRE-TRANSLATION (TAPT)

When the “Intel PT uses guest physical addresses” VM-execution control is 1, the addresses used by Intel PT are treated as guest-physical addresses, and these are translated to physical addresses using EPT.

VM entry uses **trace-address pre-translation (TAPT)** to prevent buffered trace data from being lost due to an EPT violation; see Section 26.5.4.2. VM entry uses TAPT only if Intel PT will be enabled following VM entry (IA32_RTIT_CTL.TraceEn = 1) and only if the “Intel PT uses guest physical addresses” VM-execution control is 1.

As noted in Section 26.5.4, TAPT may cause a VM exit due to an EPT violation, EPT misconfiguration, page-modification log-full event, or APIC access. If such a VM exit occurs as a result of TAPT during VM entry, the VM exit operates as if it had occurred in VMX non-root operation after the VM entry completed (in the guest context).

If TAPT during VM entry causes a VM exit, the VM entry does not perform event injection (Section 27.6), even if the valid bit in the VM-entry interruption-information field is 1. Such VM exits save the contents of VM-entry interruption-information and VM-entry exception error code fields into the IDT-vectoring information and IDT-vectoring error code fields, respectively.

### 27.6 EVENT INJECTION

If the valid bit in the VM-entry interruption-information field (see Section 25.8.3) is 1, VM entry causes an event to be delivered (or made pending) after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.

• If the interruption type in the field is 0 (external interrupt), 2 (non-maskable interrupt); 3 (hardware exception), 4 (software interrupt), 5 (privileged software exception), or 6 (software exception), the event is delivered as described in Section 27.6.1.
• If the interruption type in the field is 7 (other event) and the vector field is 0, an MTF VM exit is pending after VM entry. See Section 27.6.2.

#### 27.6.1 Vectored-Event Injection

VM entry delivers an injected vectored event within the guest context established by VM entry. This means that delivery occurs after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.\(^2\) The event is delivered using the vector in that field to select a descriptor in the IDT. Since event injection occurs after loading IDTR from the guest-state area, this is the guest IDT.

---

1. If CR0.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. If VM entry has established CR0.PG = 1, the IA32_EFER MSR should not be included in the VM-entry MSR-load area for the purpose of modifying the LME bit.

2. If the valid bit in the VM-entry interruption-information field is 1, VM entry causes an event to be delivered (or made pending) after all components of guest state have been loaded (including MSRs) and after the VM-execution control fields have been established.
Section 27.6.1.1 provides details of vectored-event injection. In general, the event is delivered exactly as if it had been generated normally.

An exception is made if the following all hold: bit 25 (UINTR) is set to 1 in the guest CR4 field and the “IA-32e mode guest” VM-entry control is 1, and VM entry is modified if it is injecting an external interrupt whose vector is the value that UINV would have after VM entry. In this case, the logical processor then performs user-interrupt notification processing as specified in Section 7.5.2 instead of the process described in Section 27.6.1.1. (If the guest activity-state field indicated the HLT state, the logical processor enters the HLT state following user-interrupt notification processing.)

If event delivery (or user-interrupt notification processing; see above) encounters a nested exception (for example, a general-protection exception because the vector indicates a descriptor beyond the IDT limit), the exception bitmap is consulted using the vector of that exception:

- If the bit for the nested exception is 0, the nested exception is delivered normally. If the nested exception is benign, it is delivered through the IDT. If it is contributory or a page fault, a double fault may be generated, depending on the nature of the event whose delivery encountered the nested exception. See Chapter 6, “Interrupt 8—Double Fault Exception (#DF)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
- If the bit for the nested exception is 1, a VM exit occurs. Section 27.6.1.2 details cases in which event injection causes a VM exit.

### 27.6.1.1 Details of Vectored-Event Injection

The event-injection process is controlled by the contents of the VM-entry interruption information field (format given in Table 25-17), the VM-entry exception error-code field, and the VM-entry instruction-length field. The following items provide details of the process:

- The value pushed on the stack for RFLAGS is generally that which was loaded from the guest-state area. The value pushed for the RF flag is not modified based on the type of event being delivered. However, the pushed value of RFLAGS may be modified if a software interrupt is being injected into a guest that will be in virtual-8086 mode (see below). After RFLAGS is pushed on the stack, the value in the RFLAGS register is modified as is done normally when delivering an event through the IDT.
- The instruction pointer that is pushed on the stack depends on the type of event and whether nested exceptions occur during its delivery. The term current guest RIP refers to the value to be loaded from the guest-state area. The value pushed is determined as follows:
  - If VM entry successfully injects (with no nested exception) an event with interruption type external interrupt, NMI, or hardware exception, the current guest RIP is pushed on the stack.
  - If VM entry successfully injects (with no nested exception) an event with interruption type software interrupt, privileged software exception, or software exception, the current guest RIP is incremented by the VM-entry instruction length before being pushed on the stack.
  - If VM entry encounters an exception while injecting an event and that exception does not cause a VM exit, the current guest RIP is pushed on the stack regardless of event type or VM-entry instruction length. If the encountered exception does cause a VM exit that saves RIP, the saved RIP is current guest RIP.
- If the deliver-error-code bit (bit 11) is set in the VM-entry interruption-information field, the contents of the VM-entry exception error-code field is pushed on the stack as an error code would be pushed during delivery of an exception.
- DR6, DR7, and the IA32_DEBUGCTL MSR are not modified by event injection, even if the event has vector 1 (normal deliveries of debug exceptions, which have vector 1, do update these registers).

---

1. Hardware exceptions with the following unused vectors are considered benign: 15 and 21–31. A hardware exception with vector 20 is considered benign unless the processor supports the 1-setting of the “EPT-violation #VE” VM-execution control; in that case, it has the same severity as page faults.
2. While these items refer to RIP, the width of the value pushed (16 bits, 32 bits, or 64 bits) is determined normally.
2. This does not imply that injection of an exception or interrupt will cause a VM exit due to the settings of VM-execution control fields (such as the exception bitmap) that would cause a VM exit if the event had occurred in VMX non-root operation. In contrast, a nested exception encountered during event delivery may cause a VM exit; see Section 27.6.1.1.
• If VM entry is injecting a software interrupt and the guest will be in virtual-8086 mode (RFLAGS.VM = 1), no general-protection exception can occur due to RFLAGS.IOPL < 3. A VM monitor should check RFLAGS.IOPL before injecting such an event and, if desired, inject a general-protection exception instead of a software interrupt.

• If VM entry is injecting a software interrupt and the guest will be in virtual-8086 mode with virtual-8086 mode extensions (RFLAGS.VM = CR4.VME = 1), event delivery is subject to VME-based interrupt redirection based on the software interrupt redirection bitmap in the task-state segment (TSS) as follows:
  — If bit \( n \) in the bitmap is clear (where \( n \) is the number of the software interrupt), the interrupt is directed to an 8086 program interrupt handler: the processor uses a 16-bit interrupt-vector table (IVT) located at linear address zero. If the value of RFLAGS.IOPL is less than 3, the following modifications are made to the value of RFLAGS that is pushed on the stack: IOPL is set to 3, and IF is set to the value of VIF.
  — If bit \( n \) in the bitmap is set (where \( n \) is the number of the software interrupt), the interrupt is directed to a protected-mode interrupt handler. (In other words, the injection is treated as described in the next item.) In this case, the software interrupt does not invoke such a handler if RFLAGS.IOPL < 3 (a general-protection exception occurs instead). However, as noted above, RFLAGS.IOPL cannot cause an injected software interrupt to cause such an exception. Thus, in this case, the injection invokes a protected-mode interrupt handler independent of the value of RFLAGS.IOPL.

Injection of events of other types are not subject to this redirection.

• If VM entry is injecting a non-maskable interrupt (NMI) and the “virtual NMIs” VM-execution control is 1, virtual-NMI blocking is in effect after VM entry.

• The transition causes a last-branch record to be logged if the LBR bit is set in the IA32_DEBUGCTL MSR. This is true even for events such as debug exceptions, which normally clear the LBR bit before delivery.

• The last-exception record MSRs (LERs) may be updated based on the setting of the LBR bit in the IA32_DEBUGCTL MSR. Events such as debug exceptions, which normally clear the LBR bit before they are delivered, and therefore do not normally update the LERs, may do so as part of VM-entry event injection.

• If injection of an event encounters a nested exception, the value of the EXT bit (bit 0) in any error code for that nested exception is determined as follows:
  — If event being injected has interruption type external interrupt, NMI, hardware exception, or privileged software exception and encounters a nested exception (but does not produce a double fault), the error code for that exception sets the EXT bit.
  — If event being injected is a software interrupt or a software exception and encounters a nested exception, the error code for that exception clears the EXT bit.
  — If event delivery encounters a nested exception and delivery of that exception encounters another exception (but does not produce a double fault), the error code for that exception sets the EXT bit.
  — If a double fault is produced, the error code for the double fault is 0000H (the EXT bit is clear).

27.6.1.2 VM Exits During Event Injection

An event being injected never causes a VM exit directly regardless of the settings of the VM-execution controls. For example, setting the “NMI exiting” VM-execution control to 1 does not cause a VM exit due to injection of an NMI. However, the event-delivery process may lead to a VM exit:

• If the vector in the VM-entry interruption-information field identifies a task gate in the IDT, the attempted task switch may cause a VM exit just as it would had the injected event occurred during normal execution in VMX non-root operation (see Section 26.4.2).

• If event delivery encounters a nested exception, a VM exit may occur depending on the contents of the exception bitmap (see Section 26.2).
• If event delivery generates a double-fault exception (due to a nested exception); the logical processor encounters another nested exception while attempting to call the double-fault handler; and that exception does not cause a VM exit due to the exception bitmap; then a VM exit occurs due to triple fault (see Section 26.2).

• If event delivery injects a double-fault exception and encounters a nested exception that does not cause a VM exit due to the exception bitmap, then a VM exit occurs due to triple fault (see Section 26.2).

• If the “virtualize APIC accesses” VM-execution control is 1 and event delivery generates an access to the APIC-access page, that access is treated as described in Section 30.4 and may cause a VM exit.¹

If the event-delivery process does cause a VM exit, the processor state before the VM exit is determined just as it would be had the injected event occurred during normal execution in VMX non-root operation. If the injected event directly accesses a task gate that cause a VM exit or if the first nested exception encountered causes a VM exit, information about the injected event is saved in the IDT-vectoring information field (see Section 28.2.4).

The material in this section applies also if injection of an external interrupt results in user-interrupt notification processing instead of event delivery (see Section 27.6.1 earlier).

### 27.6.1.3 Event Injection for VM Entries to Real-Address Mode

If VM entry is loading CR0.PE with 0, any injected vectored event is delivered as would normally be done in real-address mode.² Specifically, VM entry uses the vector provided in the VM-entry interruption-information field to select a 4-byte entry from an interrupt-vector table at the linear address in IDTR.base. Further details are provided in Section 15.1.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Because bit 11 (deliver error code) in the VM-entry interruption-information field must be 0 if CR0.PE will be 0 after VM entry (see Section 27.2.1.3), vectored events injected with CR0.PE = 0 do not push an error code on the stack. This is consistent with event delivery in real-address mode.

If event delivery encounters a fault (due to a violation of IDTR.limit or of SS.limit), the fault is treated as if it had occurred during event delivery in VMX non-root operation. Such a fault may lead to a VM exit as discussed in Section 27.6.1.2.

### 27.6.2 Injection of Pending MTF VM Exits

If the interruption type in the VM-entry interruption-information field is 7 (other event) and the vector field is 0, VM entry causes an MTF VM exit to be pending on the instruction boundary following VM entry. This is the case even if the “monitor trap flag” VM-execution control is 0. See Section 26.5.2 for the treatment of pending MTF VM exits.

### 27.7 SPECIAL FEATURES OF VM ENTRY

This section details a variety of features of VM entry. It uses the following terminology: a VM entry is **vectoring** if the valid bit (bit 31) of the VM-entry interruption information field is 1 and the interruption type in the field is 0 (external interrupt), 2 (non-maskable interrupt); 3 (hardware exception), 4 (software interrupt), 5 (privileged software exception), or 6 (software exception).

#### 27.7.1 Interruptibility State

The interruptibility-state field in the guest-state area (see Table 25-3) contains bits that control blocking by STI, blocking by MOV SS, and blocking by NMI. This field impacts event blocking after VM entry as follows:

1. “Virtualize APIC accesses” is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the “virtualize APIC accesses” VM-execution control were 0. See Section 25.6.2.

2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, VM entry must be loading CR0.PE with 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
• If the VM entry is vectoring, there is no blocking by STI or by MOV SS following the VM entry, regardless of the contents of the interruptibility-state field.

• If the VM entry is not vectoring, the following apply:
  — Events are blocked by STI if and only if bit 0 in the interruptibility-state field is 1. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry; see Section 27.7.3).
  — Events are blocked by MOV SS if and only if bit 1 in the interruptibility-state field is 1. This may affect the treatment of pending debug exceptions; see Section 27.7.3. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry).

• The blocking of non-maskable interrupts (NMIs) is determined as follows:
  — If the "virtual NMIs" VM-execution control is 0, NMIs are blocked if and only if bit 3 (blocking by NMI) in the interruptibility-state field is 1. If the "NMI exiting" VM-execution control is 0, execution of the IRET instruction removes this blocking (even if the instruction generates a fault). If the "NMI exiting" control is 1, IRET does not affect this blocking.
  — The following items describe the use of bit 3 (blocking by NMI) in the interruptibility-state field if the "virtual NMIs" VM-execution control is 1:
    • The bit’s value does not affect the blocking of NMIs after VM entry. NMIs are not blocked in VMX non-root operation (except for ordinary blocking for other reasons, such as by the MOV SS instruction, the wait-for-SIPI state, etc.)
    • The bit’s value determines whether there is virtual-NMI blocking after VM entry. If the bit is 1, virtual-NMI blocking is in effect after VM entry. If the bit is 0, there is no virtual-NMI blocking after VM entry unless the VM entry is injecting an NMI (see Section 27.6.1.1). Execution of IRET removes virtual-NMI blocking (even if the instruction generates a fault).

  If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" control must be 0; see Section 27.2.1.1.

• Blocking of system-management interrupts (SMIs) is determined as follows:
  — If the VM entry was not executed in system-management mode (SMM), SMI blocking is unchanged by VM entry.
  — If the VM entry was executed in SMM, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.

27.7.2 Activity State

The activity-state field in the guest-state area controls whether, after VM entry, the logical processor is active or in one of the inactive states identified in Section 25.4.2. The use of this field is determined as follows:

• If the VM entry is vectoring, the logical processor is in the active state after VM entry. While the consistency checks described in Section 27.3.1.5 on the activity-state field do apply in this case, the contents of the activity-state field do not determine the activity state after VM entry.

• If the VM entry is not vectoring, the logical processor ends VM entry in the activity state specified in the guest-state area. If VM entry ends with the logical processor in an inactive activity state, the VM entry generates any special bus cycle that is normally generated when that activity state is entered from the active state. If VM entry would end with the logical processor in the shutdown state and the logical processor is in SMX operation,\(^1\) an Intel® TXT shutdown condition occurs. The error code used is 0000H, indicating "legacy shutdown." See the Intel® Trusted Execution Technology Preliminary Architecture Specification.

• Some activity states unconditionally block certain events. The following blocking is in effect after any VM entry that puts the processor in the indicated state:
  — The active state blocks start-up IPIs (SIPIs). SIPIs that arrive while a logical processor is in the active state and in VMX non-root operation are discarded and do not cause VM exits.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. See Chapter 6, “Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B.
— The HLT state blocks start-up IPIs (SIPIs). SIPIs that arrive while a logical processor is in the HLT state and in VMX non-root operation are discarded and do not cause VM exits.

— The shutdown state blocks external interrupts and SIPIs. External interrupts that arrive while a logical processor is in the shutdown state and in VMX non-root operation do not cause VM exits even if the “external-interrupt exiting” VM-execution control is 1. SIPIs that arrive while a logical processor is in the shutdown state and in VMX non-root operation are discarded and do not cause VM exits.

— The wait-for-SIPI state blocks external interrupts, non-maskable interrupts (NMIs), INIT signals, and system-management interrupts (SMIs). Such events do not cause VM exits if they arrive while a logical processor is in the wait-for-SIPI state and in VMX non-root operation.

### 27.7.3 Delivery of Pending Debug Exceptions after VM Entry

The pending debug exceptions field in the guest-state area indicates whether there are debug exceptions that have not yet been delivered (see Section 25.4.2). This section describes how these are treated on VM entry.

There are no pending debug exceptions after VM entry if any of the following are true:

• The VM entry is vectoring with one of the following interruption types: external interrupt, non-maskable interrupt (NMI), hardware exception, or privileged software exception.

• The interruptibility-state field does not indicate blocking by MOV SS and the VM entry is vectoring with either of the following interruption type: software interrupt or software exception.

• The VM entry is not vectoring and the activity-state field indicates either shutdown or wait-for-SIPI.

If none of the above hold, the pending debug exceptions field specifies the debug exceptions that are pending for the guest. There are valid pending debug exceptions if either the BS bit (bit 14) or the enable-breakpoint bit (bit 12) is 1. If there are valid pending debug exceptions, they are handled as follows:

• If the VM entry is not vectoring, the pending debug exceptions are treated as they would had they been encountered normally in guest execution:
  — If the logical processor is not blocking such exceptions (the interruptibility-state field indicates no blocking by MOV SS), a debug exception is delivered after VM entry (see below).
  — If the logical processor is blocking such exceptions (due to blocking by MOV SS), the pending debug exceptions are held pending or lost as would normally be the case.

• If the VM entry is vectoring (with interruption type software interrupt or software exception and with blocking by MOV SS), the following items apply:
  — For injection of a software interrupt or of a software exception with vector 3 (#BP) or vector 4 (#OF) — or a privileged software exception with vector 1 (#DB) — the pending debug exceptions are treated as they would had they been encountered normally in guest execution if the corresponding instruction (INT1, INT3, or INTO) were executed after a MOV SS that encountered a debug trap.
  — For injection of a software exception with a vector other than 3 and 4, the pending debug exceptions may be lost or they may be delivered after injection (see below).

If there are no valid pending debug exceptions (as defined above), no pending debug exceptions are delivered after VM entry.

If a pending debug exception is delivered after VM entry, it has the priority of “traps on the previous instruction” (see Section 6.9 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). Thus, INIT signals and system-management interrupts (SMIs) take priority of such an exception, as do VM exits induced by the TPR threshold (see Section 27.7.7) and pending MTF VM exits (see Section 27.7.8. The exception takes priority over any pending non-maskable interrupt (NMI) or external interrupt and also over VM exits due to the 1-settings of the “interrupt-window exiting” and “NMI-window exiting” VM-execution controls.

A pending debug exception delivered after VM entry causes a VM exit if the bit 1 (#DB) is 1 in the exception bitmap. If it does not cause a VM exit, it updates DR6 normally.
27.7.4 VMX-Preemption Timer

If the "activate VMX-preemption timer" VM-execution control is 1, VM entry starts the VMX-preemption timer with the unsigned value in the VMX-preemption timer-value field.

It is possible for the VMX-preemption timer to expire during VM entry (e.g., if the value in the VMX-preemption timer-value field is zero). If this happens (and if the VM entry was not to the wait-for-SIPI state), a VM exit occurs with its normal priority after any event injection and before execution of any instruction following VM entry. For example, any pending debug exceptions established by VM entry (see Section 27.7.3) take priority over a timer-induced VM exit. (The timer-induced VM exit will occur after delivery of the debug exception, unless that exception or its delivery causes a different VM exit.)

See Section 26.5.1 for details of the operation of the VMX-preemption timer in VMX non-root operation, including the blocking and priority of the VM exits that it causes.

27.7.5 Interrupt-Window Exiting and Virtual-Interrupt Delivery

If "interrupt-window exiting" VM-execution control is 1, an open interrupt window may cause a VM exit immediately after VM entry (see Section 26.2 for details). If the "interrupt-window exiting" VM-execution control is 0 but the "virtual-interrupt delivery" VM-execution control is 1, a virtual interrupt may be delivered immediately after VM entry (see Section 27.3.2.5 and Section 30.2.1).

The following items detail the treatment of these events:

- These events occur after any event injection specified for VM entry.
- Non-maskable interrupts (NMIs) and higher priority events take priority over these events. These events take priority over external interrupts and lower priority events.
- These events wake the logical processor if it just entered the HLT state because of a VM entry (see Section 27.7.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state.

27.7.6 NMI-Window Exiting

The "NMI-window exiting" VM-execution control may cause a VM exit to occur immediately after VM entry (see Section 26.2 for details).

The following items detail the treatment of these VM exits:

- These VM exits follow event injection if such injection is specified for VM entry.
- Debug-trap exceptions (see Section 27.7.3) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.
- VM exits caused by this control wake the logical processor if it just entered either the HLT state or the shutdown state because of a VM entry (see Section 27.7.2). They do not occur if the logical processor just entered the wait-for-SIPI state.

27.7.7 VM Exits Induced by the TPR Threshold

If the "use TPR shadow" and "virtualize APIC accesses" VM-execution controls are both 1 and the "virtual-interrupt delivery" VM-execution control is 0, a VM exit occurs immediately after VM entry if the value of bits 3:0 of the TPR threshold VM-execution control field is greater than the value of bits 7:4 of VTPR (see Section 30.1.1).\(^1\)

The following items detail the treatment of these VM exits:

- The VM exits are not blocked if RFLAGS.IF = 0 or by the setting of bits in the interruptibility-state field in guest-state area.
- The VM exits follow event injection if such injection is specified for VM entry.

---

1. "Virtualize APIC accesses" and "virtual-interrupt delivery" are secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if these controls were 0. See Section 25.6.2.
• VM exits caused by this control take priority over system-management interrupts (SMIs), INIT signals, and lower priority events. They thus have priority over the VM exits described in Section 27.7.5, Section 27.7.6, and Section 27.7.8, as well as any interrupts or debug exceptions that may be pending at the time of VM entry.

• These VM exits wake the logical processor if it just entered the HLT state as part of a VM entry (see Section 27.7.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state. If such a VM exit is suppressed because the processor just entered the shutdown state, it occurs after the delivery of any event that cause the logical processor to leave the shutdown state while remaining in VMX non-root operation (e.g., due to an NMI that occurs while the "NMI-exiting" VM-execution control is 0).

• The basic exit reason is "TPR below threshold."

27.7.8 Pending MTF VM Exits
As noted in Section 27.6.2, VM entry may cause an MTF VM exit to be pending immediately after VM entry. The following items detail the treatment of these VM exits:

• System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these VM exits. These VM exits take priority over debug-trap exceptions and lower priority events.

• These VM exits wake the logical processor if it just entered the HLT state because of a VM entry (see Section 27.7.2). They do not occur if the logical processor just entered the shutdown state or the wait-for-SIPI state.

27.7.9 VM Entries and Advanced Debugging Features
VM entries are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

27.7.10 User-Interrupt Recognition After VM Entry
A VM entry results in recognition of a pending user interrupt if it completes with CR4.UINTR = IA32_EFER.LMA = 1 and with UIRR ≠ 0; otherwise, no pending user interrupt is recognized.

27.8 VM-ENTRY FAILURES DURING OR AFTER LOADING GUEST STATE
VM-entry failures due to the checks identified in Section 27.3.1 and failures during the MSR loading identified in Section 27.4 are treated differently from those that occur earlier in VM entry. In these cases, the following steps take place:

1. Information about the VM-entry failure is recorded in the VM-exit information fields:
   — Exit reason.
     • Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM-entry failure. The following numbers are used:
       33. VM-entry failure due to invalid guest state. A VM entry failed one of the checks identified in Section 27.3.1.
       34. VM-entry failure due to MSR loading. A VM entry failed in an attempt to load MSRs (see Section 27.4).
       41. VM-entry failure due to machine-check event. A machine-check event occurred during VM entry (see Section 27.9).
     • Bit 31 is set to 1 to indicate a VM-entry failure.
     • The remainder of the field (bits 30:16) is cleared.
— Exit qualification. This field is set based on the exit reason.

• VM-entry failure due to invalid guest state. In most cases, the exit qualification is cleared to 0. The following non-zero values are used in the cases indicated:
  1. Not used.
  2. Failure was due to a problem loading the PDPTEs (see Section 27.3.1.6).
  3. Failure was due to an attempt to inject a non-maskable interrupt (NMI) into a guest that is blocking events through the STI blocking bit in the interruptibility-state field.
  4. Failure was due to an invalid VMCS link pointer (see Section 27.3.1.5).

VM-entry checks on guest-state fields may be performed in any order. Thus, an indication by exit qualification of one cause does not imply that there are not also other errors. Different processors may give different exit qualifications for the same VMCS.

• VM-entry failure due to MSR loading. The exit qualification is loaded to indicate which entry in the VM-entry MSR-load area caused the problem (1 for the first entry, 2 for the second, etc.).

— All other VM-exit information fields are unmodified.

2. Processor state is loaded as would be done on a VM exit (see Section 28.5). If this results in \[\text{CR4.PAE} \& \text{CR0.PG} \& \sim \text{IA32_EFER.LMA} = 1\], page-directory-pointer-table entries (PDPTEs) may be checked and loaded (see Section 28.5.4).

3. The state of blocking by NMI is what it was before VM entry.

4. MSRs are loaded as specified in the VM-exit MSR-load area (see Section 28.6).

Although this process resembles that of a VM exit, many steps taken during a VM exit do not occur for these VM-entry failures:

• Most VM-exit information fields are not updated (see step 1 above).
• The valid bit in the VM-entry interruption-information field is not cleared.
• The guest-state area is not modified.
• No MSRs are saved into the VM-exit MSR-store area.

27.9 MACHINE-CHECK EVENTS DURING VM ENTRY

If a machine-check event occurs during a VM entry, one of the following occurs:

• The machine-check event is handled as if it occurred before the VM entry:
  — If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    • If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating "unrecoverable machine-check condition."
    • If the logical processor is outside SMX operation, it goes to the shutdown state.
  — If CR4.MCE = 1, a machine-check exception (#MC) is delivered through the IDT.

• The machine-check event is handled after VM entry completes:
  — If the VM entry ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    • If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
    • If the logical processor is outside SMX operation, it goes to the shutdown state.

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 7, “Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.
VM ENTRIES

If the VM entry ends with CR4.MCE = 1, a machine-check exception (#MC) is generated:

- If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
- If bit 18 of the exception bitmap is 1, the exception causes a VM exit.

A VM-entry failure occurs as described in Section 27.8. The basic exit reason is 41, for "VM-entry failure due to machine-check event."

The first option is not used if the machine-check event occurs after any guest state has been loaded. The second option is used only if VM entry is able to load all guest state.
VM exits occur in response to certain instructions and events in VMX non-root operation as detailed in Section 26.1 through Section 26.2. VM exits perform the following operations:

1. Information about the cause of the VM exit is recorded in the VM-exit information fields and VM-entry control fields are modified as described in Section 28.2.
2. Processor state is saved in the guest-state area (Section 28.3).
3. MSRs may be saved in the VM-exit MSR-store area (Section 28.4). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.
4. The following may be performed in parallel and in any order (Section 28.5):
   - Processor state is loaded based in part on the host-state area and some VM-exit controls. This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM. See Section 32.15.6 for information on how processor state is loaded by such VM exits.
   - Address-range monitoring is cleared.
5. MSRs may be loaded from the VM-exit MSR-load area (Section 28.6). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.

VM exits are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

Section 28.1 clarifies the nature of the architectural state before a VM exit begins. The steps described above are detailed in Section 28.2 through Section 28.6.

Section 32.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, ordinary transitions to SMM are replaced by VM exits to a separate SMM monitor. Called SMM VM exits, these are caused by the arrival of an SMI or the execution of VMCALL in VMX root operation. SMM VM exits differ from other VM exits in ways that are detailed in Section 32.15.2.

### 28.1 ARCHITECTURAL STATE BEFORE A VM EXIT

This section describes the architectural state that exists before a VM exit, especially for VM exits caused by events that would normally be delivered through the IDT. Note the following:

- An exception causes a VM exit **directly** if the bit corresponding to that exception is set in the exception bitmap. A non-maskable interrupt (NMI) causes a VM exit directly if the “NMI exiting” VM-execution control is 1. An external interrupt causes a VM exit directly if the “external-interrupt exiting” VM-execution control is 1. A start-up IPI (SIPI) that arrives while a logical processor is in the wait-for-SIPI activity state causes a VM exit directly. INIT signals that arrive while the processor is not in the wait-for-SIPI activity state cause VM exits directly.

- An exception, NMI, external interrupt, or software interrupt causes a VM exit **indirectly** if it does not do so directly but delivery of the event causes a nested exception, double fault, task switch, APIC access (see Section 30.4), EPT violation, EPT misconfiguration, page-modification log-full event (see Section 29.3.6), or SPP-related event (see Section 29.3.4) that causes a VM exit.

- An event **results** in a VM exit if it causes a VM exit (directly or indirectly).

The following bullets detail when architectural state is and is not updated in response to VM exits:

- If an event causes a VM exit directly, it does not update architectural state as it would have if it had it not caused the VM exit:
  - A debug exception does not update DR6, DR7, or IA32_DEBUGCTL. (Information about the nature of the debug exception is saved in the exit qualification field.)
  - A page fault does not update CR2. (The linear address causing the page fault is saved in the exit-qualification field.)
VM EXITS

— An NMI causes subsequent NMIs to be blocked, but only after the VM exit completes.
— An external interrupt does not acknowledge the interrupt controller and the interrupt remains pending, unless the "acknowledge interrupt on exit" VM-exit control is 1. In such a case, the interrupt controller is acknowledged and the interrupt is no longer pending.
— The flags L0 – L3 in DR7 (bit 0, bit 2, bit 4, and bit 6) are not cleared when a task switch causes a VM exit.
— If a task switch causes a VM exit, none of the following are modified by the task switch: old task-state segment (TSS); new TSS; old TSS descriptor; new TSS descriptor; RFLAGS.NT1; or the TR register.
— No last-exception record is made if the event that would do so directly causes a VM exit.
— If a machine-check exception causes a VM exit directly, this does not prevent machine-check MSRs from being updated. These are updated by the machine-check event itself and not the resulting machine-check exception.
— If the logical processor is in an inactive state (see Section 25.4.2) and not executing instructions, some events may be blocked but others may return the logical processor to the active state. Unblocked events may cause VM exits. If an unblocked event causes a VM exit directly, a return to the active state occurs only after the VM exit completes. The VM exit generates any special bus cycle that is normally generated when the active state is entered from that activity state.

MTF VM exits (see Section 26.5.2 and Section 27.7.8) are not blocked in the HLT activity state. If an MTF VM exit occurs in the HLT activity state, the logical processor returns to the active state only after the VM exit completes. MTF VM exits are blocked the shutdown state and the wait-for-SIPI state.

• If an event causes a VM exit indirectly, the event does update architectural state:
— A debug exception updates DR6, DR7, and the IA32_DEBUGCTL MSR. No debug exceptions are considered pending.
— A page fault updates CR2.
— An NMI causes subsequent NMIs to be blocked before the VM exit commences.
— An external interrupt acknowledges the interrupt controller and the interrupt is no longer pending.
— If the logical processor had been in an inactive state, it enters the active state and, before the VM exit commences, generates any special bus cycle that is normally generated when the active state is entered from that activity state.
— There is no blocking by STI or by MOV SS when the VM exit commences.
— Processor state that is normally updated as part of delivery through the IDT (CS, RIP, SS, RSP, RFLAGS) is not modified. However, the incomplete delivery of the event may write to the stack.
— The treatment of last-exception records is implementation dependent:
  • Some processors make a last-exception record when beginning the delivery of an event through the IDT (before it can encounter a nested exception). Such processors perform this update even if the event encounters a nested exception that causes a VM exit (including the case where nested exceptions lead to a triple fault).
  • Other processors delay making a last-exception record until event delivery has reached some event handler successfully (perhaps after one or more nested exceptions). Such processors do not update the last-exception record if a VM exit or triple fault occurs before an event handler is reached.

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
2. If a VM exit takes the processor from an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.
3. An exception is made if the logical processor had been inactive due to execution of MWAIT; in this case, it is considered to have become active before the VM exit.
• If the “virtual NMIs” VM-execution control is 1, VM entry injects an NMI, and delivery of the NMI causes a nested exception, double fault, task switch, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event, or APIC access that causes a VM exit, virtual-NMI blocking is in effect before the VM exit commences.

• If a VM exit results from a fault, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered during execution of IRET and the "NMI exiting" VM-execution control is 0, any blocking by NMI is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 28.2.3.

• If a VM exit results from a fault, EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered during execution of IRET and the "virtual NMIs" VM-execution control is 1, virtual-NMI blocking is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 28.2.3.

• Suppose that a VM exit is caused directly by an x87 FPU Floating-Point Error (#MF) or by any of the following events if the event was unblocked due to (and given priority over) an x87 FPU Floating-Point Error: an INIT signal, an external interrupt, an NMI, an SMI; or a machine-check exception. In these cases, there is no blocking by STI or by MOV SS when the VM exit commences.

• Normally, a last-branch record may be made when an event is delivered through the IDT. However, if such an event results in a VM exit before delivery is complete, no last-branch record is made.

• If machine-check exception results in a VM exit, processor state is suspect and may result in suspect state being saved to the guest-state area. A VM monitor should consult the RIPV and EIPV bits in the IA32_MCG_STATUS MSR before resuming a guest that caused a VM exit resulting from a machine-check exception.

• If a VM exit results from a fault, APIC access (see Section 30.4), EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that is encountered while executing an instruction, data breakpoints due to that instruction may have been recognized and information about them may be saved in the pending debug exceptions field (unless the VM exit clears that field; see Section 28.3.4).

• The following VM exits are considered to happen after an instruction is executed:
  — VM exits resulting from debug traps (single-step, I/O breakpoints, and data breakpoints).
  — VM exits resulting from debug exceptions (data breakpoints) whose recognition was delayed by blocking by MOV SS.
  — VM exits resulting from some machine-check exceptions.
  — Trap-like VM exits due to execution of MOV to CR8 when the "CR8-load exiting" VM-execution control is 0 and the "use TPR shadow" VM-execution control is 1 (see Section 30.3). (Such VM exits can occur only from 64-bit mode and thus only on processors that support Intel 64 architecture.)
  — Trap-like VM exits due to execution of WRMSR when the "use MSR bitmaps" VM-execution control is 1; the value of ECX is in the range 800H–8FFH; and the bit corresponding to the ECX value in write bitmap for low MSRs is 0; and the “virtualize x2APIC mode” VM-execution control is 1. See Section 30.5.
  — VM exits caused by APIC-write emulation (see Section 30.4.3.2) that result from APIC accesses as part of instruction execution.

For these VM exits, the instruction’s modifications to architectural state complete before the VM exit occurs. Such modifications include those to the logical processor’s interruptibility state (see Table 25-3). If there had been blocking by MOV SS, POP SS, or STI before the instruction executed, such blocking is no longer in effect.

A VM exit that occurs in enclave mode sets bit 27 of the exit-reason field and bit 4 of the guest interruptibility-state field. Before such a VM exit is delivered, an Asynchronous Enclave Exit (AEX) occurs (see Chapter 37, “Enclave Exiting Events”). An AEX modifies architectural state (Section 37.3). In particular, the processor establishes the following architectural state as indicated:

  • The following bits in RFLAGS are cleared: CF, PF, AF, ZF, SF, OF, and RF.
  • FS and GS are restored to the values they had prior to the most recent enclave entry.
  • RIP is loaded with the AEP of interrupted enclave thread.
  • RSP is loaded from the URSP field in the enclave’s state-save area (SSA).
28.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS

VM exits begin by recording information about the nature of and reason for the VM exit in the VM-exit information fields. Section 28.2.1 to Section 28.2.5 detail the use of these fields.

In addition to updating the VM-exit information fields, the valid bit (bit 31) is cleared in the VM-entry interruption-information field. If bit 5 of the IA32_VMX_MISC MSR (index 485H) is read as 1 (see Appendix A.6), the value of IA32_EFER.LMA is stored into the “IA-32e mode guest” VM-entry control.¹

28.2.1 Basic VM-Exit Information

Section 25.9.1 defines the basic VM-exit information fields. The following items detail their use.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM exit. Appendix C lists the numbers used and their meaning.
  - Bit 25 is set if the “prematurely busy shadow stack” VM-exit control is 1 and the VM exit caused a shadow stack become prematurely busy (see Section 26.4.3). Otherwise, the bit is cleared.
  - Bit 26 of this field is set to 1 if the VM exit occurred after assertion of a bus lock while the “VMM bus-lock detection” VM-execution control was 1. Such VM exits include those that occur due to the 1-setting of that control as well as others that might occur during execution of an instruction that asserted a bus lock.
  - Bit 27 of this field is set to 1 if the VM exit occurred while the logical processor was in enclave mode. Such VM exits include those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.
  - The remainder of the field (bits 31:28 and bits 24:16) is cleared to 0 (certain SMM VM exits may set some of these bits; see Section 32.15.2.3).²

- **Exit qualification.** This field is saved for VM exits due to the following causes: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); system-management interrupts (SMIs) that arrive immediately after the execution of I/O instructions; task switches; INVEPT; INVLPG; INVPCID; INVPID; LGDT; LDIT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPtrLD; VMPtrST; VMREAD; VMWRITE; VMXON; WBINVD; WBBINOVD; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; MWAIT; accesses to the APIC-access page (see Section 30.4); EPT violations (see Section 29.3.3.2); EOI virtualization (see Section 30.1.4); APIC-write emulation (see Section 30.4.3.3); page-modification log full (see Section 29.3.6); SPP-related events (see Section 29.3.4); and instruction timeout (see Section 26.2). For all other VM exits, this field is cleared. The following items provide details:
  - For a debug exception, the exit qualification contains information about the debug exception. The information has the format given in Table 28-1.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>B3 – B0. When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if its corresponding enabling bit in DR7 is not set.</td>
</tr>
<tr>
<td>10:4</td>
<td>Not currently defined.</td>
</tr>
</tbody>
</table>

1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.
2. Bit 31 of this field is set on certain VM-entry failures; see Section 27.8.

---

¹ Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.
² Bit 31 of this field is set on certain VM-entry failures; see Section 27.8.
VM EXITS

For a page-fault exception, the exit qualification contains the linear address that caused the page fault. On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the page-fault exception occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

— For a start-up IPI (SIPI), the exit qualification contains the SIPI vector information in bits 7:0. Bits 63:8 of the exit qualification are cleared to 0.

— For INVLPG, the exit qualification contains the linear-address operand of the instruction.
  • On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
  • If the INVLPG source operand specifies an unusable segment, the linear address specified in the exit qualification will match the linear address that the INVLPG would have used if no VM exit occurred. This address is not architecturally defined and may be implementation-specific.

### Table 28-1. Exit Qualification for Debug Exceptions (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>BLD. When set, this bit indicates that a bus lock was asserted while OS bus-lock detection was enabled and CPL &gt; 0 (see Section 18.3.1.6 (“OS Bus-Lock Detection”)).¹</td>
</tr>
<tr>
<td>12</td>
<td>Not currently defined.</td>
</tr>
<tr>
<td>13</td>
<td>BD. When set, this bit indicates that the cause of the debug exception is “debug register access detected.”</td>
</tr>
<tr>
<td>14</td>
<td>BS. When set, this bit indicates that the cause of the debug exception is either the execution of a single instruction (if RFLAGS.TF = 1 and IA32_DEBUGCTL.BTF = 0) or a taken branch (if RFLAGS.TF = DEBUGCTL.BTF = 1).</td>
</tr>
<tr>
<td>15</td>
<td>Not currently defined.</td>
</tr>
<tr>
<td>16</td>
<td>RTM. When set, this bit indicates that a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 16.3.7, “RTM-Enabled Debugger Support,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).²</td>
</tr>
<tr>
<td>63:17</td>
<td>Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.</td>
</tr>
</tbody>
</table>

### NOTES:

1. In general, the format of this field matches that of DR6. However, DR6 clears bit 11 to indicate detection of a bus lock, while this field sets the bit to indicate that condition.

2. In general, the format of this field matches that of DR6. However, DR6 clears bit 16 to indicate an RTM-related exception, while this field sets the bit to indicate that condition.

— For a page-fault exception, the exit qualification contains the linear address that caused the page fault. On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
  
  If the page-fault exception occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

— For a start-up IPI (SIPI), the exit qualification contains the SIPI vector information in bits 7:0. Bits 63:8 of the exit qualification are cleared to 0.

— For a task switch, the exit qualification contains details about the task switch, encoded as shown in Table 28-2.

— For INVLPG, the exit qualification contains the linear-address operand of the instruction.
  • On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.
  • If the INVLPG source operand specifies an unusable segment, the linear address specified in the exit qualification will match the linear address that the INVLPG would have used if no VM exit occurred. This address is not architecturally defined and may be implementation-specific.

### Table 28-2. Exit Qualification for Task Switches

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:0</td>
<td>Selector of task-state segment (TSS) to which the guest attempted to switch</td>
</tr>
<tr>
<td>29:16</td>
<td>Not currently defined</td>
</tr>
</tbody>
</table>

¹ Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.

² In general, the format of this field matches that of DR6. However, DR6 clears bit 16 to indicate an RTM-related exception, while this field sets the bit to indicate that condition.

Vol. 3C 28-5
For INVEPT, INVPCID, INVVPID, LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, and XSAVES, the exit qualification receives the value of the instruction’s displacement field, which is sign-extended to 64 bits if necessary (32 bits on processors that do not support Intel 64 architecture). If the instruction has no displacement (for example, has a register operand), zero is stored into the exit qualification.

On processors that support Intel 64 architecture, an exception is made for RIP-relative addressing (used only in 64-bit mode). Such addressing causes an instruction to use an address that is the sum of the displacement field and the value of RIP that references the following instruction. In this case, the exit qualification is loaded with the sum of the displacement field and the appropriate RIP value.

In all cases, bits of this field beyond the instruction’s address size are undefined. For example, suppose that the address-size field in the VM-exit instruction-information field (see Section 25.9.4 and Section 28.2.5) reports an $n$-bit address size. Then bits $63:n$ (bits $31:n$ on processors that do not support Intel 64 architecture) of the instruction displacement are undefined.

For a control-register access, the exit qualification contains information about the access and has the format given in Table 28-3.

For MOV DR, the exit qualification contains information about the instruction and has the format given in Table 28-4.

For an I/O instruction, the exit qualification contains information about the instruction and has the format given in Table 28-5.

For MWAIT, the exit qualification contains a value that indicates whether address-range monitoring hardware was armed. The exit qualification is set either to 0 (if address-range monitoring hardware is not armed) or to 1 (if address-range monitoring hardware is armed).

WBINVD and WBNNOINVD use the same basic exit reason (see Appendix C). For WBinVD, the exit qualification is 0, while for WBNNOINVD it is 1.

For an APIC-access VM exit resulting from a linear access or a guest-physical access to the APIC-access page (see Section 30.4), the exit qualification contains information about the access and has the format given in Table 28-6.¹

If the access to the APIC-access page occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

Such a VM exit that set bits 15:12 of the exit qualification to 0000b (data read during instruction execution) or 0001b (data write during instruction execution) set bit 12—which distinguishes data read from data write—to that which would have been stored in bit 1—W/R—of the page-fault error code had the access caused a page fault instead of an APIC-access VM exit. This implies the following:

- For an APIC-access VM exit caused by the CLFLUSH and CLFLUSHOPT instructions, the access type is “data read during instruction execution.”
- For an APIC-access VM exit caused by the ENTER instruction, the access type is “data write during instruction execution.”

¹ The exit qualification is undefined if the access was part of the logging of a branch record or a processor-event-based-sampling (PEBS) record to the DS save area. It is recommended that software configure the paging structures so that no address in the DS save area translates to an address on the APIC-access page.
For an APIC-access VM exit caused by the MASKMOVQ instruction or the MASKMOVDQU instruction, the access type is “data write during instruction execution.”

For an APIC-access VM exit caused by the MONITOR instruction, the access type is “data read during instruction execution.”

For an APIC-access VM exit caused directly by an access to a linear address in the DS save area (BTS or PEBS), the access type is “linear access for monitoring.”

For an APIC-access VM exit caused by a guest-physical access performed for an access to the DS save area (e.g., to access a paging structure to translate a linear address), the access type is “guest-physical access for monitoring or trace.”

For an APIC-access VM exit caused by trace-address pre-translation (TAPT) when the “Intel PT uses guest physical addresses” VM-execution control is 1, the access type is “guest-physical access for monitoring or trace.”

Such a VM exit stores 1 for bit 31 for IDT-vectoring information field (see Section 28.2.4) if and only if it sets bits 15:12 of the exit qualification to 0011b (linear access during event delivery) or 1010b (guest-physical access during event delivery).

### Table 28-3. Exit Qualification for Control-Register Accesses

<table>
<thead>
<tr>
<th>Bit Positions</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>Number of control register (0 for CLTS and LMSW). Bit 3 is always 0 on processors that do not support Intel 64 architecture as they do not support CR8.</td>
</tr>
<tr>
<td>5:4</td>
<td>Access type:</td>
</tr>
<tr>
<td></td>
<td>0 = MOV to CR</td>
</tr>
<tr>
<td></td>
<td>1 = MOV from CR</td>
</tr>
<tr>
<td></td>
<td>2 = CLTS</td>
</tr>
<tr>
<td></td>
<td>3 = LMSW</td>
</tr>
<tr>
<td>6</td>
<td>LMSW operand type:</td>
</tr>
<tr>
<td></td>
<td>0 = register</td>
</tr>
<tr>
<td></td>
<td>1 = memory</td>
</tr>
<tr>
<td></td>
<td>For CLTS and MOV CR, cleared to 0</td>
</tr>
<tr>
<td>7</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>11:8</td>
<td>For MOV CR, the general-purpose register:</td>
</tr>
<tr>
<td></td>
<td>0 = RAX</td>
</tr>
<tr>
<td></td>
<td>1 = RCX</td>
</tr>
<tr>
<td></td>
<td>2 = RDX</td>
</tr>
<tr>
<td></td>
<td>3 = RBX</td>
</tr>
<tr>
<td></td>
<td>4 = RSP</td>
</tr>
<tr>
<td></td>
<td>5 = RBP</td>
</tr>
<tr>
<td></td>
<td>6 = RSI</td>
</tr>
<tr>
<td></td>
<td>7 = RDI</td>
</tr>
<tr>
<td></td>
<td>8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>For CLTS and LMSW, cleared to 0</td>
</tr>
<tr>
<td>15:12</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>31:16</td>
<td>For LMSW, the LMSW source data</td>
</tr>
<tr>
<td></td>
<td>For CLTS and MOV CR, cleared to 0</td>
</tr>
<tr>
<td>63:32</td>
<td>Not currently defined. These bits exist only on processors that support Intel 64 architecture.</td>
</tr>
</tbody>
</table>
See Section 30.4.4 for further discussion of these instructions and APIC-access VM exits.
For APIC-access VM exits resulting from physical accesses to the APIC-access page (see Section 30.4.6),
the exit qualification is undefined.

— For an EPT violation, the exit qualification contains information about the access causing the EPT violation
and has the format given in Table 28-7.

As noted in that table, the format and meaning of the exit qualification depends on the setting of the
“mode-based execute control for EPT” VM-execution control and whether the processor supports advanced
VM-exit information for EPT violations.¹

An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data
write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation,
may differ for different kinds of read-modify-write operations.

### Table 28-4. Exit Qualification for MOV DR

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Number of debug register</td>
</tr>
<tr>
<td>3</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>4</td>
<td>Direction of access (0 = MOV to DR; 1 = MOV from DR)</td>
</tr>
<tr>
<td>7:5</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>11:8</td>
<td>General-purpose register:</td>
</tr>
<tr>
<td></td>
<td>0 = RAX</td>
</tr>
<tr>
<td></td>
<td>1 = RCX</td>
</tr>
<tr>
<td></td>
<td>2 = RDX</td>
</tr>
<tr>
<td></td>
<td>3 = RBX</td>
</tr>
<tr>
<td></td>
<td>4 = RSP</td>
</tr>
<tr>
<td></td>
<td>5 = RBP</td>
</tr>
<tr>
<td></td>
<td>6 = RSI</td>
</tr>
<tr>
<td></td>
<td>7 = RDI</td>
</tr>
<tr>
<td></td>
<td>8–15 = R8 – R15, respectively</td>
</tr>
<tr>
<td>63:12</td>
<td>Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture.</td>
</tr>
</tbody>
</table>

### Table 28-5. Exit Qualification for I/O Instructions

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Size of access:</td>
</tr>
<tr>
<td></td>
<td>0 = 1-byte</td>
</tr>
<tr>
<td></td>
<td>1 = 2-byte</td>
</tr>
<tr>
<td></td>
<td>3 = 4-byte</td>
</tr>
<tr>
<td></td>
<td>Other values not used</td>
</tr>
<tr>
<td>3</td>
<td>Direction of the attempted access (0 = OUT, 1 = IN)</td>
</tr>
<tr>
<td>4</td>
<td>String instruction (0 = not string; 1 = string)</td>
</tr>
</tbody>
</table>

¹ Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).
Table 28-5. Exit Qualification for I/O Instructions (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>REP prefixed (0 = not REP; 1 = REP)</td>
</tr>
<tr>
<td>6</td>
<td>Operand encoding (0 = DX, 1 = immediate)</td>
</tr>
<tr>
<td>15:7</td>
<td>Not currently defined</td>
</tr>
<tr>
<td>31:16</td>
<td>Port number (as specified in DX or in an immediate operand)</td>
</tr>
<tr>
<td>63:32</td>
<td>Not currently defined. These bits exist only on processors that support Intel 64 architecture.</td>
</tr>
</tbody>
</table>

Table 28-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
</table>
| 11:0            | • If the APIC-access VM exit is due to a linear access, the offset of access within the APIC page.  
• Undefined if the APIC-access VM exit is due a guest-physical access |
| 15:12           | Access type:  
0 = linear access for a data read during instruction execution  
1 = linear access for a data write during instruction execution  
2 = linear access for an instruction fetch  
3 = linear access (read or write) during event delivery  
4 = linear access for monitoring  
10 = guest-physical access during event delivery  
11 = guest-physical access for monitoring or trace  
15 = guest-physical access for an instruction fetch or during instruction execution  
Other values not used |
| 16              | This bit is set for certain accesses that are asynchronous to instruction execution and not part of event delivery. These includes guest-physical accesses related to trace output by Intel PT (see Section 26.5.4), accesses related to PEBS on processors with the “EPT-friendly” enhancement (see Section 20.9.5), and accesses that occur during user-interrupt delivery (see Section 7.4.2). |
| 63:17           | Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture. |

Bit 12 reports "NMI unblocking due to IRET"; see Section 28.2.3.

Bit 16 is set for certain accesses that are asynchronous to instruction execution and not part of event delivery. These include trace-address pre-translation (TAPT) for Intel PT (see Section 26.5.4), accesses related to PEBS on processors with the “EPT-friendly” enhancement (see Section 20.9.5), and accesses as part of user-interrupt delivery (see Section 7.4.2).

— For VM exits caused as part of EOI virtualization (Section 30.1.4), bits 7:0 of the exit qualification are set to vector of the virtual interrupt that was dismissed by the EOI virtualization. Bits above bit 7 are cleared.

— For APIC-write VM exits (Section 30.4.3.3), bits 11:0 of the exit qualification are set to the page offset of the write access that caused the VM exit.1 Bits above bit 11 are cleared.

— For a VM exit due to a page-modification log-full event (Section 29.3.6), bit 12 of the exit qualification reports "NMI unblocking due to IRET" (see Section 28.2.3). Bit 16 is set if the VM exit occurs during TAPT, EPT-friendly PEBS, or user-interrupt delivery. All other bits of the exit qualification are undefined.

— For a VM exit due to an SPP-related event (Section 29.3.4), bit 11 of the exit qualification indicates the type of event: 0 indicates an SPP misconfiguration and 1 indicates an SPP miss. Bit 12 of the exit qualification

1. Execution of WRMSR with ECX = B3FH (self-IPI MSR) can lead to an APIC-write VM exit; the exit qualification for such an APIC-write VM exit is 3F0H.
reports "NMI unblocking due to IRET" (see Section 28.2.3). Bit 16 is set if the VM exit occurs during TAPT EPT-friendly PEBS, or user-interrupt delivery. All other bits of the exit qualification are undefined.

— If the "PASID translation" VM-execution control, PASID translation is performed for executions of the ENQCMD and ENQCMDS instructions (see Section 26.5.8). PASID translation may fail, resulting in a VM exit. Such a VM exit saves an exit qualification specified in the following items:
  • For ENQCMD, the exit qualification is IA32_PASID[19:0].
  • For ENQCMDS, the exit qualification contains the low 32 bits of the instruction’s source operand (which had been read from memory prior to PASID translation).

— For a VM exit due to an instruction timeout (Section 26.2), bit 0 indicates (if set) that the context of the virtual machine is invalid and that the VM should not be resumed. Bit 12 of the exit qualification reports "NMI unblocking due to IRET" (see Section 28.2.3). All other bits of the exit qualification are undefined.

**Guest linear address.** For some VM exits, this field receives a linear address that pertains to the VM exit. The field is set for different VM exits as follows:

— VM exits due to attempts to execute LMSW with a memory operand. In these cases, this field receives the linear address of that operand. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

— VM exits due to attempts to execute INS or OUTS for which the relevant segment is usable (if the relevant segment is not usable, the value is undefined). (ES is always the relevant segment for INS; for OUTS, the relevant segment is DS unless overridden by an instruction prefix.) The linear address is the base address of relevant segment plus (E)DI (for INS) or (E)SI (for OUTS). Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

### Table 28-7. Exit Qualification for EPT Violations

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set if the access causing the EPT violation was a data read.¹</td>
</tr>
<tr>
<td>1</td>
<td>Set if the access causing the EPT violation was a data write.¹</td>
</tr>
<tr>
<td>2</td>
<td>Set if the access causing the EPT violation was an instruction fetch.</td>
</tr>
<tr>
<td>3</td>
<td>The logical-AND of bit 0 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was readable).²</td>
</tr>
<tr>
<td>4</td>
<td>The logical-AND of bit 1 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was writeable).²</td>
</tr>
<tr>
<td>5</td>
<td>The logical-AND of bit 2 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation.² If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, this indicates whether the guest-physical address was executable. If that control is 1, this indicates whether the guest-physical address was executable for supervisor-mode linear addresses.</td>
</tr>
<tr>
<td>6</td>
<td>If the &quot;mode-based execute control&quot; VM-execution control is 0, the value of this bit is undefined. If that control is 1, this bit is the logical-AND of bit 10 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation. In this case, it indicates whether the guest-physical address was executable for user-mode linear addresses.³</td>
</tr>
<tr>
<td>7</td>
<td>Set if the guest linear-address field is valid. The guest linear-address field is valid for all EPT violations except those resulting from an attempt to load the guest PDPTES as part of the execution of the MOV CR instruction and those due to trace-address pre-translation (TAPT; Section 26.5.4).</td>
</tr>
</tbody>
</table>
Table 28-7. Exit Qualification for EPT Violations (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
</table>
| 8               | If bit 7 is 1:  
|                 |   • Set if the access causing the EPT violation is to a guest-physical address that is the translation of a linear address.  
|                 |   • Clear if the access causing the EPT violation is to a paging-structure entry as part of a page walk or the update of an accessed or dirty bit.  
|                 |   Reserved if bit 7 is 0 (cleared to 0). |
| 9               | If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations; if bit 7 is 0, this bit is 0 if the linear address is a supervisor-mode linear address and 1 if it is a user-mode linear address.  
|                 |   If CR0.PG = 0, the translation of every linear address is a user-mode linear address and thus this bit will be 1.) Otherwise, this bit is undefined. |
| 10              | If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations; if bit 7 is 0, this bit is 0 if paging translates the linear address to a read-only page and 1 if it translates to a read/write page.  
|                 |   If CR0.PG = 0, every linear address is read/write and thus this bit will be 1.) Otherwise, this bit is undefined. |
| 11              | If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations; if bit 7 is 0, this bit is 0 if paging translates the linear address to an executable page and 1 if it translates to an execute-disable page.  
|                 |   If CR0.PG = 0, CR4.PAE = 0, or IA32_EFER.NXE = 0, every linear address is executable and thus this bit will be 0.) Otherwise, this bit is undefined. |
| 12              | NMI unblocking due to IRET (see Section 28.2.3). |
| 13              | Set if the access causing the EPT violation was a shadow-stack access. |
| 14              | If supervisor shadow-stack control is enabled (by setting bit 7 of EPTP), this bit is the same as bit 60 in the EPT paging-structure entry that maps the page of the guest-physical address of the access causing the EPT violation.  
|                 |   Otherwise (or if translation of the guest-physical address terminates before reaching an EPT paging-structure entry that maps a page), this bit is undefined. |
| 15              | This bit is set if the EPT violation was caused as a result of guest-paging verification. See Section 29.3.3.2. |
| 16              | This bit is set if the access was asynchronous to instruction execution not the result of event delivery. The bit is set if the access is related to trace output by Intel PT (see Section 26.5.4), accesses related to PEBS on processors with the "EPT-friendly" enhancement (see Section 20.9.5), or to user-interrupt delivery (see Section 7.4.2). Otherwise, this bit is cleared. |
| 63:17           | Not currently defined. Bits 63:32 exist only on processors that support Intel 64 architecture. |

NOTES:

1. If accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations (see Section 29.3.3.2). If such an access causes an EPT violation, the processor sets both bit 0 and bit 1 of the exit qualification.
2. Bits 5:3 are cleared to 0 if either (1) any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present; or (2) 4-level EPT is in use and the guest-physical address sets any bits in the range 51:48 (see Section 29.3.2).
3. Bit 6 is cleared to 0 if (1) the "mode-based execute control" VM-execution control is 1; and (2) either (a) any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present; or (b) 4-level EPT is in use and the guest-physical address sets any bits in the range 51:48 (see Section 29.3.2).
4. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

---

VM exits due to EPT violations that set bit 7 of the exit qualification (see Table 28-7; these are all EPT violations except those resulting from an attempt to load the PDPTes as of execution of the MOV CR instruction and those due to TAPT). The linear address may translate to the guest-physical address whose access caused the EPT violation. Alternatively, translation of the linear address may reference a paging-
structure entry whose access caused the EPT violation. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the EPT violation occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

— VM exits due to SPP-related events.

— If the "prematurely busy shadow stack" VM-exit control is 1, certain VM exits (besides those noted above) save the linear address that pertains to the VM exit if the VM exit caused a shadow stack to become prematurely busy (see Section 26.4.3). This is true for VM exits due for these reasons: EPT misconfiguration, page-modification log-full event, and instruction timeout. (A VM exit due to instruction timeout that sets bit 0 of the exit qualification, indicating that VM context is invalid, does not save a valid linear address.)

— For all other VM exits, the field is undefined.

• **Guest-physical address.** For a VM exit due to an EPT violation, an EPT misconfiguration, or an SPP-related event, this field receives the guest-physical address that caused the EPT violation or EPT misconfiguration. For all other VM exits, the field is undefined.

If the EPT violation or EPT misconfiguration occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

### 28.2.2 Information for VM Exits Due to Vectored Events

Section 25.9.2 defines fields containing information for VM exits due to the following events: exceptions (including those generated by the instructions INT1, INT3, INTO, BOUND, UD0, UD1, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs). Such VM exits include those that occur on an attempt at a task switch that causes an exception before generating the VM exit due to the task switch that causes the VM exit.

The following items detail the use of these fields:

• **VM-exit interruption information** (format given in Table 25-19). The following items detail how this field is established for VM exits due to these events:

  — For an exception, bits 7:0 receive the exception vector (at most 31). For an NMI, bits 7:0 are set to 2. For an external interrupt, bits 7:0 receive the vector.

  — Bits 10:8 are set to 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 5 (privileged software exception), or 6 (software exception). Hardware exceptions comprise all exceptions except the following:

    • Debug exceptions (#DB) generated by the INT1 instruction; these are privileged software exceptions. (Other debug exceptions are considered hardware exceptions, as are those caused by executions of INT1 in enclave mode.)

    • Breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.)

  BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD0, UD1, and UD2 are hardware exceptions.

  — Bit 11 is set to 1 if the VM exit is caused by a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0). If bit 11 is set to 1, the error code is placed in the VM-exit interruption error code (see below).

  — Bit 12 reports “NMI unblocking due to IRET”; see Section 28.2.3. The value of this bit is undefined if the VM exit is due to a double fault (the interruption type is hardware exception and the vector is 8).

1. INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT n with value 1 or 3 for n.

2. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
— Bits 30:13 are always set to 0.
— Bit 31 is always set to 1.

For other VM exits (including those due to external interrupts when the "acknowledge interrupt on exit" VM-exit control is 0), the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- **VM-exit interruption error code.**
  - For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the VM-exit interruption-information field, this field receives the error code that would have been pushed on the stack had the event causing the VM exit been delivered normally through the IDT. The EXT bit is set in this field exactly when it would be set normally. For exceptions that occur during the delivery of double fault (if the IDT-vectoring information field indicates a double fault), the EXT bit is set to 1, assuming that (1) the exception would produce an error code normally (if not incident to double-fault delivery) and (2) that the error code uses the EXT bit (not for page faults, which use a different format).
  - For other VM exits, the value of this field is undefined.

### 28.2.3 Information About NMI Unblocking Due to IRET

A VM exit may occur during execution of the IRET instruction for reasons including the following: faults, EPT violations, page-modification log-full events, SPP-related events, or instruction timeouts.

An execution of IRET that commences while non-maskable interrupts (NMIs) are blocked will unblock NMIs even if a fault or VM exit occurs; the state saved by such a VM exit will indicate that NMIs were not blocked.

VM exits for the reasons enumerated above provide more information to software by saving a bit called "NMI unblocking due to IRET." This bit is defined if (1) either the "NMI exiting" VM-execution control is 0 or the "virtual NMIs" VM-execution control is 1; (2) the VM exit does not set the valid bit in the IDT-vectoring information field (see Section 28.2.4); and (3) the VM exit is not due to a double fault. In these cases, the bit is defined as follows:

- The bit is 1 if the VM exit resulted from a memory access as part of execution of the IRET instruction and one of the following holds:
  - The "virtual NMIs" VM-execution control is 0 and blocking by NMI (see Table 25-3) was in effect before execution of IRET.
  - The "virtual NMIs" VM-execution control is 1 and virtual-NMI blocking was in effect before execution of IRET.
- The bit is 0 for all other relevant VM exits.

For VM exits due to faults, NMI unblocking due to IRET is saved in bit 12 of the VM-exit interruption-information field (Section 28.2.2). For VM exits due to EPT violations, page-modification log-full events, SPP-related events, and instruction timeouts, NMI unblocking due to IRET is saved in bit 12 of the exit qualification (Section 28.2.1).

(Executions of IRET may also incur VM exits due to APIC accesses and EPT misconfigurations. These VM exits do not report information about NMI unblocking due to IRET.)

### 28.2.4 Information for VM Exits During Event Delivery

Section 25.9.3 defined fields containing information for VM exits that occur while delivering an event through the IDT and as a result of any of the following cases:

- A fault occurs during event delivery and causes a VM exit (because the bit associated with the fault is set to 1 in the exception bitmap).
- A task switch is invoked through a task gate in the IDT. The VM exit occurs due to the task switch only after the initial checks of the task switch pass (see Section 26.4.2).
- Event delivery causes an APIC-access VM exit (see Section 30.4).

---

1. This includes the case in which a VM exit occurs while delivering a software interrupt (INT n) through the 16-bit IVT (interrupt vector table) that is used in virtual-8086 mode with virtual-machine extensions (if RFLAGS.VM = CR4.VME = 1).
VM EXITS

- An EPT violation, EPT misconfiguration, page-modification log-full event, or SPP-related event that occurs during event delivery.
- Any of the above VM exits that occur during user-interrupt notification processing (see Section 7.5.2). Such VM exits will be treated as if they occurred during delivery of an external interrupt with the vector UINV.

These fields are used for VM exits that occur during delivery of events injected as part of VM entry (see Section 27.6.1.2).

A VM exit is not considered to occur during event delivery in any of the following circumstances:

- The original event causes the VM exit directly (for example, because the original event is a non-maskable interrupt (NMI) and the "NMI exiting" VM-execution control is 1).
- The original event results in a double-fault exception that causes the VM exit directly.
- The VM exit occurred as a result of fetching the first instruction of the handler invoked by the event delivery.
- The VM exit is caused by a triple fault.
- The original event was a software interrupt (INT \( n \)) executed in virtual-8086 mode with EFLAGS.IOPL < 3 and the VM exit was due to a general-protection exception (#GP) that occurred because either CR4.VME = 0 or bit \( n \) of the software interrupt redirection bit map in the TSS is set.

The following items detail the use of these fields:

- **IDT-vectoring information** (format given in Table 25-20). The following items detail how this field is established for VM exits that occur during event delivery:
  - If the VM exit occurred during delivery of an exception, bits 7:0 receive the exception vector (at most 31). If the VM exit occurred during delivery of an NMI, bits 7:0 are set to 2. If the VM exit occurred during delivery of an external interrupt, bits 7:0 receive the vector.
  - Bits 10:8 are set to indicate the type of event that was being delivered when the VM exit occurred: 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 4 (software interrupt), 5 (privileged software interrupt), or 6 (software exception).
  - Bit 11 is set to 1 if the VM exit occurred during delivery of a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).\(^2\) If bit 11 is set to 1, the error code is placed in the IDT-vectoring error code (see below).
  - Bit 12 is undefined.
  - Bits 30:13 are always set to 0.
  - Bit 31 is always set to 1.
  - For other VM exits, the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- **IDT-vectoring error code**.

---

1. In the following items, INT1 and INT3 refer to the instructions with opcodes F1 and CC, respectively, and not to INT \( n \) with value 1 or 3 for \( n \).
2. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the IDT-vectoring information field, this field receives the error code that would have been pushed on the stack by the event that was being delivered through the IDT at the time of the VM exit. The EXT bit is set in this field when it would be set normally.

For other VM exits, the value of this field is undefined.

28.2.5 Information for VM Exits Due to Instruction Execution

Section 25.9.4 defined fields containing information for VM exits that occur due to instruction execution. (The VM-exit instruction length is also used for VM exits that occur during the delivery of a software interrupt or software exception.) The following items detail their use.

- **VM-exit instruction length.** This field is used in the following cases:
  - For fault-like VM exits due to attempts to execute one of the following instructions that cause VM exits unconditionally (see Section 26.1.2) or based on the settings of VM-execution controls (see Section 26.1.3): CLTS, CPUID, ENCLS, GETSEC, HLT, IN, INS, INVD, INVEPT, INVPCID, INVVID, LGDT, LIDT, LLDT, LMMSW, LOADIWK, LTR, MONITOR, MOV CR, MOV DR, MWAIT, OUT, OUTS, PAUSE, PCONFIG, RDMSR, RDPMC, RDRAND, RDSEED, RDTSVC, RDTSCP, RSM, SGDT, SIDT, SLDT, STR, TRIGGER, UMWAIT, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON, WBINVD, WBINVD, WRMSR, XRSTORS, XSETBV, and XSAVES.\(^1\)
  - For VM exits due to software exceptions (those generated by executions of INT3 or INTO) or privileged software exceptions (those generated by executions of IN1).
  - For VM exits due to faults encountered during delivery of a software interrupt, privileged software exception, or software exception.
  - For VM exits due to attempts to effect a task switch via instruction execution. These are VM exits that produce an exit reason indicating task switch and either of the following:
    - An exit qualification indicating execution of CALL, IRET, or JMP instruction.
    - An exit qualification indicating a task gate in the IDT and an IDT-vectoring information field indicating that the task gate was encountered during delivery of a software interrupt, privileged software exception, or software exception.
  - For APIC-access VM exits and for VM exits caused by EPT violations, page-modification log-full events, and SPP-related events encountered during delivery of a software interrupt, privileged software exception, or software exception.\(^2\)
  - For VM exits due executions of VMFUNC that fail because one of the following is true:
    - EAX indicates a VM function that is not enabled (the bit at position EAX is 0 in the VM-function controls; see Section 26.5.6.2).
    - EAX = 0 and either ECX \(\geq 512\) or the value of ECX selects an invalid tentative EPTP value (see Section 26.5.6.3).

In all the above cases, this field receives the length in bytes (1–15) of the instruction (including any instruction prefixes) whose execution led to the VM exit (see the next paragraph for one exception).

The cases of VM exits encountered during delivery of a software interrupt, privileged software exception, or software exception include those encountered during delivery of events injected as part of VM entry (see Section 27.6.1.2). If the original event was injected as part of VM entry, this field receives the value of the VM-entry instruction length.

All VM exits other than those listed in the above items leave this field undefined.

---

1. This item applies only to fault-like VM exits. It does not apply to trap-like VM exits following executions of the MOV to CR8 instruction when the “use TPR shadow” VM-execution control is 1 or to those following executions of the WRMSR instruction when the “virtualize x2APIC mode” VM-execution control is 1.

2. The VM-exit instruction-length field is not defined following APIC-access VM exits resulting from physical accesses (see Section 30.4.6) even if encountered during delivery of a software interrupt, privileged software exception, or software exception.
If the VM exit occurred in enclave mode, this field is cleared (none of the previous items apply).

### Table 28-8. Format of the VM-Exit Instruction-Information Field as Used for INS and OUTS

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>6:0</td>
<td>Undefined.</td>
</tr>
<tr>
<td>9:7</td>
<td>Address size:</td>
</tr>
<tr>
<td></td>
<td>0: 16-bit</td>
</tr>
<tr>
<td></td>
<td>1: 32-bit</td>
</tr>
<tr>
<td></td>
<td>2: 64-bit (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Other values not used.</td>
</tr>
<tr>
<td>14:10</td>
<td>Undefined.</td>
</tr>
<tr>
<td>17:15</td>
<td>Segment register:</td>
</tr>
<tr>
<td></td>
<td>0: ES</td>
</tr>
<tr>
<td></td>
<td>1: CS</td>
</tr>
<tr>
<td></td>
<td>2: SS</td>
</tr>
<tr>
<td></td>
<td>3: DS</td>
</tr>
<tr>
<td></td>
<td>4: FS</td>
</tr>
<tr>
<td></td>
<td>5: GS</td>
</tr>
<tr>
<td></td>
<td>Other values not used. Undefined for VM exits due to execution of INS.</td>
</tr>
<tr>
<td>31:18</td>
<td>Undefined.</td>
</tr>
</tbody>
</table>

**VM-exit instruction information.** For VM exits due to attempts to execute INS, INVEPT, INVPICD, INVPID, LIDT, LGDT, LLDT, LOADIWK, LTR, OUTS, RDRAND, RDSEED, SIDT, SGDT, SLDT, STR, TPAUSE, UMWAIT, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, or XSAVES, this field receives information about the instruction that caused the VM exit. The format of the field depends on the identity of the instruction causing the VM exit:

- For VM exits due to attempts to execute INS or OUTS, the field has the format is given in Table 28-8.\(^1\)
- For VM exits due to attempts to execute INVEPT, INVPICD, or INVPID, the field has the format is given in Table 28-9.
- For VM exits due to attempts to execute LIDT, LGDT, SIDT, or SGDT, the field has the format is given in Table 28-10.
- For VM exits due to attempts to execute LLDT, LTR, SLDT, or STR, the field has the format is given in Table 28-11.
- For VM exits due to attempts to execute RDRAND or RDSEED, the field has the format is given in Table 28-12.
- For VM exits due to attempts to execute TPAUSE or UMWAIT, the field has the format is given in Table 28-13.
- For VM exits due to attempts to execute VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, or XSAVES, the field has the format is given in Table 28-14.
- For VM exits due to attempts to execute VMREAD or VMWRITE, the field has the format is given in Table 28-15.
- For VM exits due to attempts to execute LOADIWK, the field has the format is given in Table 28-16.

For all other VM exits, the field is undefined, unless the VM exit occurred in enclave mode, in which case the field is cleared.

---

1. The format of the field was undefined for these VM exits on the first processors to support the virtual-machine extensions. Software can determine whether the format specified in Table 28-8 is used by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).
• **I/O RCX, I/O RSI, I/O RDI, I/O RIP.** These fields are undefined except for SMM VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions. See Section 32.15.2.3. Note that, if the VM exit occurred in enclave mode, these fields are all cleared.

**Table 28-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID**

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>Scaling:</td>
</tr>
<tr>
<td></td>
<td>0: no scaling</td>
</tr>
<tr>
<td></td>
<td>1: scale by 2</td>
</tr>
<tr>
<td></td>
<td>2: scale by 4</td>
</tr>
<tr>
<td></td>
<td>3: scale by 8 (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Undefined for instructions with no index register (bit 22 is set).</td>
</tr>
<tr>
<td>6:2</td>
<td>Undefined.</td>
</tr>
<tr>
<td>9:7</td>
<td>Address size:</td>
</tr>
<tr>
<td></td>
<td>0: 16-bit</td>
</tr>
<tr>
<td></td>
<td>1: 32-bit</td>
</tr>
<tr>
<td></td>
<td>2: 64-bit (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Other values not used.</td>
</tr>
<tr>
<td>10</td>
<td>Cleared to 0.</td>
</tr>
<tr>
<td>14:11</td>
<td>Undefined.</td>
</tr>
<tr>
<td>17:15</td>
<td>Segment register:</td>
</tr>
<tr>
<td></td>
<td>0: ES</td>
</tr>
<tr>
<td></td>
<td>1: CS</td>
</tr>
<tr>
<td></td>
<td>2: SS</td>
</tr>
<tr>
<td></td>
<td>3: DS</td>
</tr>
<tr>
<td></td>
<td>4: FS</td>
</tr>
<tr>
<td></td>
<td>5: GS</td>
</tr>
<tr>
<td></td>
<td>Other values not used.</td>
</tr>
<tr>
<td>21:18</td>
<td>IndexReg:</td>
</tr>
<tr>
<td></td>
<td>0 = RAX</td>
</tr>
<tr>
<td></td>
<td>1 = RCX</td>
</tr>
<tr>
<td></td>
<td>2 = RDX</td>
</tr>
<tr>
<td></td>
<td>3 = RBX</td>
</tr>
<tr>
<td></td>
<td>4 = RSP</td>
</tr>
<tr>
<td></td>
<td>5 = RBP</td>
</tr>
<tr>
<td></td>
<td>6 = RSI</td>
</tr>
<tr>
<td></td>
<td>7 = RDI</td>
</tr>
<tr>
<td></td>
<td>8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Undefined for instructions with no index register (bit 22 is set).</td>
</tr>
<tr>
<td>22</td>
<td>IndexReg invalid (0 = valid; 1 = invalid)</td>
</tr>
<tr>
<td>26:23</td>
<td>BaseReg (encoded as IndexReg above)</td>
</tr>
<tr>
<td></td>
<td>Undefined for memory instructions with no base register (bit 27 is set).</td>
</tr>
<tr>
<td>27</td>
<td>BaseReg invalid (0 = valid; 1 = invalid)</td>
</tr>
<tr>
<td>31:28</td>
<td>Reg2 (same encoding as IndexReg above)</td>
</tr>
</tbody>
</table>
### Table 28-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
</table>
| 1:0             | Scaling:  
|                 | 0: no scaling  
|                 | 1: scale by 2  
|                 | 2: scale by 4  
|                 | 3: scale by 8 (used only on processors that support Intel 64 architecture)  
|                 | Undefined for instructions with no index register (bit 22 is set). |
| 6:2             | Undefined. |
| 9:7             | Address size:  
|                 | 0: 16-bit  
|                 | 1: 32-bit  
|                 | 2: 64-bit (used only on processors that support Intel 64 architecture)  
|                 | Other values not used. |
| 10              | Cleared to 0. |
| 11              | Operand size:  
|                 | 0: 16-bit  
|                 | 1: 32-bit  
|                 | Undefined for VM exits from 64-bit mode. |
| 14:12           | Undefined. |
| 17:15           | Segment register:  
|                 | 0: ES  
|                 | 1: CS  
|                 | 2: SS  
|                 | 3: DS  
|                 | 4: FS  
|                 | 5: GS  
|                 | Other values not used. |
| 21:18           | IndexReg:  
|                 | 0 = RAX  
|                 | 1 = RCX  
|                 | 2 = RDX  
|                 | 3 = RBX  
|                 | 4 = RSP  
|                 | 5 = RBP  
|                 | 6 = RSI  
|                 | 7 = RDI  
|                 | 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)  
|                 | Undefined for instructions with no index register (bit 22 is set). |
| 22              | IndexReg invalid (0 = valid; 1 = invalid) |
| 26:23           | BaseReg (encoded as IndexReg above)  
|                 | Undefined for instructions with no base register (bit 27 is set). |
| 27              | BaseReg invalid (0 = valid; 1 = invalid) |
| 29:28           | Instruction identity:  
|                 | 0: SGDT  
|                 | 1: SIDT  
|                 | 2: LGDT  
|                 | 3: LIDT |
### Table 28-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:30</td>
<td>Undefined.</td>
</tr>
</tbody>
</table>

### Table 28-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
</table>
| 1:0             | Scaling:  
|                 | 0: no scaling  
|                 | 1: scale by 2  
|                 | 2: scale by 4  
|                 | 3: scale by 8 (used only on processors that support Intel 64 architecture)  
|                 | Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set). |
| 2               | Undefined. |
| 6:3             | Reg1:  
|                 | 0 = RAX  
|                 | 1 = RCX  
|                 | 2 = RDX  
|                 | 3 = RBX  
|                 | 4 = RSP  
|                 | 5 = RBP  
|                 | 6 = RSI  
|                 | 7 = RDI  
|                 | 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)  
|                 | Undefined for memory instructions (bit 10 is clear). |
| 9:7             | Address size:  
|                 | 0: 16-bit  
|                 | 1: 32-bit  
|                 | 2: 64-bit (used only on processors that support Intel 64 architecture)  
|                 | Other values not used. Undefined for register instructions (bit 10 is set). |
| 10              | Mem/Reg (0 = memory; 1 = register). |
| 14:11           | Undefined. |
| 17:15           | Segment register:  
|                 | 0: ES  
|                 | 1: CS  
|                 | 2: SS  
|                 | 3: DS  
|                 | 4: FS  
|                 | 5: GS  
|                 | Other values not used. Undefined for register instructions (bit 10 is set). |
| 21:18           | IndexReg (encoded as Reg1 above)  
|                 | Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set). |
| 22              | IndexReg invalid (0 = valid; 1 = invalid)  
|                 | Undefined for register instructions (bit 10 is set). |
| 26:23           | BaseReg (encoded as Reg1 above)  
|                 | Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set). |
### Table 28-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR (Contd.)

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
</table>
| 27              | BaseReg invalid (0 = valid; 1 = invalid)  
Undefned for register instructions (bit 10 is set). |
| 29:28           | Instruction identity:  
0: SLDT  
1: STR  
2: LLDT  
3: LTR |
| 31:30           | Undefined. |

### Table 28-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND and RDSEED

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Undefined.</td>
</tr>
</tbody>
</table>
| 6:3             | Operand register (destination register):  
0 = RAX  
1 = RCX  
2 = RDX  
3 = RBX  
4 = RSP  
5 = RBP  
6 = RSI  
7 = RDI  
8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) |
| 10:7            | Undefined. |
| 12:11           | Operand size:  
0: 16-bit  
1: 32-bit  
2: 64-bit  
The value 3 is not used. |
| 31:13           | Undefined. |

### Table 28-13. Format of the VM-Exit Instruction-Information Field as Used for TPAUSE and UMWAIT

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Undefined.</td>
</tr>
</tbody>
</table>
| 6:3             | Operand register (source register):  
0 = RAX  
1 = RCX  
2 = RDX  
3 = RBX  
4 = RSP  
5 = RBP  
6 = RSI  
7 = RDI  
8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) |
| 31:7            | Undefined. |
Table 28-14. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>Scaling:</td>
</tr>
<tr>
<td></td>
<td>0: no scaling</td>
</tr>
<tr>
<td></td>
<td>1: scale by 2</td>
</tr>
<tr>
<td></td>
<td>2: scale by 4</td>
</tr>
<tr>
<td></td>
<td>3: scale by 8 (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Undefined for instructions with no index register (bit 22 is set).</td>
</tr>
<tr>
<td>6:2</td>
<td>Undefined.</td>
</tr>
<tr>
<td>9:7</td>
<td>Address size:</td>
</tr>
<tr>
<td></td>
<td>0: 16-bit</td>
</tr>
<tr>
<td></td>
<td>1: 32-bit</td>
</tr>
<tr>
<td></td>
<td>2: 64-bit (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Other values not used.</td>
</tr>
<tr>
<td>10</td>
<td>Cleared to 0.</td>
</tr>
<tr>
<td>14:11</td>
<td>Undefined.</td>
</tr>
<tr>
<td>17:15</td>
<td>Segment register:</td>
</tr>
<tr>
<td></td>
<td>0: ES</td>
</tr>
<tr>
<td></td>
<td>1: CS</td>
</tr>
<tr>
<td></td>
<td>2: SS</td>
</tr>
<tr>
<td></td>
<td>3: DS</td>
</tr>
<tr>
<td></td>
<td>4: FS</td>
</tr>
<tr>
<td></td>
<td>5: GS</td>
</tr>
<tr>
<td></td>
<td>Other values not used.</td>
</tr>
<tr>
<td>21:18</td>
<td>IndexReg:</td>
</tr>
<tr>
<td></td>
<td>0 = RAX</td>
</tr>
<tr>
<td></td>
<td>1 = RCX</td>
</tr>
<tr>
<td></td>
<td>2 = RDX</td>
</tr>
<tr>
<td></td>
<td>3 = RBX</td>
</tr>
<tr>
<td></td>
<td>4 = RSP</td>
</tr>
<tr>
<td></td>
<td>5 = RBP</td>
</tr>
<tr>
<td></td>
<td>6 = RSI</td>
</tr>
<tr>
<td></td>
<td>7 = RDI</td>
</tr>
<tr>
<td></td>
<td>8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Undefined for instructions with no index register (bit 22 is set).</td>
</tr>
<tr>
<td>22</td>
<td>IndexReg invalid (0 = valid; 1 = invalid)</td>
</tr>
<tr>
<td>26:23</td>
<td>BaseReg (encoded as IndexReg above)</td>
</tr>
<tr>
<td></td>
<td>Undefined for instructions with no base register (bit 27 is set).</td>
</tr>
<tr>
<td>27</td>
<td>BaseReg invalid (0 = valid; 1 = invalid)</td>
</tr>
<tr>
<td>31:28</td>
<td>Undefined.</td>
</tr>
</tbody>
</table>

28.3 SAVING GUEST STATE

VM exits save certain components of processor state into corresponding fields in the guest-state area of the VMCS (see Section 25.4). On processors that support Intel 64 architecture, the full value of each natural-width field (see Section 25.11.2) is saved regardless of the mode of the logical processor before and after the VM exit.
In general, the state saved is that which was in the logical processor at the time the VM exit commences. See Section 28.1 for a discussion of which architectural updates occur at that time.

### Table 28-15. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>Scaling:</td>
</tr>
<tr>
<td></td>
<td>0: no scaling</td>
</tr>
<tr>
<td></td>
<td>1: scale by 2</td>
</tr>
<tr>
<td></td>
<td>2: scale by 4</td>
</tr>
<tr>
<td></td>
<td>3: scale by 8 (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).</td>
</tr>
<tr>
<td>2</td>
<td>Undefined.</td>
</tr>
<tr>
<td>6:3</td>
<td>Reg1:</td>
</tr>
<tr>
<td></td>
<td>0 = RAX</td>
</tr>
<tr>
<td></td>
<td>1 = RCX</td>
</tr>
<tr>
<td></td>
<td>2 = RDX</td>
</tr>
<tr>
<td></td>
<td>3 = RBX</td>
</tr>
<tr>
<td></td>
<td>4 = RSP</td>
</tr>
<tr>
<td></td>
<td>5 = RBP</td>
</tr>
<tr>
<td></td>
<td>6 = RSI</td>
</tr>
<tr>
<td></td>
<td>7 = RDI</td>
</tr>
<tr>
<td></td>
<td>8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Undefined for memory instructions (bit 10 is clear).</td>
</tr>
<tr>
<td>9:7</td>
<td>Address size:</td>
</tr>
<tr>
<td></td>
<td>0: 16-bit</td>
</tr>
<tr>
<td></td>
<td>1: 32-bit</td>
</tr>
<tr>
<td></td>
<td>2: 64-bit (used only on processors that support Intel 64 architecture)</td>
</tr>
<tr>
<td></td>
<td>Other values not used. Undefined for register instructions (bit 10 is set).</td>
</tr>
<tr>
<td>10</td>
<td>Mem/Reg (0 = memory; 1 = register).</td>
</tr>
<tr>
<td>14:11</td>
<td>Undefined.</td>
</tr>
<tr>
<td>17:15</td>
<td>Segment register:</td>
</tr>
<tr>
<td></td>
<td>0: ES</td>
</tr>
<tr>
<td></td>
<td>1: CS</td>
</tr>
<tr>
<td></td>
<td>2: SS</td>
</tr>
<tr>
<td></td>
<td>3: DS</td>
</tr>
<tr>
<td></td>
<td>4: FS</td>
</tr>
<tr>
<td></td>
<td>5: GS</td>
</tr>
<tr>
<td></td>
<td>Other values not used. Undefined for register instructions (bit 10 is set).</td>
</tr>
<tr>
<td>21:18</td>
<td>IndexReg (encoded as Reg1 above)</td>
</tr>
<tr>
<td></td>
<td>Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).</td>
</tr>
<tr>
<td>22</td>
<td>IndexReg invalid (0 = valid; 1 = invalid)</td>
</tr>
<tr>
<td></td>
<td>Undefined for register instructions (bit 10 is set).</td>
</tr>
<tr>
<td>26:23</td>
<td>BaseReg (encoded as Reg1 above)</td>
</tr>
<tr>
<td></td>
<td>Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).</td>
</tr>
<tr>
<td>27</td>
<td>BaseReg invalid (0 = valid; 1 = invalid)</td>
</tr>
<tr>
<td></td>
<td>Undefined for register instructions (bit 10 is set).</td>
</tr>
<tr>
<td>31:28</td>
<td>Reg2 (same encoding as Reg1 above)</td>
</tr>
</tbody>
</table>
Section 28.3.1 through Section 28.3.4 provide details for how various components of processor state are saved. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

### 28.3.1 Saving Control Registers, Debug Registers, and MSRs

Contents of certain control registers, debug registers, and MSRs are saved as follows:

- The contents of CR0, CR3, CR4, and the IA32_SYSENTER_CS, IA32_SYSENTER_ESP, and IA32_SYSENTER_EIP MSRs are saved into the corresponding fields. Bits 63:32 of the IA32_SYSENTER_CS MSR are not saved. On processors that do not support Intel 64 architecture, bits 63:32 of the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are not saved.
- If the “save debug controls” VM-exit control is 1, the contents of DR7 and the IA32_DEBUGCTL MSR are saved into the corresponding fields. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always saved data into these fields.
- If the “save IA32_PAT” VM-exit control is 1, the contents of the IA32_PAT MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the “load IA32_BNDCFGS” VM-entry control or that of the “clear IA32_BNDCFGS” VM-exit control, the contents of the IA32_BNDCFGS MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the “load IA32_RTIT_CTL” VM-entry control or that of the “clear IA32_RTIT_CTL” VM-exit control, the contents of the IA32_RTIT_CTL MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the “load CET” VM-entry control, the contents of the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are saved into the corresponding fields. On processors that do not support Intel 64 architecture, bits 63:32 of these MSRs are not saved.
- If the processor supports either the 1-setting of the “load guest IA32_LBR_CTL” VM-entry control or that of the “clear IA32_LBR_CTL” VM-exit control, the contents of the IA32_LBR_CTL MSR are saved into the corresponding field.
- If the processor supports the 1-setting of the “load PKRS” VM-entry control, the contents of the IA32_PKRS MSR are saved into the corresponding field.
- If a processor supports user interrupts, every VM exit saves UINV into the guest UINV field in the VMCS (bits 15:8 of the field are cleared).
- If the “save IA32_PERF_GLOBAL_CTL” VM-exit control is 1, the contents of the IA32_PERF_GLOBAL_CTL MSR are saved into the corresponding field.
- The value of the SMBASE field is undefined after all VM exits except SMM VM exits. See Section 32.15.2.

### 28.3.2 Saving Segment Registers and Descriptor-Table Registers

For each segment register (CS, SS, DS, ES, FS, GS, LDTR, or TR), the values saved for the base-address, segment-limit, and access rights are based on whether the register was unusable (see Section 25.4.1) before the VM exit:

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Undefined.</td>
</tr>
<tr>
<td>6:3</td>
<td>Reg1: identifies the first XMM register operand (XMM0–XMM15; values 8–15 are used only on processors that support Intel 64 architecture).</td>
</tr>
<tr>
<td>30:7</td>
<td>Undefined.</td>
</tr>
<tr>
<td>31:28</td>
<td>Reg2: identifies the second XMM register operand (see above).</td>
</tr>
</tbody>
</table>
If the register was unusable, the values saved into the following fields are undefined: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in the access-rights field. The following exceptions apply:

- CS.
  - The base-address and segment-limit fields are saved.
  - The L, D, and G bits are saved in the access-rights field.
- SS.
  - DPL is saved in the access-rights field.
  - On processors that support Intel 64 architecture, bits 63:32 of the value saved for the base address are always zero.
- DS and ES. On processors that support Intel 64 architecture, bits 63:32 of the values saved for the base addresses are always zero.
- FS and GS. The base-address field is saved.
- LDTR. The value saved for the base address is always canonical.

If the register was not unusable, the values saved into the following fields are those which were in the register before the VM exit: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in access rights.

- Bits 31:17 and 11:8 in the access-rights field are always cleared. Bit 16 is set to 1 if and only if the segment is unusable.

The contents of the GDTR and IDTR registers are saved into the corresponding base-address and limit fields.

### 28.3.3 Saving RIP, RSP, RFLAGS, and SSP

The contents of the RIP, RSP, RFLAGS, and SSP (shadow-stack pointer) registers are saved as follows:

- The value saved in the RIP field is determined by the nature and cause of the VM exit:
  - If the VM exit occurred in enclave mode, the value saved is the AEP of interrupted enclave thread (the remaining items do not apply).
  - If the VM exit occurs due to an attempt to execute an instruction that causes VM exits unconditionally or that has been configured to cause a VM exit via the VM-execution controls, the value saved references that instruction.
  - If the VM exit is caused by an occurrence of an INIT signal, a start-up IPI (SIPI), or system-management interrupt (SMI), the value saved is that which was in RIP before the event occurred.
  - If the VM exit occurs due to the 1-setting of either the “interrupt-window exiting” VM-execution control or the “NMI-window exiting” VM-execution control, the value saved is that which would be in the register had the VM exit not occurred.
  - If the VM exit is due to an external interrupt, non-maskable interrupt (NMI), or hardware exception (as defined in Section 28.2.2), the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate,¹ or into the old task-state segment had the event been delivered through a task gate).
  - If the VM exit is due to a triple fault, the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate, or into the old task-state segment had the event been delivered through a task gate) had delivery of the double fault not encountered the nested exception that caused the triple fault.
  - If the VM exit is due to a software exception (due to an execution of INT3 or INTO) or a privileged software exception (due to an execution of INT1), the value saved references the INT3, INTO, or INT1 instruction that caused that exception.

---

1. The reference here is to the full value of RIP before any truncation that would occur had the stack width been only 32 bits or 16 bits.
— Suppose that the VM exit is due to a task switch that was caused by execution of CALL, IRET, or JMP or by execution of a software interrupt (INT \( n \)), software exception (due to execution of INT3 or INTO), or privileged software exception (due to execution of INT1) that encountered a task gate in the IDT. The value saved references the instruction that caused the task switch (CALL, IRET, JMP, INT \( n \), INT3, INTO, INT1).

— Suppose that the VM exit is due to a task switch that was caused by a task gate in the IDT that was encountered for any reason except the direct access by a software interrupt or software exception. The value saved is that which would have been saved in the old task-state segment had the task switch completed normally.

— If the VM exit is due to an execution of MOV to CR8 or WRMSR that reduced the value of bits 7:4 of VTPR (see Section 30.1.1) below that of TPR threshold VM-execution control field (see Section 30.1.2), the value saved references the instruction following the MOV to CR8 or WRMSR.

— If the VM exit was caused by APIC-write emulation (see Section 30.4.3.2) that results from an APIC access as part of instruction execution, the value saved references the instruction following the one whose execution caused the APIC-write emulation.

- The contents of the RSP register are saved into the RSP field.

- With the exception of the resume flag (RF; bit 16), the contents of the RFLAGS register is saved into the RFLAGS field. RFLAGS.RF is saved as follows:
  - If the VM exit occurred in enclave mode, the value saved is 0 (the remaining items do not apply).
  - If the VM exit is caused directly by an event that would normally be delivered through the IDT, the value saved is that which would appear in the saved RFLAGS image (either that which would be saved on the stack had the event been delivered through a trap or interrupt gate\(^1\) or into the old task-state segment had the event been delivered through a task gate) had the event been delivered through the IDT. See below for VM exits due to task switches caused by task gates in the IDT.
  - If the VM exit is caused by a triple fault, the value saved is that which the logical processor would have in RF in the RFLAGS register had the triple fault taken the logical processor to the shutdown state.
  - If the VM exit is caused by a task switch (including one caused by a task gate in the IDT), the value saved is that which would have been saved in the RFLAGS image in the old task-state segment (TSS) had the task switch completed normally without exception.
  - If the VM exit is caused by an attempt to execute an instruction that unconditionally causes VM exits or one that was configured to do with a VM-execution control, the value saved is 0.\(^2\)
  - For APIC-access VM exits and for VM exits caused by EPT violations, EPT misconfigurations, page-modification log-full events, or SPP-related events, the value saved depends on whether the VM exit occurred during delivery of an event through the IDT:
    - If the VM exit stored 0 for bit 31 for IDT-vectoring information field (because the VM exit did not occur during delivery of an event through the IDT; see Section 28.2.4), the value saved is 1.
    - If the VM exit stored 1 for bit 31 for IDT-vectoring information field (because the VM exit did occur during delivery of an event through the IDT), the value saved is the value that would have appeared in the saved RFLAGS image had the event been delivered through the IDT (see above).
  - For all other VM exits, the value saved is the value RFLAGS.RF had before the VM exit occurred.

- If the processor supports the 1-setting of the “load CET” VM-entry control, the contents of the SSP register are saved into the SSP field.

---

\(^1\) The reference here is to the full value of RFLAGS before any truncation that would occur had the stack width been only 32 bits or 16 bits.

\(^2\) This is true even if RFLAGS.RF was 1 before the instruction was executed. If, in response to such a VM exit, a VM monitor re-enters the guest to re-execute the instruction that caused the VM exit (for example, after clearing the VM-execution control that caused the VM exit), the instruction may encounter a code breakpoint that has already been processed. A VM monitor can avoid this by setting the guest value of RFLAGS.RF to 1 before resuming guest software.
28.3.4 Saving Non-Register State

Information corresponding to guest non-register state is saved as follows:

- The activity-state field is saved with the logical processor’s activity state before the VM exit. See Section 28.1 for details of how events leading to a VM exit may affect the activity state. If the VM exit occurred during user-interrupt notification processing (see Section 7.5.2) and the logical processor would have entered the HLT state following user-interrupt notification processing, the saved activity state is “HLT”.

- The interruptibility-state field is saved to reflect the logical processor’s interruptibility before the VM exit.
  - See Section 28.1 for details of how events leading to a VM exit may affect this state.
  - VM exits that end outside system-management mode (SMM) save bit 2 (blocking by SMI) as 0 regardless of the state of such blocking before the VM exit.
  - Bit 3 (blocking by NMI) is treated specially if the “virtual NMIs” VM-execution control is 1. In this case, the value saved for this field does not indicate the blocking of NMIs but rather the state of virtual-NMI blocking.
  - Bit 4 (enclave interruption) is set to 1 if the VM exit occurred while the logical processor was in enclave mode.
    Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.
    A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.

- The pending debug exceptions field is saved as clear for all VM exits except the following:
  - A VM exit caused by an INIT signal, a machine-check exception, or a system-management interrupt (SMI).
  - A VM exit with basic exit reason “TPR below threshold”, “virtualized EOI”, “APIC write”, ”monitor trap flag,” or “bus-lock detected.”
  - A VM exit due to trace-address pre-translation (TAPT; see Section 26.5.4) or due to accesses related to PEBS on processors with the “EPT-friendly” enhancement (see Section 20.9.5). Such VM exits can have basic exit reason “APIC access,” “EPT violation,” “EPT misconfiguration,” “page-modification log full,” or “SPP-related event.” When due to TAPT or PEBS, these VM exits (with the exception of those due to EPT misconfigurations) set bit 16 of the exit qualification, indicating that they are asynchronous to instruction execution and not part of event delivery.
  - VM exits that are not caused by debug exceptions and that occur while there is MOV-SS blocking of debug exceptions.

For VM exits that do not clear the field, the value saved is determined as follows:

- Each of bits 3:0 may be set if it corresponds to a matched breakpoint. This may be true even if the corresponding breakpoint is not enabled in DR7.

- Suppose that a VM exit is due to an INIT signal, a machine-check exception, or an SMI; or that a VM exit has basic exit reason “TPR below threshold” or ”monitor trap flag.” In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit.

If the VM exit occurs immediately after VM entry, the value saved may match that which was loaded on VM entry (see Section 27.7.3). Otherwise, the following items apply:

- Bit 12 (enabled breakpoint) is set to 1 in any of the following cases:
  - If there was at least one matched data or I/O breakpoint that was enabled in DR7.
  - If it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 27.7.3) and the VM exit occurred before those exceptions were either delivered or lost.
  - If the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled (see Section 16.3.7, ”RTM-Enabled Debugger

1. If this activity state was an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.
2. This item includes VM exits that occur as a result of certain VM entries (Section 27.7.7).
Support,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). (This does not apply to VM exits with basic exit reason "monitor trap flag.")

— If a bus lock was asserted while CPL > 0 and OS bus-lock detection was enabled.
In other cases, bit 12 is cleared to 0.

• Bit 14 (BS) is set if RFLAGS.TF = 1 in either of the following cases:
  — IA32_DEBUGCTL.BTF = 0 and the cause of a pending debug exception was the execution of a single instruction.
  — IA32_DEBUGCTL.BTF = 1 and the cause of a pending debug exception was a taken branch.

• Bit 16 (RTM) is set if a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions had been enabled. (This does not apply to VM exits with basic exit reason "monitor trap flag.")

— Suppose that a VM exit is due to another reason (but not a debug exception) and occurs while there is MOV-SS blocking of debug exceptions. In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit. If the VM exit occurs immediately after VM entry (no instructions were executed in VMX non-root operation), the value saved may match that which was loaded on VM entry (see Section 27.7.3). Otherwise, the following items apply:

• Bit 12 (enabled breakpoint) is set to 1 if there was at least one matched data or I/O breakpoint that was enabled in DR7. Bit 12 is also set if it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 27.7.3) and the VM exit occurred before those exceptions were either delivered or lost. In other cases, bit 12 is cleared to 0.

• The setting of bit 14 (BS) is implementation-specific. However, it is not set if RFLAGS.TF = 0 or IA32_DEBUGCTL.BTF = 1.

— The reserved bits in the field are cleared.

• If the “save VMX-preemption timer value” VM-exit control is 1, the value of timer is saved into the VMX-preemption timer-value field. This is the value loaded from this field on VM entry as subsequently decremented (see Section 26.5.1). VM exits due to timer expiration save the value 0. Other VM exits may also save the value 0 if the timer expired during VM exit. (If the “save VMX-preemption timer value” VM-exit control is 0, VM exit does not modify the value of the VMX-preemption timer-value field.)

• If the logical processor supports the 1-setting of the “enable EPT” VM-execution control, values are saved into the four (4) PDPTE fields as follows:
  — If the “enable EPT” VM-execution control is 1 and the logical processor was using PAE paging at the time of the VM exit, the PDPTPE values currently in use are saved:
    • The values saved into bits 11:9 of each of the fields is undefined.
    • If the value saved into one of the fields has bit 0 (present) clear, the value saved into bits 63:1 of that field is undefined. That value need not correspond to the value that was loaded by VM entry or to any value that might have been loaded in VMX non-root operation.
    • If the value saved into one of the fields has bit 0 (present) set, the value saved into bits 63:12 of the field is a guest-physical address.
    — If the “enable EPT” VM-execution control is 0 or the logical processor was not using PAE paging at the time of the VM exit, the values saved are undefined.

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM exit functions as if the "enable EPT" VM-execution control were 0. See Section 25.6.2.
28.4 SAVING MSRS

After processor state is saved to the guest-state area, values of MSRs may be stored into the VM-exit MSR-store area (see Section 25.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-store count) is processed in order by storing the value of the MSR indexed by bits 31:0 (as they would be read by RDMSR) into bits 127:64. Processing of an entry fails in either of the following cases:

- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be read only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMBASE is an MSR that can be read only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be saved on VM exits for model-specific reasons. A processor may prevent certain MSRs (based on the value of bits 31:0) from being stored on VM exits, even if they can normally be read by RDMSR. Such model-specific behavior is documented in Chapter 2, “Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
- Bits 63:32 of the entry are not all 0.
- An attempt to read the MSR indexed by bits 31:0 would cause a general-protection exception if executed via RDMSR with CPL = 0.

A VMX abort occurs if processing fails for any entry. See Section 28.7.

28.5 LOADING HOST STATE

Processor state is updated on VM exits in the following ways:

- Some state is loaded from or otherwise determined by the contents of the host-state area.
- Some state is determined by VM-exit controls.
- Some state is established in the same way on every VM exit.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order.

On processors that support Intel 64 architecture, the full values of each 64-bit field loaded (for example, the base address for GDTR) is loaded regardless of the mode of the logical processor before and after the VM exit.

The loading of host state is detailed in Section 28.5.1 to Section 28.5.5. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

A logical processor is in IA-32e mode after a VM exit only if the "host address-space size” VM-exit control is 1. If the logical processor was in IA-32e mode before the VM exit and this control is 0, a VMX abort occurs. See Section 28.7.

In addition to loading host state, VM exits clear address-range monitoring (Section 28.5.6).

After the state loading described in this section, VM exits may load MSRs from the VM-exit MSR-load area (see Section 28.6). This loading occurs only after the state loading described in this section.

28.5.1 Loading Host Control Registers, Debug Registers, MSRs

VM exits load new values for controls registers, debug registers, and some MSRs:

- CR0, CR3, and CR4 are loaded from the CR0 field, the CR3 field, and the CR4 field, respectively, with the following exceptions:
  - The following bits are not modified:
    - For CR0, ET, CD, NW; bits 63:32 (on processors that support Intel 64 architecture), 28:19, 17, and 15:6; and any bits that are fixed in VMX operation (see Section 24.8).\(^1\)
    - For CR3, bits 63:52 and bits in the range 51:32 beyond the processor’s physical-address width (they are cleared to 0).\(^2\) (This item applies only to processors that support Intel 64 architecture.)
For CR4, any bits that are fixed in VMX operation (see Section 24.8).

- CR4.PAE is set to 1 if the "host address-space size" VM-exit control is 1.
- CR4.PCIDE is set to 0 if the "host address-space size" VM-exit control is 0.

- DR7 is set to 400H.
- If the “clear UINV” VM-exit control is 1, VM exit clears UINV.

The following MSRs are established as follows:

- The IA32_DEBUGCTL MSR is cleared to 00000000_00000000H.
- The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since that field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
- The IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are loaded from the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP fields, respectively.
  - If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
  - If the processor supports the Intel 64 architecture with N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.
- The following steps are performed on processors that support Intel 64 architecture:
  - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 28.5.2).
  - The LMA and LME bits in the IA32_EFER MSR are each loaded with the setting of the "host address-space size" VM-exit control.
  - If the "load IA32_PERF_GLOBAL_CTRL” VM-exit control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "load IA32_PAT” VM-exit control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "load IA32_EFER” VM-exit control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the “clear IA32_BNDCFGS” VM-exit control is 1, the IA32_BNDCFGS MSR is cleared to 00000000_00000000H; otherwise, it is not modified.
  - If the “clear IA32_RTIT_CTL” VM-exit control is 1, the IA32_RTIT_CTL MSR is cleared to 00000000_00000000H; otherwise, it is not modified.
  - If the "load CET” VM-exit control is 1, the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR MSRs are loaded from the IA32_S_CET and IA32_INTERRUPT_SSP_TABLE_ADDR fields, respectively.
    - If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
    - If the processor supports the Intel 64 architecture with N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.
  - If the "load PKRS” VM-exit control is 1, the IA32_PKRS MSR is loaded from the IA32_PKRS field. Bits 63:32 of that MSR are maintained with zeroes.

With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-exit MSR-load area. See Section 28.6.

---

1. Bits 28:19, 17, and 15:6 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. CR0.ET is always 1 and the other bits are always 0.
2. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.
28.5.2 Loading Host Segment and Descriptor-Table Registers

Each of the registers CS, SS, DS, ES, FS, GS, and TR is loaded as follows (see below for the treatment of LDTR):

- The selector is loaded from the selector field. The segment is unusable if its selector is loaded with zero. The checks specified in Section 27.2.3 limit the selector values that may be loaded. In particular, CS and TR are never loaded with zero and are thus never unusable. SS can be loaded with zero only on processors that support Intel 64 architecture and only if the VM exit is to 64-bit mode (64-bit mode allows use of segments marked unusable).

- The base address is set as follows:
  - CS. Cleared to zero.
  - SS, DS, and ES. Undefined if the segment is unusable; otherwise, cleared to zero.
  - FS and GS. Undefined (but, on processors that support Intel 64 architecture, canonical) if the segment is unusable and the VM exit is not to 64-bit mode; otherwise, loaded from the base-address field.
  - TR. Loaded from the host-state area. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.1 The values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.

- The segment limit is set as follows:
  - CS. Set to FFFFFFFFH (corresponding to a descriptor limit of FFFFH and a G-bit setting of 1).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to FFFFFFFFH.
  - TR. Set to 00000067H.

- The type field and S bit are set as follows:
  - CS. Type set to 11 and S set to 1 (execute/read, accessed, non-conforming code segment).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, type set to 3 and S set to 1 (read/write, accessed, expand-up data segment).
  - TR. Type set to 11 and S set to 0 (busy 32-bit task-state segment).

- The DPL is set as follows:
  - CS, SS, and TR. Set to 0. The current privilege level (CPL) will be 0 after the VM exit completes.
  - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 0.

- The P bit is set as follows:
  - CS, TR. Set to 1.
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.

- On processors that support Intel 64 architecture, CS.L is loaded with the setting of the "host address-space size" VM-exit control. Because the value of this control is also loaded into IA32_EFER.LMA (see Section 28.5.1), no VM exit is ever to compatibility mode (which requires IA32_EFER.LMA = 1 and CS.L = 0).

- D/B.
  - CS. Loaded with the inverse of the setting of the "host address-space size" VM-exit control. For example, if that control is 0, indicating a 32-bit guest, CS.D/B is set to 1.
  - SS. Set to 1.
  - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
  - TR. Set to 0.

- G.

---

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.
— CS. Set to 1.
— SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
— TR. Set to 0.

The host-state area does not contain a selector field for LDTR. LDTR is established as follows on all VM exits: the selector is cleared to 0000H, the segment is marked unusable and is otherwise undefined (although the base address is always canonical).

The base addresses for GDTR and IDTR are loaded from the GDTR base-address field and the IDTR base-address field, respectively. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N of each base address is set to the value of bit N–1 of that base address. The GDTR and IDTR limits are each set to FFFFH.

28.5.3 Loading Host RIP, RSP, RFLAGS, and SSP

RIP and RSP are loaded from the RIP field and the RSP field, respectively. RFLAGS is cleared, except bit 1, which is always set.

If the "load CET" VM-exit control is 1, SSP (shadow-stack pointer) is loaded from the SSP field.

28.5.4 Checking and Loading Host Page-Directory-Pointer-Table Entries

If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LMA = 0, the logical processor uses PAE paging. See Section 4.4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. When in PAE paging is in use, the physical address in CR3 references a table of page-directory-pointer-table entries (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs and, if they are valid, loads them into the processor (into internal, non-architectural registers).

A VM exit is to a VMM that uses PAE paging if (1) bit 5 (corresponding to CR4.PAE) is set in the CR4 field in the host-state area of the VMCS; and (2) the "host address-space size" VM-exit control is 0. Such a VM exit may check the validity of the PDPTEs referenced by the CR3 field in the host-state area of the VMCS. Such a VM exit must check their validity if either (1) PAE paging was not in use before the VM exit; or (2) the value of CR3 is changing as a result of the VM exit. A VM exit to a VMM that does not use PAE paging must not check the validity of the PDPTEs.

A VM exit that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use. If MOV to CR3 would cause a general-protection exception due to the PDPTEs that would be loaded (e.g., because a reserved bit is set), a VMX abort occurs (see Section 28.7). If a VM exit to a VMM that uses PAE does not cause a VMX abort, the PDPTEs are loaded into the processor as would MOV to CR3, using the value of CR3 being load by the VM exit.

28.5.5 Updating Non-Register State

VM exits affect the non-register state of a logical processor as follows:

• A logical processor is always in the active state after a VM exit.
• Event blocking is affected as follows:
  — There is no blocking by STI or by MOV SS after a VM exit.
  — VM exits caused directly by non-maskable interrupts (NMIs) cause blocking by NMI (see Table 25-3). Other VM exits do not affect blocking by NMI. (See Section 28.1 for the case in which an NMI causes a VM exit indirectly.)
• There are no pending debug exceptions after a VM exit.

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
Section 29.4 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM exits invalidate cached mappings:

- If the "enable VPID" VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP).
- VM exits are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the "enable VPID" VM-execution control is 1.

28.5.6 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 9.10.4 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. VM exits clear any address-range monitoring that may be in effect.

28.6 LOADING MSRS

VM exits may load MSRs from the VM-exit MSR-load area (see Section 25.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32_FS_BASE MSR) or C0000101H (the IA32_GS_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM exits for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, "Model-Specific Registers (MSRs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.1

If processing fails for any entry, a VMX abort occurs. See Section 28.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM exit, the logical processor does not use any translations that were cached before the transition.

28.7 VMX ABORTS

A problem encountered during a VM exit leads to a VMX abort. A VMX abort takes a logical processor into a shut-down state as described below.

A VMX abort does not modify the VMCS data in the VMCS region of any active VMCS. The contents of these data are thus suspect after the VMX abort.

On a VMX abort, a logical processor saves a nonzero 32-bit VMX-abort indicator field at byte offset 4 in the VMCS region of the VMCS whose misconfiguration caused the failure (see Section 25.2). The following values are used:

1. Note the following about processors that support Intel 64 architecture. If CR0.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. Since CR0.PG is always 1 in VMX operation, the IA32_EFER MSR should not be included in the VM-exit MSR-load area for the purpose of modifying the LME bit.
1. There was a failure in saving guest MSRs (see Section 28.4).
2. Host checking of the page-directory-pointer-table entries (PDPTEs) failed (see Section 28.5.4).
3. The current VMCS has been corrupted (through writes to the corresponding VMCS region) in such a way that the logical processor cannot complete the VM exit properly.
4. There was a failure on loading host MSRs (see Section 28.6).
5. There was a machine-check event during VM exit (see Section 28.8).
6. The logical processor was in IA-32e mode before the VM exit and the "host address-space size" VM-exit control was 0 (see Section 28.5).

Some of these causes correspond to failures during the loading of state from the host-state area. Because the loading of such state may be done in any order (see Section 28.5) a VM exit that might lead to a VMX abort for multiple reasons (for example, the current VMCS may be corrupt and the host PDPTEs might not be properly configured). In such cases, the VMX-abort indicator could correspond to any one of those reasons.

A logical processor never reads the VMX-abort indicator in a VMCS region and writes it only with one of the non-zero values mentioned above. The VMX-abort indicator allows software on one logical processor to diagnose the VMX-abort on another. For this reason, it is recommended that software running in VMX root operation zero the VMX-abort indicator in the VMCS region of any VMCS that it uses.

After saving the VMX-abort indicator, operation of a logical processor experiencing a VMX abort depends on whether the logical processor is in SMX operation:

- If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000DH, indicating “VMX abort.” See the Intel® Trusted Execution Technology Measured Launched Environment Programming Guide.
- If the logical processor is outside SMX operation, it issues a special bus cycle (to notify the chipset) and enters the **VMX-abort shutdown state**. RESET is the only event that wakes a logical processor from the VMX-abort shutdown state. The following events do not affect a logical processor in this state: machine-check events; INIT signals; external interrupts; non-maskable interrupts (NMIs); start-up IPIs (SIPIs); and system-management interrupts (SMIs).

### 28.8 MACHINE-CHECK EVENTS DURING VM EXIT

If a machine-check event occurs during VM exit, one of the following occurs:

- The machine-check event is handled as if it occurred before the VM exit:
  - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating “unrecoverable machine-check condition.”
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If CR4.MCE = 1, a machine-check exception (#MC) is generated:
    - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
    - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- The machine-check event is handled after VM exit completes:
  - If the VM exit ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 7, “Safer Mode Extensions Reference,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.
VM EXITS

- If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).
- If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If the VM exit ends with CR4.MCE = 1, a machine-check exception (#MC) is delivered through the host IDT.
- A VMX abort is generated (see Section 28.7). The logical processor blocks events as done normally in VMX abort. The VMX abort indicator is 5, for “machine-check event during VM exit.”

The first option is not used if the machine-check event occurs after any host state has been loaded. The second option is used only if VM entry is able to load all host state.

28.9 USER-INTERRUPT RECOGNITION AFTER VM EXIT

A VM exit results in recognition of a pending user interrupt if it completes with CR4.UINTR = IA32_EFER.LMA = 1 and with UIRR \neq 0; otherwise, no pending user interrupt is recognized.
The architecture for VMX operation includes two features that support address translation: virtual-processor identifiers (VPIDs) and the extended page-table mechanism (EPT). VPIIDs are a mechanism for managing translations of linear addresses. EPT defines a layer of address translation that augments the translation of linear addresses.

Section 29.1 details the architecture of VPIDs. Section 29.3 provides the details of EPT. Section 29.4 explains how a logical processor may cache information from the paging structures, how it may use that cached information, and how software can managed the cached information.

### 29.1 VIRTUAL PROCESSOR IDENTIFIERS (VPIDS)

The original architecture for VMX operation required VMX transitions to flush the TLBs and paging-structure caches. This ensured that translations cached for the old linear-address space would not be used after the transition.

Virtual-processor identifiers (VPIDs) introduce to VMX operation a facility by which a logical processor may cache information for multiple linear-address spaces. When VPIDs are used, VMX transitions may retain cached information and the logical processor switches to a different linear-address space.

Section 29.4 details the mechanisms by which a logical processor manages information cached for multiple address spaces. A logical processor may tag some cached information with a 16-bit VPID. This section specifies how the current VPID is determined at any point in time:

- The current VPID is 0000H in the following situations:
  - Outside VMX operation. (This includes operation in system-management mode under the default treatment of SMIs and SMM with VMX operation; see Section 32.14.)
  - In VMX root operation.
  - In VMX non-root operation when the “enable VPID” VM-execution control is 0.
- If the logical processor is in VMX non-root operation and the “enable VPID” VM-execution control is 1, the current VPID is the value of the VPID VM-execution control field in the VMCS. (VM entry ensures that this value is never 0000H; see Section 27.2.1.1.)

VPIDs and PCIDs (see Section 4.10.1) can be used concurrently. When this is done, the processor associates cached information with both a VPID and a PCID. Such information is used only if the current VPID and PCID both match those associated with the cached information.

### 29.2 HYPERVISOR-MANAGED LINEAR-ADDRESS TRANSLATION (HLAT)

Hypervisor-managed linear-address translation (HLAT) is a feature that changes the way in which linear addresses are translated in VMX non-root operation. Instead of ordinary paging, translation uses a modified process called HLAT paging.

HLAT paging is used only if the “enable HLAT” VM-execution control is 1. HLAT paging is used only for the paging modes 4-level paging and 5-level paging. Because HLAT paging is a modification of ordinary paging, details of the feature are given in Section 4.5, which describes the operation of 4-level paging and 5-level paging.

### 29.3 THE EXTENDED PAGE TABLE MECHANISM (EPT)

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain addresses that would normally be treated as physical addresses (and used to access memory) are instead treated as guest-physical addresses. Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.
VMX SUPPORT FOR ADDRESS TRANSLATION

• Section 29.3.1 gives an overview of EPT.
• Section 29.3.2 describes operation of EPT-based address translation.
• Section 29.3.3 discusses VM exits that may be caused by EPT.
• Section 29.3.7 describes interactions between EPT and memory typing.

29.3.1 EPT Overview

EPT is used when the "enable EPT" VM-execution control is 1. It translates the guest-physical addresses used in VMX non-root operation and those used by VM entry for event injection.

The translation from guest-physical addresses to physical addresses is determined by a set of EPT paging structures. The EPT paging structures are similar to those used to translate linear addresses while the processor is in IA-32e mode. Section 29.3.2 gives the details of the EPT paging structures.

If CR0.PG = 1, linear addresses are translated through paging structures referenced through control register CR3. While the "enable EPT" VM-execution control is 1, these are called guest paging structures. There are no guest paging structures if CR0.PG = 0.

When the "enable EPT" VM-execution control is 1, the identity of guest-physical addresses depends on the value of CR0.PG:
• If CR0.PG = 0, each linear address is treated as a guest-physical address.
• If CR0.PG = 1, guest-physical addresses are those derived from the contents of control register CR3 and the guest paging structures. (This includes the values of the PDPTEs, which logical processors store in internal, non-architectural registers.) The latter includes (in page-table entries and in other paging-structure entries for which bit 7—PS—is 1) the addresses to which linear addresses are translated by the guest paging structures.

If CR0.PG = 1, the translation of a linear address to a physical address requires multiple translations of guest-physical addresses using EPT. Assume, for example, that CR4.PAE = CR4.PSE = 0. The translation of a 32-bit linear address then operates as follows:
• Bits 31:22 of the linear address select an entry in the guest page directory located at the guest-physical address in CR3. The guest-physical address of the guest page-directory entry (PDE) is translated through EPT to determine the guest PDE's physical address.
• Bits 21:12 of the linear address select an entry in the guest page table located at the guest-physical address in the guest PDE. The guest-physical address of the guest page-table entry (PTE) is translated through EPT to determine the guest PTE's physical address.
• Bits 11:0 of the linear address is the offset in the page frame located at the guest-physical address in the guest PTE. The guest-physical address determined by this offset is translated through EPT to determine the physical address to which the original linear address translates.

In addition to translating a guest-physical address to a physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called EPT violations and cause VM exits. See Section 29.3.3.

A processor uses EPT to translate guest-physical addresses only when those addresses are used to access memory. This principle implies the following:
• The MOV to CR3 instruction loads CR3 with a guest-physical address. Whether that address is translated through EPT depends on whether PAE paging is being used.

1. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, the logical processor operates as if the "enable EPT" VM-execution control were 0. See Section 25.6.2.
2. If HLAT paging is enabled, the processor uses the HLATP VMCS field instead of CR3. See Section 4.5.1. For simplicity, the remainder of this section refers only to CR3.
3. If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
4. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.
— If PAE paging is not being used, the instruction does not use that address to access memory and does not cause it to be translated through EPT. (If CR0.PG = 1, the address will be translated through EPT on the next memory accessing using a linear address.)

— If PAE paging is being used, the instruction loads the four (4) page-directory-pointer-table entries (PDPTEs) from that address and it does cause the address to be translated through EPT.

• Section 4.4.1 identifies executions of MOV to CR0 and MOV to CR4 that load the PDPTEs from the guest-physical address in CR3. Such executions cause that address to be translated through EPT.

• The PDPTEs contain guest-physical addresses. The instructions that load the PDPTEs (see above) do not use those addresses to access memory and do not cause them to be translated through EPT. The address in a PDPTE will be translated through EPT on the next memory accessing using a linear address that uses that PDPTE.

29.3.2 EPT Translation Mechanism

The EPT translation mechanism can be configured in either of two modes: 4-level EPT or 5-level EPT. 4-level EPT accesses at most 4 EPT paging-structure entries (an EPT page-walk length of 4) to translate a guest-physical address and uses only bits 47:0 of each guest-physical address. In contrast, 5-level EPT may access up to 5 EPT paging-structure entries (an EPT page-walk length of 5) and uses guest-physical address bits 56:0.1

The EPT page-walk length is configured using the extended-page-table pointer (EPTP), a VM-execution control field (see Table 25-9 in Section 25.6.11). Specifically, bits 5:3 contain a value one less than EPT page-walk length. Thus, a value of 3 configures 4-level EPT, while a value of 4 configures 5-level EPT.2

The remainder of this section describes the translation process used by 4-level EPT and 5-level EPT. Because the processes used by the two EPT modes are similar, they are described together in the following items (with any differences identified):

• With 5-level EPT, 4-KByte naturally aligned EPT PML5 table is located at the physical address specified in bits 51:12 of the EPTP. An EPT PML5 table comprises 512 64-bit entries (EPT PML5Es). An EPT PML5E is selected using the physical address defined as follows:
  — Bits 63:52 are all 0.
  — Bits 51:12 are from the EPTP.
  — Bits 11:3 are bits 56:48 of the guest-physical address.3
  — Bits 2:0 are all 0.

Because an EPT PML5E is identified using bits 56:48 of the guest-physical address, it controls access to a 256-TByte region of the guest-physical-address space. The format of an EPT PML5E is given in Table 29-1.

• With 4-level EPT, bits 51:48 of the guest-physical address must all be zero; otherwise, an EPT violation occurs (see Section 29.3.3).

• A 4-KByte naturally aligned EPT PML4 table is located at the physical address specified in the EPTP (for 4-level EPT) or in the EPT PML5E (for 5-level EPT). An EPT PML4 table comprises 512 64-bit entries (EPT PML4Es). An EPT PML4E is selected using the physical address defined as follows:
  — Bits 63:52 are all 0.
  — Bits 51:12 are from the EPTP (for 4-level EPT) or from bits 51:12 of the EPT PML4E (for 5-level EPT).
  — Bits 11:3 are bits 47:39 of the guest-physical address.
  — Bits 2:0 are all 0.

1. Physical addresses and guest-physical addresses are architecturally limited to 52 bits (e.g., by paging), so in practice bits 56:52 are zero.

2. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine what EPT page-walk lengths are supported.

3. Bits 56:52 of each guest-physical address are necessarily zero because guest-physical addresses are architecturally limited to 52 bits.
VMX SUPPORT FOR ADDRESS TRANSLATION

Because an EPT PML4E is identified using bits 47:39 of the guest-physical address, it controls access to a 512-GByte region of the guest-physical-address space. The format of an EPT PML4E is given in Table 29-2.

- A 4-KByte naturally aligned EPT page-directory-pointer table is located at the physical address specified in bits 51:12 of the EPT PML4E. An EPT page-directory-pointer table comprises 512 64-bit entries (EPT PDPTEs).
  - Bits 63:52 are all 0.
  - Bits 51:30 are from the EPT PML4E.
  - Bits 29:0 are from the original guest-physical address.

Because an EPT PDPTE is identified using bits 47:30 of the guest-physical address, it controls access to a 1-GByte region of the guest-physical-address space. Use of the EPT PDPTE depends on the value of bit 7 in that entry: 1

- If bit 7 of the EPT PDPTE is 1, the EPT PDPTE maps a 1-GByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:30 are from the EPT PDPTE.
  - Bits 29:0 are from the original guest-physical address.

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 256-TByte region controlled by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 256-TByte region controlled by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 256-TByte region controlled by this entry. If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 256-TByte region controlled by this entry.</td>
</tr>
<tr>
<td>7:3</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 256-TByte region controlled by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>Ignored.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the &quot;mode-based execute control for EPT&quot; VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 256-TByte region controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>(N–1):12</td>
<td>Physical address of 4-KByte aligned EPT PML4 table referenced by this entry. 1</td>
</tr>
<tr>
<td>51:N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>63:52</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>

NOTES:
1. N is the physical-address width supported by the processor. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

Because an EPT PML4E is identified using bits 47:39 of the guest-physical address, it controls access to a 512-GByte region of the guest-physical-address space. The format of an EPT PML4E is given in Table 29-2.

- A 4-KByte naturally aligned EPT page-directory-pointer table is located at the physical address specified in bits 51:12 of the EPT PML4E. An EPT page-directory-pointer table comprises 512 64-bit entries (EPT PDPTEs).
  - Bits 63:52 are all 0.
  - Bits 51:12 are from the EPT PML4E.
  - Bits 11:3 are bits 38:30 of the guest-physical address.
  - Bits 2:0 are all 0.

Because an EPT PDPTE is identified using bits 47:30 of the guest-physical address, it controls access to a 1-GByte region of the guest-physical-address space. Use of the EPT PDPTE depends on the value of bit 7 in that entry: 1

- If bit 7 of the EPT PDPTE is 1, the EPT PDPTE maps a 1-GByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:30 are from the EPT PDPTE.
  - Bits 29:0 are from the original guest-physical address.

1. Not all processors allow bit 7 of an EPT PDPTE to be set to 1. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine whether this is allowed.
VMX SUPPORT FOR ADDRESS TRANSLATION

Table 29-2. Format of an EPT PML4 Entry (PML4E) that References an EPT Page-Directory-Pointer Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 512-GByte region controlled by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 512-GByte region controlled by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the “mode-based execute control for EPT” VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 512-GByte region controlled by this entry.</td>
</tr>
<tr>
<td></td>
<td>If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 512-GByte region controlled by this entry.</td>
</tr>
<tr>
<td>7:3</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 512-GByte region controlled by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>Ignored.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 512-GByte region controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>(N–1):12</td>
<td>Physical address of 4-KByte aligned EPT page-directory-pointer table referenced by this entry.</td>
</tr>
<tr>
<td>51:N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>63:52</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>

NOTES:
1. N is the physical-address width supported by the processor.

The format of an EPT PDPTE that maps a 1-GByte page is given in Table 29-3.

- If bit 7 of the EPT PDPTE is 0, a 4-KByte naturally aligned EPT page directory is located at the physical address specified in bits 51:12 of the EPT PDPTE. The format of an EPT PDPTE that references an EPT page directory is given in Table 29-4.
Table 29-3. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-Byte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 1-Byte page referenced by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 1-Byte page referenced by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-Byte page controlled by this entry. If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-Byte page controlled by this entry.</td>
</tr>
<tr>
<td>5:3</td>
<td>EPT memory type for this 1-Byte page (see Section 29.3.7).</td>
</tr>
<tr>
<td>6</td>
<td>Ignore PAT memory type for this 1-Byte page (see Section 29.3.7).</td>
</tr>
<tr>
<td>7</td>
<td>Must be 1 (otherwise, this entry references an EPT page directory).</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-Byte page referenced by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 1-Byte page referenced by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the &quot;mode-based execute control for EPT&quot; VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-Byte page controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>29:12</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>(N−1):30</td>
<td>Physical address of the 1-Byte page referenced by this entry. ¹</td>
</tr>
<tr>
<td>51:N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>56:52</td>
<td>Ignored.</td>
</tr>
<tr>
<td>57</td>
<td>Verify guest paging. If the &quot;guest-paging verification&quot; VM-execution control is 1, indicates limits on the guest paging structures used to access the 1-Byte page controlled by this entry (see Section 29.3.3.2). If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>58</td>
<td>Paging-write access. If the &quot;EPT paging-write control&quot; VM-execution control is 1, indicates that guest paging may update the 1-Byte page controlled by this entry (see Section 29.3.3.2). If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>59</td>
<td>Ignored.</td>
</tr>
<tr>
<td>60</td>
<td>Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 1-Byte page mapped by this entry (see Section 29.3.3.2). Ignored if bit 7 of EPTP is 0.</td>
</tr>
<tr>
<td>62:61</td>
<td>Ignored.</td>
</tr>
<tr>
<td>63</td>
<td>Suppress #VE. If the &quot;EPT-violation #VE&quot; VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 26.5.7.1). If &quot;EPT-violation #VE&quot; VM-execution control is 0, this bit is ignored.</td>
</tr>
</tbody>
</table>

NOTES:
1. N is the physical-address width supported by the processor.
VMX SUPPORT FOR ADDRESS TRANSFORMATION

An EPT page-directory comprises 512 64-bit entries (PDEs). An EPT PDE is selected using the physical address defined as follows:

- Bits 63:52 are all 0.
- Bits 51:12 are from the EPT PDPTE.
- Bits 11:3 are bits 29:21 of the guest-physical address.
- Bits 2:0 are all 0.

Because an EPT PDE is identified using bits 47:21 of the guest-physical address, it controls access to a 2-MByte region of the guest-physical-address space. Use of the EPT PDE depends on the value of bit 7 in that entry:

- If bit 7 of the EPT PDE is 1, the EPT PDE maps a 2-MByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:21 are from the EPT PDE.
  - Bits 20:0 are from the original guest-physical address.

- If bit 7 of the EPT PDE is 0, a 4-KByte naturally aligned EPT page table is located at the physical address specified in bits 51:12 of the EPT PDE. The format of an EPT PDE that references an EPT page table is given in Table 29-6.

An EPT page table comprises 512 64-bit entries (PTEs). An EPT PTE is selected using a physical address defined as follows:

- Bits 63:52 are all 0.

Table 29-4. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that References an EPT Page Directory

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 1-GByte region controlled by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 1-GByte region controlled by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 1-GByte region controlled by this entry. If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 1-GByte region controlled by this entry.</td>
</tr>
<tr>
<td>7:3</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 1-GByte region controlled by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>Ignored.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the &quot;mode-based execute control for EPT&quot; VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 1-GByte region controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>(N-1):12</td>
<td>Physical address of 4-KByte aligned EPT page directory referenced by this entry.</td>
</tr>
<tr>
<td>51:N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>63:52</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>

NOTES:
1. N is the physical-address width supported by the processor.
### Table 29-5. Format of an EPT Page-Directory Entry (PDE) that Maps a 2-MByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 2-MByte page referenced by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 2-MByte page referenced by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte page controlled by this entry. If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte page controlled by this entry.</td>
</tr>
<tr>
<td>5:3</td>
<td>EPT memory type for this 2-MByte page (see Section 29.3.7).</td>
</tr>
<tr>
<td>6</td>
<td>Ignore PAT memory type for this 2-MByte page (see Section 29.3.7).</td>
</tr>
<tr>
<td>7</td>
<td>Must be 1 (otherwise, this entry references an EPT page table).</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the &quot;mode-based execute control for EPT&quot; VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte page controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>20:12</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>(N–1):21</td>
<td>Physical address of the 2-MByte page referenced by this entry.¹</td>
</tr>
<tr>
<td>51:N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>56:52</td>
<td>Ignored.</td>
</tr>
<tr>
<td>57</td>
<td>Verify guest paging. If the “guest-paging verification” VM-execution control is 1, indicates limits on the guest paging structures used to access the 2-MByte page controlled by this entry (see Section 29.3.3.2). If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>58</td>
<td>Paging-write access. If the “EPT paging-write control” VM-execution control is 1, indicates that guest paging may update the 2-MByte page controlled by this entry (see Section 29.3.3.2). If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>59</td>
<td>Ignored.</td>
</tr>
<tr>
<td>60</td>
<td>Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 2-MByte page mapped by this entry (see Section 29.3.3.2). Ignored if bit 7 of EPTP is 0.</td>
</tr>
<tr>
<td>62:61</td>
<td>Ignored.</td>
</tr>
<tr>
<td>63</td>
<td>Suppress #VE. If the “EPT-violation #VE” VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 26.5.7.1). If “EPT-violation #VE” VM-execution control is 0, this bit is ignored.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. N is the physical-address width supported by the processor.

   — Bits 51:12 are from the EPT PDE.
Bits 11:3 are bits 20:12 of the guest-physical address.
Bits 2:0 are all 0.

Because an EPT PTE is identified using bits 47:12 of the guest-physical address, every EPT PTE maps a 4-KByte page. The final physical address is computed as follows:

- Bits 63:52 are all 0.
- Bits 51:12 are from the EPT PTE.
- Bits 11:0 are from the original guest-physical address.

The format of an EPT PTE is given in Table 29-7.

An EPT paging-structure entry is present if any of bits 2:0 is 1; otherwise, the entry is not present. The processor ignores bits 62:3 and uses the entry neither to reference another EPT paging-structure entry nor to produce a physical address. A reference using a guest-physical address whose translation encounters an EPT paging-structure that is not present causes an EPT violation (see Section 29.3.3.2). (If the "EPT-violation #VE" VM-execution control is 1, the EPT violation is convertible to a virtualization exception only if bit 63 is 0; see Section 26.5.7.1. If the "EPT-violation #VE" VM-execution control is 0, this bit is ignored.)

Table 29-6. Format of an EPT Page-Directory Entry (PDE) that References an EPT Page Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 2-MByte region controlled by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 2-MByte region controlled by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 2-MByte region controlled by this entry. If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 2-MByte region controlled by this entry.</td>
</tr>
<tr>
<td>6:3</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>7</td>
<td>Must be 0 (otherwise, this entry maps a 2-MByte page).</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 2-MByte region controlled by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>Ignored.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the &quot;mode-based execute control for EPT&quot; VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 2-MByte region controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>(N−1):12</td>
<td>Physical address of 4-KByte aligned EPT page table referenced by this entry.</td>
</tr>
<tr>
<td>51:N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>63:52</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>

NOTES:
1. N is the physical-address width supported by the processor.

NOTE
If the "mode-based execute control for EPT" VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 or bit 10 is 1. If bits 2:0 are all 0 but bit 10 is 1, the entry is used normally to reference another EPT paging-structure entry or to produce a physical address.
The discussion above describes how the EPT paging structures reference each other and how the logical processor traverses those structures when translating a guest-physical address. It does not cover all details of the translation process. Additional details are provided as follows:

- Situations in which the translation process may lead to VM exits (sometimes before the process completes) are described in Section 29.3.3.
- Interactions between the EPT translation mechanism and memory typing are described in Section 29.3.7.

Figure 29-1 gives a summary of the formats of the EPTP and the EPT paging-structure entries. For the EPT paging structure entries, it identifies separately the format of entries that map pages, those that reference other EPT paging structures, and those that do neither because they are not present; bits $2:0$ and bit $7$ are highlighted because they determine how a paging-structure entry is used. (Figure 29-1 does not comprehend the fact that, if the "mode-based execute control for EPT" VM-execution control is 1, an entry is present if any of bits $2:0$ or bit $10$ is 1.)

### 29.3.3 EPT-Induced VM Exits

Accesses using guest-physical addresses may cause VM exits due to EPT misconfigurations, EPT violations, and page-modification log-full events. An **EPT misconfiguration** occurs when, in the course of translating a guest-physical address, the logical processor encounters an EPT paging-structure entry that contains an unsupported value (see Section 29.3.3.1). An **EPT violation** occurs when there is no EPT misconfiguration but the EPT paging-structure entries disallow an access using the guest-physical address (see Section 29.3.3.2). A **page-modification log-full event** occurs when the logical processor determines a need to create a page-modification log entry and the current log is full (see Section 29.3.6).

These events occur only due to an attempt to access memory with a guest-physical address. Loading CR3 with a guest-physical address with the MOVC to CR3 instruction can cause neither an EPT configuration nor an EPT violation until that address is used to access a paging structure. If the “EPT-violation #VE” VM-execution control is 1, certain EPT violations may cause virtualization exceptions instead of VM exits. See Section 26.5.7.1.

#### 29.3.3.1 EPT Misconfigurations

An EPT misconfiguration occurs if translation of a guest-physical address encounters an EPT paging-structure entry that meets any of the following conditions:

- Bit 0 of the entry is clear (indicating that data reads are not allowed) and any of the following hold:
  - Bit 1 is set (indicating that data writes are allowed).
  - The processor does not support execute-only translations and either of the following hold:
    - Bit 2 is set (indicating that instruction fetches are allowed).\(^2\)
    - The "mode-based execute control for EPT" VM-execution control is 1 and bit 10 is set (indicating that instruction fetches are allowed from user-mode linear addresses).
  - The “EPT paging-write control” VM-execution control is 1, the entry maps a page, and bit 58 is set (indicating that paging writes are allowed).
- The entry is present (see Section 29.3.2) and of the following holds:

---

1. If the logical processor is using PAE paging—because CR0.PG = CR4.PAE = 1 and IA32_EFER.LMA = 0—the MOVC to CR3 instruction loads the PDPTes from memory using the guest-physical address being loaded into CR3. In this case, therefore, the MOVC to CR3 instruction may cause an EPT misconfiguration, an EPT violation, or a page-modification log-full event.
2. If the "mode-based execute control for EPT" VM-execution control is 1, setting bit 2 indicates that instruction fetches are allowed from supervisor-mode linear addresses.
Table 29-7. Format of an EPT Page-Table Entry that Maps a 4-KByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read access; indicates whether reads are allowed from the 4-KByte page referenced by this entry.</td>
</tr>
<tr>
<td>1</td>
<td>Write access; indicates whether writes are allowed to the 4-KByte page referenced by this entry.</td>
</tr>
<tr>
<td>2</td>
<td>If the &quot;mode-based execute control for EPT&quot; VM-execution control is 0, execute access; indicates whether instruction fetches are allowed from the 4-KByte page controlled by this entry. If that control is 1, execute access for supervisor-mode linear addresses; indicates whether instruction fetches are allowed from supervisor-mode linear addresses in the 4-KByte page controlled by this entry.</td>
</tr>
<tr>
<td>5:3</td>
<td>EPT memory type for this 4-KByte page (see Section 29.3.7).</td>
</tr>
<tr>
<td>6</td>
<td>Ignore PAT memory type for this 4-KByte page (see Section 29.3.7).</td>
</tr>
<tr>
<td>7</td>
<td>Ignored.</td>
</tr>
<tr>
<td>8</td>
<td>If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>9</td>
<td>If bit 6 of EPTP is 1, dirty flag for EPT; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 29.3.5). Ignored if bit 6 of EPTP is 0.</td>
</tr>
<tr>
<td>10</td>
<td>Execute access for user-mode linear addresses. If the &quot;mode-based execute control for EPT&quot; VM-execution control is 1, indicates whether instruction fetches are allowed from user-mode linear addresses in the 4-KByte page controlled by this entry. If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>11</td>
<td>Ignored.</td>
</tr>
<tr>
<td>(N-1):12</td>
<td>Physical address of the 4-KByte page referenced by this entry.¹</td>
</tr>
<tr>
<td>51:5N</td>
<td>Reserved (must be 0).</td>
</tr>
<tr>
<td>56:52</td>
<td>Ignored.</td>
</tr>
<tr>
<td>57</td>
<td>Verify guest paging. If the &quot;guest-paging verification&quot; VM-execution control is 1, indicates limits on the guest paging structures used to access the 4-KByte page controlled by this entry (see Section 29.3.3.2). If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>58</td>
<td>Paging-write access. If the &quot;EPT paging-write control&quot; VM-execution control is 1, indicates that guest paging may update the 4-KByte page controlled by this entry (see Section 29.3.3.2). If that control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>59</td>
<td>Ignored.</td>
</tr>
<tr>
<td>60</td>
<td>Supervisor shadow stack. If bit 7 of EPTP is 1, indicates whether supervisor shadow stack accesses are allowed to guest-physical addresses in the 4-KByte page mapped by this entry (see Section 29.3.3.2). Ignored if bit 7 of EPTP is 0.</td>
</tr>
<tr>
<td>61</td>
<td>Sub-page write permissions. If the &quot;sub-page write permissions for EPT&quot; VM-execution control is 1, writes to individual 128-byte regions of the 4-KByte page referenced by this entry may be allowed even if the page would normally not be writable (see Section 29.3.4). If &quot;sub-page write permissions for EPT&quot; VM-execution control is 0, this bit is ignored.</td>
</tr>
<tr>
<td>62</td>
<td>Ignored.</td>
</tr>
<tr>
<td>63</td>
<td>Suppress #VE. If the &quot;EPT-violation #VE&quot; VM-execution control is 1, EPT violations caused by accesses to this page are convertible to virtualization exceptions only if this bit is 0 (see Section 26.5.7.1). If &quot;EPT-violation #VE&quot; VM-execution control is 0, this bit is ignored.</td>
</tr>
</tbody>
</table>

¹ Physical address of the 4-KByte page referenced by this entry.
VMX SUPPORT FOR ADDRESS TRANSLATION

NOTES:
1. \( N \) is the physical-address width supported by the processor.

   — A reserved bit is set. This includes the setting of a bit in the range 51:12 that is beyond the logical processor’s physical-address width.\(^1\) See Section 29.3.2 for details of which bits are reserved in which EPT paging-structure entries.

   — The entry is the last one used to translate a guest physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE) and the value of bits 5:3 (EPT memory type) is 2, 3, or 7 (these values are reserved).

EPT misconfigurations result when an EPT paging-structure entry is configured with settings reserved for future functionality. Software developers should be aware that such settings may be used in the future and that an EPT paging-structure entry that causes an EPT misconfiguration on one processor might not do so in the future.

29.3.3.2 EPT Violations

An EPT violation may occur during an access using a guest-physical address whose translation does not cause an EPT misconfiguration. An EPT violation occurs in any of the following situations:

- Translation of the guest-physical address encounters an EPT paging-structure entry that is not present (see Section 29.3.2).

- The access is a data read and, for any byte to be read, bit 0 (read access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte. Reads by the logical processor of guest paging structures to translate a linear address are considered to be data reads.

---

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
### Figure 29-1. Formats of EPTP and EPT Paging-Structure Entries

**NOTES:**

1. M is an abbreviation for MAXPHYADDR.
2. Supervisor shadow-stack control.
3. See Section 25.6.11 for details of the EPTP.
4. Execute access for user-mode linear addresses. If the “mode-based execute control for EPT” VM-execution control is 0, this bit is ignored.
5. Execute access. If the “mode-based execute control for EPT” VM-execution control is 1, this bit controls execute access for supervisor-mode linear addresses.
6. If the “mode-based execute control for EPT” VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 or bit 10 is 1. This table does not comprehend that fact.

7. Suppress #VE. If the “EPT-violation #VE” VM-execution control is 0, this bit is ignored.

8. Supervisor shadow-stack page. If bit 7 of the EPTP is 0, this bit is ignored.

9. Paging-write access. If the “EPT paging-write control” VM-execution control is 0, this bit is ignored.

10. Verify guest paging. If the “guest-paging verification” VM-execution control is 0, this bit is ignored.

11. Sub-page write permissions. If the “sub-page write permissions for EPT” VM-execution control is 0, this bit is ignored.

• The access is a data write and, for any byte to be written, bit 1 (write access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte. Writes by the logical processor to guest paging structures to update accessed and dirty flags are considered to be data writes.

If bit 6 of the EPT pointer (EPTP) is 1 (enabling accessed and dirty flags for EPT), processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations. Thus, if bit 1 is clear in any of the EPT paging-structure entries used to translate the guest-physical address of a guest paging-structure entry, an attempt to use that entry to translate a linear address causes an EPT violation.

(This does not apply to loads of the PDPTL registers by the MOV to CR instruction for PAE paging; see Section 4.4.1. Those loads of guest PDPTEs are treated as reads and do not cause EPT violations due to a guest-physical address not being writable.)

Processor writes to guest paging structures to update accessed and dirty flags are called **paging writes**. (When accessed and dirty flags for EPT are enabled all processor accesses to guest paging structures are considered paging writes.) If the “EPT paging-write control” VM-execution control is 1, clearing the write-access bit in the EPT paging-structure entry that maps a page does not prevent a paging write if bit 58 (paging-write access) in that entry is 1. (EPT paging-structure entries that reference other EPT paging structures do not use bit 58, and it remains the case that they must set the write-access bit for paging writes to be allowed.)

If the “sub-page write permissions for EPT” VM-execution control is 1, data writes to a guest-physical address that would cause an EPT violation (as indicated above) do not do so in certain situations. If the guest-physical address is mapped using a 4-KByte page and bit 61 (sub-page write permissions) of the EPT PTE used to map the page is 1, writes to certain 128-byte sub-pages may be allowed. See Section 29.3.4 for details.

• The access is an instruction fetch and the EPT paging structures prevent execute access to any of the bytes being fetched. Whether this occurs depends upon the setting of the “mode-based execute control for EPT” VM-execution control:

  — If the control is 0, an instruction fetch from a byte is prevented if bit 2 (execute access) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.

  — If the control is 1, an instruction fetch from a byte is prevented in either of the following cases:

    • Paging maps the linear address of the byte as a supervisor-mode address and bit 2 (execute access for supervisor-mode linear addresses) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.

    Paging maps a linear address as a supervisor-mode address if the U/S flag (bit 2) is 0 in at least one of the paging-structure entries controlling the translation of the linear address.

    • Paging maps the linear address of the byte as a user-mode address and bit 10 (execute access for user-mode linear addresses) was clear in any of the EPT paging-structure entries used to translate the guest-physical address of the byte.

    Paging maps a linear address as a user-mode address if the U/S flag is 1 in all of the paging-structure entries controlling the translation of the linear address. If paging is disabled (CR0.PG = 0), every linear address is a user-mode address.

• If supervisor shadow-stack control is enabled (by setting bit 7 of EPTP), the access is a supervisor shadow-stack access, and the EPT paging-structure entries used to translate the guest-physical address of the access disallow supervisor shadow-stack accesses. Such an access is disallowed if any of the following hold:

  — Bit 0 (read access) is clear in any EPT paging-structure entry used to translate the guest-physical address of the access.
VMX SUPPORT FOR ADDRESS TRANSLATION

— Bit 1 (write access) is clear in any EPT paging-structure entry that references an EPT paging structure in the translation of the guest-physical address. (Clearing bit 1 in the EPT paging-structure entry that maps the page of the guest-physical address does not disallow shadow-stack reads and writes.)

— Bit 60 (supervisor-shadow stack access) is clear in the EPT paging-structure entry that maps the page of the guest-physical address.

Supervisor shadow-stack control and the supervisor-shadow stack access bits in EPT paging-structure entries do not affect other accesses (including user shadow-stack accesses).

• If the “guest-paging verification” and “EPT paging-write control” VM-execution controls are both 1, guest-paging verification is performed for certain accesses related to translation of a linear address.

Specifically, guest-paging verification may apply to an access using a guest-physical address to the guest paging-structure entry that maps a page for the linear address (see Chapter 4, "Paging"). It applies if bit 57 (verify guest paging) is set in the EPT paging-structure entry that maps a page for that guest-physical address.

When guest-paging verification occurs, there is an EPT violation if bit 58 (paging-write access) is clear in the EPT paging-structure entry that was used to map a page for the guest-physical address of any guest paging-structure entry that was used during translation of the original linear address.¹

See Section 29.3.3.3 for details regarding the prioritization of such EPT violations relative to any page faults that may occur due to the original reference to the linear address being translated.

29.3.3.3  Prioritization of EPT Misconfigurations and EPT Violations

The translation of a linear address to a physical address requires one or more translations of guest-physical addresses using EPT (see Section 29.3.1). This section specifies the relative priority of EPT-induced VM exits with respect to each other and to other events that may be encountered when accessing memory using a linear address.

For an access to a guest-physical address, determination of whether an EPT misconfiguration or an EPT violation occurs is based on an iterative process:²

1. An EPT paging-structure entry is read (initially, this is an EPT PML5 entry or an EPT PML4 entry):
   a. If the entry is not present (see Section 29.3.2), an EPT violation occurs.
   b. If the entry is present but its contents are not configured properly (see Section 29.3.3.1), an EPT misconfiguration occurs.
   c. If the entry is present and its contents are configured properly, operation depends on whether the entry references another EPT paging structure (or if instead it is an EPT PDPTE or EPT PDE with bit 7 set to 1, or an EPT PTE):
      i) If the entry does reference another EPT paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
      ii) Otherwise, the entry is used to produce the ultimate physical address (the translation of the original guest-physical address); step 2 is executed.

2. Once the ultimate physical address is determined, the privileges determined by the EPT paging-structure entries are evaluated:
   a. If the access to the guest-physical address is not allowed by these privileges (see Section 29.3.3.2), an EPT violation occurs.
   b. If the access to the guest-physical address is allowed by these privileges, memory is accessed using the ultimate physical address.

If CR0.PG = 1, the translation of a linear address is also an iterative process, with the processor first accessing an entry in the guest paging structure referenced by the guest-physical address in CR3,³ then accessing an entry in

¹ When there is a restart of HLAT paging (see Section 4.5), the guest paging-structure entries used by HLAT paging (prior to the restart) are not relevant to guest-paging verification.

² This is a simplification of the more detailed description given in Section 29.3.2.

³ If PAE paging is in use, the processor instead uses the guest-physical address in the appropriate PDPTE register. If HLAT paging is enabled, the processor instead uses the HLATP VMCS field. For simplicity, the remainder of this section refers only to CR3.
another guest paging structure referenced by the guest-physical address in the first guest paging-structure entry, etc. Each guest-physical address is itself translated using EPT and may cause an EPT-induced VM exit. The following items detail how page faults and EPT-induced VM exits are recognized during this iterative process:

1. An attempt is made to access a guest paging-structure entry with a guest-physical address (initially, the address in CR3).
   a. If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.
   b. If the access does not cause an EPT-induced VM exit, the translation continues. If guest-paging verification is enabled (see Section 29.3.3.2), the processor notes whether the EPT paging-structure entry used to map a page for the guest-physical address set bit 58 (paging write). Then, bit 0 (the present flag) of the guest paging-structure entry is consulted:
      i) If the present flag is 0 or any reserved bit is set, a page fault occurs.
      ii) If the present flag is 1, no reserved bit is set, operation depends on whether the entry references another guest paging structure (whether it is a guest PDE with PS = 1 or a guest PTE):
          • If the entry does reference another guest paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
          • Otherwise, the entry is used to produce the ultimate guest-physical address (the translation of the original linear address); step 2 is executed.

2. Once the ultimate guest-physical address is determined, the privileges determined by the guest paging-structure entries are evaluated:
   a. If the access to the linear address is not allowed by these privileges (e.g., it was a write to a read-only page), a page fault occurs.
   b. If the access to the linear address is allowed by these privileges, an attempt is made to access memory at the ultimate guest-physical address:
      i) If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs. It is at this point that guest-paging verification occurs (if enabled), using the paging-write bits that were noted at occurrences of step 1b above (see Section 29.3.3.2 for details of guest-paging verification).
      ii) If the access does not cause an EPT-induced VM exit, memory is accessed using the ultimate physical address (the translation, using EPT, of the ultimate guest-physical address).

If CR0.PG = 0, a linear address is treated as a guest-physical address and is translated using EPT (see above). This process, if it completes without an EPT violation or EPT misconfiguration, produces a physical address and determines the privileges allowed by the EPT paging-structure entries. If these privileges do not allow the access to the physical address (see Section 29.3.3.2), an EPT violation occurs. Otherwise, memory is accessed using the physical address.

### 29.3.4 Sub-Page Write Permissions

Section 29.3.3.2 explained how EPT enforces the access rights for guest-physical addresses using EPT violations. Since these access rights are determined using the EPT paging-structure entries that are used to translate a guest-physical address, their granularity is limited to that which is used to map pages (1-GByte, 2-MByte, or 4-KByte).

The **sub-page write-permission** feature allows the control of write accesses to guest-physical addresses to be controlled at finer granularity. Sub-page write permissions allow write accesses to be controlled at the granularity of naturally aligned 128-byte **sub-pages**. Specifically, the feature allows writes to selected sub-pages of 4-KByte page that would otherwise not be writable.

Sub-page write permissions are enabled setting the "sub-page write permissions for EPT" VM-execution control to 1. The remainder of this section describes changes to processor operation with this control setting.

Section 29.3.4.1 identifies the data accesses that are eligible for sub-page write permissions. Section 29.3.4.2 explains how the processor determines whether to allow such an access.
29.3.4.1 Write Accesses That Are Eligible for Sub-Page Write Permissions

A guest-physical address is eligible for sub-page write permissions if writes to it would be disallowed following Section 29.3.3.2: bit 1 (write access) is clear in any of the EPT paging-structure entries used to translate the guest-physical address. Guest-physical addresses to which writes would be disallowed for other reasons (e.g., the translation encounters an EPT paging-structure entry that is not present) are not eligible for sub-page write permissions.

In addition, a guest-physical address is eligible for sub-page write permissions only if it is mapped using a 4-KByte page and bit 61 (sub-page write permissions) of the EPT PTE used to map the page is 1. (Guest-physical addresses mapped with larger pages are not eligible for sub-page write permissions.)

For some memory accesses, the processor ignores bit 61 in an EPT PTE used to map a 4-KByte page and does not apply sub-page write permissions to the access. (In such a case, the access causes an EPT violation when indicated by the conditions given in Section 29.3.3.2.) Sub-page write permissions never apply to the following accesses:

- A write access performed within a transactional region.
- A write access by an enclave to an address within the enclave’s ELRANGE. (Sub-page write permissions may apply to write accesses by enclaves to addresses outside its ELRANGE.)
- A write access to the enclave page cache (EPC) by an Intel SGX instruction.
- A write access to a guest paging structure to update an accessed or dirty flag.
- Processor accesses to guest paging-structure entries when accessed and dirty flags for EPT are enabled (such accesses are treated as writes with regard to EPT violations).

There are additional accesses to which sub-page write permissions might not be applied (behavior is model-specific). The following items enumerate examples:

- A write access that crosses two 4-KByte pages. In this case, sub-page permissions may be applied to neither or to only one of the pages. (There is no write to either page unless the write is allowed to both pages.)
- A write access by an instruction that performs multiple write accesses (sub-page write permissions are intended principally for basic instructions such as AND, MOV, OR, TEST, XCHG, and XOR).

If a guest-physical address is eligible for sub-page write permissions, the processor determines whether to allow write to the address using the process described in Section 29.3.4.2.

If a guest-physical address is eligible for sub-page write permissions and that address translates to an address on the APIC-access page (see Section 30.4), the processor may treat a write access to the address as if the "virtualize APIC accesses” VM-execution control were 0. For that reason, it is recommended that software not configure any guest-physical address that translates to an address on the APIC-access page to be eligible for sub-page write permissions.

29.3.4.2 Determining an Access’s Sub-Page Write Permission

Sub-page write permissions control write accesses individually to each of the 32 128-byte sub-pages of a 4-KByte page. Bits 11:7 of guest-physical address identify the sub-page.

For each guest-physical address eligible for sub-page write permissions, there is a 64-bit sub-page permission vector (SPP vector). All addresses on a 4-KByte page use the same SPP vector. If an address’s sub-page number (bits 11:7 of the address) is S, writes to address are allowed if and only if bit 2S of the sub-page permission is set to 1. (The bits at odd positions in a SPP vector are not used and must be zero.)

Each page’s SPP vector is located in memory. For a write to a guest-physical address eligible for sub-page write permissions, the processor uses the following process to locate the address’s SPP vector:

1. The SPPTP (sub-page-permission-table pointer) VM-execution control field contains the physical address of the 4-KByte root SPP table (SSPL4 table). Bits 47:39 of the guest-physical address identify a 64-bit entry in that table, called an SPPL4E.
2. A 4-KByte SPPL3 table is located at the physical address in the selected SPPL4E. Bits 38:30 of the guest-physical address identify a 64-bit entry in that table, called an SPPL3E.
3. A 4-KByte SPPL2 table is located at the physical address in the selected SPPL3E. Bits 29:21 of the guest-physical address identify a 64-bit entry in that table, called an SPPL2E.
A 4-KByte SPP-vector table (SSPL1 table) is located at the physical address in the selected SPPL2E. Bits 20:12 of the guest-physical address identify the 64-bit SPP vector for the address. As noted earlier, bit 2S of the sub-page permission vector determines whether the address may be written, where S is the value of address bits 11:7.

(The memory type used to access these tables is reported in bits 53:50 of the IA32_VMX_BASIC MSR. See Appendix A.1.)

A write access to multiple 128-byte sub-pages on a single 4-KByte page is allowed only if the indicated bit in the page's SPP vector for each of those sub-pages is set to 1. The following items apply to cases in which an access writes to two 4-KByte pages:

- If a write to either page would be disallowed according to Section 29.3.3.2, the access might be disallowed even if the guest-physical address of that page is eligible for sub-page write permissions. (This behavior is model-specific.)
- The access is allowed only if, for each page, either (1) a write to the page would be allowed following Section 29.3.3.2; or (2) both (a) the guest-physical address of that page is eligible for sub-page write permissions; and (b) the page's sub-page vector allows the write (as described above).

Bit 0 of each entry (SPPL4E, SPPL3E, or SPPL2E) is the entry's valid bit. If the process above accesses an entry in which this bit is 0, the process stops and the logical processor incurs an SPP miss.

In each entry (SPPL4E, SPPL3E, or SPPL2E), bits 11:1 are reserved, as are bits 63:N, where N is the processor's physical-address width. If the process above accesses an entry in which the valid bit is 1 and in which some reserved bit is set, the process stops and the logical processor incurs an SPP misconfiguration. Bits in an SPP vector in odd positions are also reserved; an SPP misconfiguration occurs also any of those bits are set in the final SPP vector.

SPP misses and SPP misconfigurations are called SPP-related events and cause VM exits.

29.3.5 Accessed and Dirty Flags for EPT

The Intel 64 architecture supports accessed and dirty flags in ordinary paging-structure entries (see Section 4.8). Some processors also support corresponding flags in EPT paging-structure entries. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10) to determine whether the processor supports this feature.

Software can enable accessed and dirty flags for EPT using bit 6 of the extended-page-table pointer (EPTP), a VM-execution control field (see Table 25-9 in Section 25.6.11). If this bit is 1, the processor will set the accessed and dirty flags for EPT as described below. In addition, setting this flag causes processor accesses to guest paging-structure entries to be treated as writes (see below and Section 29.3.3.2).

For any EPT paging-structure entry that is used during guest-physical-address translation, bit 8 is the accessed flag. For a EPT paging-structure entry that maps a page (as opposed to referencing another EPT paging structure), bit 9 is the dirty flag.

Whenever the processor uses an EPT paging-structure entry as part of guest-physical-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a guest-physical address, the processor sets the dirty flag (if it is not already set) in the EPT paging-structure entry that identifies the final physical address for the guest-physical address (either an EPT PTE or an EPT paging-structure entry in which bit 7 is 1).

When accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes (see Section 29.3.3.2). Thus, such an access will cause the processor to set the dirty flag in the EPT paging-structure entry that identifies the final physical address of the guest paging-structure entry.

(This does not apply to loads of the PDPTEs registers for PAE paging by the MOV to CR instruction; see Section 4.4.1. Those loads of guest PDPTEs are treated as reads and do not cause the processor to set the dirty flag in any EPT paging-structure entry.)

These flags are “sticky,” meaning that, once set, the processor does not clear them; only software can clear them.
A processor may cache information from the EPT paging-structure entries in TLBs and paging-structure caches (see Section 29.4). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the processor might not set the corresponding bit in memory on a subsequent access using an affected guest-physical address.

### 29.3.6 Page-Modification Logging

When accessed and dirty flags for EPT are enabled, software can track writes to guest-physical addresses using a feature called **page-modification logging**.

Software can enable page-modification logging by setting the "enable PML" VM-execution control (see Table 25-7 in Section 25.6.2). When this control is 1, the processor adds entries to the **page-modification log** as described below. The page-modification log is a 4-KByte region of memory located at the physical address in the PML address VM-execution control field. The page-modification log consists of 512 64-bit entries; the PML index VM-execution control field indicates the next entry to use.

Before allowing a guest-physical access, the processor may determine that it first needs to set an accessed or dirty flag for EPT (see Section 29.3.5). When this happens, the processor examines the PML index. If the PML index is not in the range 0–511, there is a **page-modification log-full event** and a VM exit occurs. In this case, the accessed or dirty flag is not set, and the guest-physical access that triggered the event does not occur.

If instead the PML index is in the range 0–511, the processor proceeds to update accessed or dirty flags for EPT as described in Section 29.3.5. If the processor updated a dirty flag for EPT (changing it from 0 to 1), it then operates as follows:

1. The guest-physical address of the access is written to the page-modification log. Specifically, the guest-physical address is written to physical address determined by adding 8 times the PML index to the PML address. Bits 11:0 of the value written are always 0 (the guest-physical address written is thus 4-KByte aligned).
2. The PML index is decremented by 1 (this may cause the value to transition from 0 to FFFFH).

Because the processor decrements the PML index with each log entry, the value may transition from 0 to FFFFH. At that point, no further logging will occur, as the processor will determine that the PML index is not in the range 0–511 and will generate a page-modification log-full event (see above).

### 29.3.7 EPT and Memory Typing

This section specifies how a logical processor determines the memory type use for a memory access while EPT is in use. (See Chapter 12, "Memory Cache Control," of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for details of memory typing in the Intel 64 architecture.) Section 29.3.7.1 explains how the memory type is determined for accesses to the EPT paging structures. Section 29.3.7.2 explains how the memory type is determined for an access using a guest-physical address that is translated using EPT.

#### 29.3.7.1 Memory Type Used for Accessing EPT Paging Structures

This section explains how the memory type is determined for accesses to the EPT paging structures. The determination is based first on the value of bit 30 (cache disable—CD) in control register CR0:

- If CR0.CD = 0, the memory type used for any such reference is the EPT paging-structure memory type, which is specified in bits 2:0 of the extended-page-table pointer (EPTP), a VM-execution control field (see Section 25.6.11). A value of 0 indicates the uncacheable type (UC), while a value of 6 indicates the write-back type (WB). Other values are reserved.
- If CR0.CD = 1, the memory type used for any such reference is uncacheable (UC).

The MTRRs have no effect on the memory type used for an access to an EPT paging structure.

#### 29.3.7.2 Memory Type Used for Translated Guest-Physical Addresses

The **effective memory type** of a memory access using a guest-physical address (an access that is translated using EPT) is the memory type that is used to access memory. The effective memory type is based on the value of
bit 30 (cache disable—CD) in control register CR0; the last EPT paging-structure entry used to translate the guest-physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE); and the PAT memory type (see below):

- The **PAT memory type** depends on the value of CR0.PG:
  - If CR0.PG = 0, the PAT memory type is WB (writeback).
  
- If CR0.PG = 1, the PAT memory type is the memory type selected from the IA32_PAT MSR as specified in Section 12.12.3, "Selecting a Memory Type from the PAT."

- The **EPT memory type** is specified in bits 5:3 of the last EPT paging-structure entry: 0 = UC; 1 = WC; 4 = WT; 5 = WP; and 6 = WB. Other values are reserved and cause EPT misconfigurations (see Section 29.3.3).

- If CR0.CD = 0, the effective memory type depends upon the value of bit 6 of the last EPT paging-structure entry:
  - If the value is 0, the effective memory type is the combination of the EPT memory type and the PAT memory type specified in Table 12-7 in Section 12.5.2.2, using the EPT memory type in place of the MTRR memory type.
  - If the value is 1, the memory type used for the access is the EPT memory type. The PAT memory type is ignored.

- If CR0.CD = 1, the effective memory type is UC.

The MTRRs have no effect on the memory type used for an access to a guest-physical address.

## 29.4 CACHING TRANSLATION INFORMATION

Processors supporting Intel® 64 and IA-32 architectures may accelerate the address-translation process by caching on the processor data from the structures in memory that control that process. Such caching is discussed in Section 4.10, "Caching Translation Information," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. The current section describes how this caching interacts with the VMX architecture.

The VPID and EPT features of the architecture for VMX operation augment this caching architecture. EPT defines the guest-physical address space and defines translations to that address space (from the linear-address space) and from that address space (to the physical-address space). Both features control the ways in which a logical processor may create and use information cached from the paging structures.

Section 29.4.1 describes the different kinds of information that may be cached. Section 29.4.2 specifies when such information may be cached and how it may be used. Section 29.4.3 details how software can invalidate cached information.

### 29.4.1 Information That May Be Cached

Section 4.10, "Caching Translation Information," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, identifies two kinds of translation-related information that may be cached by a logical processor: **translations**, which are mappings from linear page numbers to physical page frames, and **paging-structure caches**, which map the upper bits of a linear page number to information from the paging-structure entries used to translate linear addresses matching those upper bits.

The same kinds of information may be cached when VPIDs and EPT are in use. A logical processor may cache and use such information based on its function. Information with different functionality is identified as follows:

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

2. Table 12-11 in Section 12.12.3, “Selecting a Memory Type from the PAT,” illustrates how the PAT memory type is selected based on the values of the PAT, PCD, and PWT bits in a page-table entry (or page-directory entry with PS = 1). For accesses to a guest paging-structure entry X, the PAT memory type is selected from the table by using a value of 0 for the PAT bit with the values of PCD and PWT from the paging-structure entry Y that references X (or from CR3 if X is in the root paging structure). With PAE paging, the PAT memory type for accesses to the PDPTes is WB.
• **Linear mappings.** There are two kinds:
  — Linear translations. Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
  — Linear paging-structure-cache entries. Each of these is a mapping from the upper portion of a linear address to the physical address of the paging structure used to translate the corresponding region of the linear-address space, along with information about access privileges. For example, bits 47:39 of a linear address would map to the address of the relevant page-directory-pointer table.

Linear mappings do not contain information from any EPT paging structure.

• **Guest-physical mappings.** There are two kinds:
  — Guest-physical translations. Each of these is a mapping from a guest-physical page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
  — Guest-physical paging-structure-cache entries. Each of these is a mapping from the upper portion of a guest-physical address to the physical address of the EPT paging structure used to translate the corresponding region of the guest-physical address space, along with information about access privileges.

The information in guest-physical mappings about access privileges and memory typing is derived from EPT paging structures.

• **Combined mappings.** There are two kinds:
  — Combined translations. Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing.
  — Combined paging-structure-cache entries. Each of these is a mapping from the upper portion of a linear address to the physical address of the paging structure used to translate the corresponding region of the linear-address space, along with information about access privileges.

The information in combined mappings about access privileges and memory typing is derived from both guest paging structures and EPT paging structures.

Guest-physical mappings and combined mappings may also include SPP vectors and information about the data structures used to locate SPP vectors (see Section 29.3.4.2).

### 29.4.2 Creating and Using Cached Translation Information

The following items detail the creation of the mappings described in the previous section:

• The following items describe the creation of mappings while EPT is not in use (including execution outside VMX non-root operation):
  — Linear mappings may be created. They are derived from the paging structures referenced (directly or indirectly) by the current value of CR3 and are associated with the current VPID and the current PCID.
  — No linear mappings are created with information derived from paging-structure entries that are not present (bit 0 is 0) or that set reserved bits. For example, if a PTE is not present, no linear mapping are created for any linear page number whose translation would use that PTE.
  — No guest-physical or combined mappings are created while EPT is not in use.
• The following items describe the creation of mappings while EPT is in use:
  — Guest-physical mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by bits 51:12 of the current EPTEP. These 40 bits contain the address of the EPT root

---

1. Earlier versions of this manual used the term “VPID-tagged” to identify linear mappings.
2. Earlier versions of this manual used the term “EPT-tagged” to identify guest-physical mappings.
3. Earlier versions of this manual used the term “dual-tagged” to identify combined mappings.
4. This section associated cached information with the current VPID and PCID. If PCIDs are not supported or are not being used (e.g., because CR4.PCIDE = 0), all the information is implicitly associated with PCID 000H; see Section 4.10.1, “Process-Context Identifiers (PCIDs),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
VMX SUPPORT FOR ADDRESS TRANSLATION

Table (a EPT PML4 table with 4-level EPT or a EPT PML5 table with 5-level EPT); the notation **EPTRTA** refers to those 40 bits. Newly created guest-physical mappings are associated with the current EPTRTA.

— Combined mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by the current EPTRTA. If CR0.PG = 1, they are also derived from the paging structures referenced (directly or indirectly) by the current value of CR3. They are associated with the current VPID, the current PCID, and the current EPTRTA.¹ No combined paging-structure-cache entries are created if CR0.PG = 0.²

— No guest-physical mappings or combined mappings are created with information derived from EPT paging-structure entries that are not present (see Section 29.3.2) or that are misconfigured (see Section 29.3.3.1).

— No combined mappings are created with information derived from guest paging-structure entries that are not present or that set reserved bits.

— No linear mappings are created while EPT is in use.

The following items detail the use of the various mappings:

• If EPT is not in use (e.g., when outside VMX non-root operation), a logical processor may use cached mappings as follows:
  — For accesses using linear addresses, it may use linear mappings associated with the current VPID and the current PCID. It may also use global TLB entries (linear mappings) associated with the current VPID and any PCID.
  — No guest-physical or combined mappings are used while EPT is not in use.

• If EPT is in use, a logical processor may use cached mappings as follows:
  — For accesses using linear addresses, it may use combined mappings associated with the current VPID, the current PCID, and the current EPTRTA. It may also use global TLB entries (combined mappings) associated with the current VPID, the current EPTRTA, and any PCID.
  — For accesses using guest-physical addresses, it may use guest-physical mappings associated with the current EPTRTA.
  — No linear mappings are used while EPT is in use.

29.4.3 Invalidating Cached Translation Information

Software modifications of paging structures (including EPT paging structures and the data structures used to locate SPP vectors) may result in inconsistencies between those structures and the mappings cached by a logical processor. Certain operations invalidate information cached by a logical processor and can be used to eliminate such inconsistencies.

29.4.3.1 Operations that Invalidate Cached Mappings

The following operations invalidate cached mappings as indicated:

• Operations that architecturally invalidate entries in the TLBs or paging-structure caches independent of VMX operation (e.g., the INVLPG and INVPCID instructions) invalidate linear mappings and combined mappings.³ They are required to do so only for the current VPID (but, for combined mappings, all EPTRTAs). Linear

---

¹. At any given time, a logical processor may be caching combined mappings for a VPID and a PCID that are associated with different EPTRTAs. Similarly, it may be caching combined mappings for an EPTRTA that are associated with different VPIDs and PCIDs.

². If the capability MSR IA32_VMX_CRO_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

mappings for the current VPID are invalidated even if EPT is in use. An EPT violation invalidates any guest-physical mappings (associated with the current EPTRTA) that would be used to translate the guest-physical address that caused the EPT violation. If that guest-physical address was the translation of a linear address, the EPT violation also invalidates any combined mappings for that linear address associated with the current PCID, the current VPID and the current EPTRTA.

If the “enable VPID” VM-execution control is 0, VM entries and VM exits invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs). Combined mappings for VPID 0000H are invalidated for all EPTRTAs.

Execution of the INVPID instruction invalidates linear mappings and combined mappings. Invalidation is based on instruction operands, called the INVPID type and the INVPID descriptor. Four INVPID types are currently defined:

- **Individual-address.** If the INVPID type is 0, the logical processor invalidates linear mappings and combined mappings associated with the VPID specified in the INVPID descriptor and that would be used to translate the linear address specified in of the INVPID descriptor. Linear mappings and combined mappings for that VPID and linear address are invalidated for all PCIDs and, for combined mappings, all EPTRTAs. (The instruction may also invalidate mappings associated with other VPIDs and for other linear addresses.)

- **Single-context.** If the INVPID type is 1, the logical processor invalidates all linear mappings and combined mappings associated with the VPID specified in the INVPID descriptor. Linear mappings and combined mappings for that VPID are invalidated for all PCIDs and, for combined mappings, all EPTRTAs. (The instruction may also invalidate mappings associated with other VPIDs.)

- **All-context.** If the INVPID type is 2, the logical processor invalidates linear mappings and combined mappings associated with all VPIDs except VPID 0000H and with all PCIDs. (The instruction may also invalidate linear mappings with VPID 0000H.) Combined mappings are invalidated for all EPTRTAs.

- **Single-context-retaining-globals.** If the INVPID type is 3, the logical processor invalidates linear mappings and combined mappings associated with the VPID specified in the INVPID descriptor. Linear mappings and combined mappings for that VPID are invalidated for all PCIDs and, for combined mappings, all EPTRTAs. The logical processor is not required to invalidate information that was used for global translations (although it may do so). See Section 4.10, “Caching Translation Information,” for details regarding global translations. (The instruction may also invalidate mappings associated with other VPIDs.)

See Chapter 31 for details of the INVPID instruction. See Section 29.4.3.3 for guidelines regarding use of this instruction.

Execution of the INVEPT instruction invalidates guest-physical mappings and combined mappings. Invalidation is based on instruction operands, called the INVEPT type and the INVEPT descriptor. Two INVEPT types are currently defined:

- **Single-context.** If the INVEPT type is 1, the logical processor invalidates all guest-physical mappings and combined mappings associated with the EPTRTA specified in the INVEPT descriptor. Combined mappings for that EPTRTA are invalidated for all VPIDs and all PCIDs. (The instruction may invalidate mappings associated with other EPTRTAs.)

- **All-context.** If the INVEPT type is 2, the logical processor invalidates guest-physical mappings and combined mappings associated with all EPTRTAs (and, for combined mappings, for all VPIDs and PCIDs).

See Chapter 31 for details of the INVEPT instruction. See Section 29.4.3.4 for guidelines regarding use of this instruction.

A power-up or a reset invalidates all linear mappings, guest-physical mappings, and combined mappings.

---

1. While no linear mappings are created while EPT is in use, a logical processor may retain, while EPT is in use, linear mappings (for the same VPID as the current one) there were created earlier, when EPT was not in use.

2. While no combined mappings are created while EPT is not in use, a logical processor may retain, while EPT is in not use, combined mappings (for the same VPID as the current one) there were created earlier, when EPT was in use.
29.4.3.2  Operations that Need Not Invalidate Cached Mappings

The following items detail cases of operations that are not required to invalidate certain cached mappings:

• Operations that architecturally invalidate entries in the TLBs or paging-structure caches independent of VMX operation are not required to invalidate any guest-physical mappings.
• The INVVPID instruction is not required to invalidate any guest-physical mappings.
• The INVEPT instruction is not required to invalidate any linear mappings.
• VMX transitions are not required to invalidate any guest-physical mappings. If the "enable VPID" VM-execution control is 1, VMX transitions are not required to invalidate any linear mappings or combined mappings.
• The VMXOFF and VMXON instructions are not required to invalidate any linear mappings, guest-physical mappings, or combined mappings.

A logical processor may invalidate any cached mappings at any time. For this reason, the operations identified above may invalidate the indicated mappings despite the fact that doing so is not required.

29.4.3.3  Guidelines for Use of the INVVPID Instruction

The need for VMM software to use the INVVPID instruction depends on how that software is virtualizing memory. If EPT is not in use, it is likely that the VMM is virtualizing the guest paging structures. Such a VMM may configure the VMCS so that all or some of the operations that invalidate entries the TLBs and the paging-structure caches (e.g., the INVLPG instruction) cause VM exits. If VMM software is emulating these operations, it may be necessary to use the INVVPID instruction to ensure that the logical processor’s TLBs and the paging-structure caches are appropriately invalidated.

Requirements of when software should use the INVVPID instruction depend on the specific algorithm being used for page-table virtualization. The following items provide guidelines for software developers:

• Emulation of the INVLPG instruction may require execution of the INVVPID instruction as follows:
  — The INVVPID type is individual-address (0).
  — The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.
  — The linear address in the INVVPID descriptor is that of the operand of the INVLPG instruction being emulated.

• Some instructions invalidate all entries in the TLBs and paging-structure caches—except for global translations. An example is the MOV to CR3 instruction. (See Section 4.10, “Caching Translation Information,” in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, for details regarding global translations.) Emulation of such an instruction may require execution of the INVVPID instruction as follows:
  — The INVVPID type is single-context-retaining-globals (3).
  — The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.

• Some instructions invalidate all entries in the TLBs and paging-structure caches—including for global translations. An example is the MOV to CR4 instruction if the value of value of bit 4 (page global enable—PGE) is changing. Emulation of such an instruction may require execution of the INVVPID instruction as follows:
  — The INVVPID type is single-context (1).
  — The VPID in the INVVPID descriptor is the one assigned to the virtual processor whose execution is being emulated.

If EPT is not in use, the logical processor associates all mappings it creates with the current VPID, and it will use such mappings to translate linear addresses. For that reason, a VMM should not use the same VPID for different non-EPT guests that use different page tables. Doing so may result in one guest using translations that pertain to the other.

If EPT is in use, the instructions enumerated above might not be configured to cause VM exits and the VMM might not be emulating them. In that case, executions of the instructions by guest software properly invalidate the
required entries in the TLBs and paging-structure caches (see Section 29.4.3.1); execution of the INVPID instruc-
tion is not required.

If EPT is in use, the logical processor associates all mappings it creates with the value of bits 51:12 of current EPTP. If a VMM uses different EPTP values for different guests, it may use the same VPID for those guests. Doing so cannot result in one guest using translations that pertain to the other.

The following guidelines apply more generally and are appropriate even if EPT is in use:

• As detailed in Section 30.4.5, an access to the APIC-access page might not cause an APIC-access VM exit if software does not properly invalidate information that may be cached from the paging structures. If, at one time, the current VPID on a logical processor was a non-zero value X, it is recommended that software use the INVPID instruction with the “single-context” INVPID type and with VPID X in the INVPID descriptor before a VM entry on the same logical processor that establishes VPID X and either (a) the “virtualize APIC accesses” VM-execution control was changed from 0 to 1; or (b) the value of the APIC-access address was changed.

• Software can use the INVPID instruction with the “all-context” INVPID type immediately after execution of the VMXON instruction or immediately prior to execution of the VMXOFF instruction. Either prevents potentially undesired retention of information cached from paging structures between separate uses of VMX operation.

29.4.3.4 Guidelines for Use of the INVEPT Instruction

The following items provide guidelines for use of the INVEPT instruction to invalidate information cached from the EPT paging structures.

• Software should use the INVEPT instruction with the “single-context” INVEPT type after making any of the following changes to an EPT paging-structure entry (the INVEPT descriptor should contain an EPTP value that references — directly or indirectly — the modified EPT paging structure):
  — Changing any of the privilege bits 2:0 from 1 to 0.1
  — Changing the physical address in bits 51:12.
  — Clearing bit 8 (the accessed flag) if accessed and dirty flags for EPT will be enabled.
  — For an EPT PDPT or an EPT PDE, changing bit 7 (which determines whether the entry maps a page).
  — For the last EPT paging-structure entry used to translate a guest-physical address (an EPT PDPT with bit 7 set to 1, an EPT PDE with bit 7 set to 1, or an EPT PTE), changing either bits 5:3 or bit 6. (These bits determine the effective memory type of accesses using that EPT paging-structure entry; see Section 29.3.7.)
  — For the last EPT paging-structure entry used to translate a guest-physical address (an EPT PDPT with bit 7 set to 1, an EPT PDE with bit 7 set to 1, or an EPT PTE), clearing bit 9 (the dirty flag) if accessed and dirty flags for EPT will be enabled.

• Software should use the INVEPT instruction with the “single-context” INVEPT type before a VM entry with an EPTP value X such that X[6] = 1 (accessed and dirty flags for EPT are enabled) if the logical processor had earlier been in VMX non-root operation with an EPTP value Y such that Y[6] = 0 (accessed and dirty flags for EPT are not enabled) and Y[51:12] = X[51:12].

• Software should use the INVEPT instruction with the “single-context” INVEPT type before a VM entry with an EPTP value X if the logical processor had earlier been in VMX non-root operation with an EPTP value Y such that Y[5:3] ≠ X[5:3] (different EPT page-walk length) and Y[51:12] = X[51:12].

• Software may use the INVEPT instruction after modifying a present EPT paging-structure entry (see Section 29.3.2) to change any of the privilege bits 2:0 from 0 to 1.2 Failure to do so may cause an EPT violation that would not otherwise occur. Because an EPT violation invalidates any mappings that would be used by the access that caused the EPT violation (see Section 29.4.3.1), an EPT violation will not recur if the original access is performed again, even if the INVEPT instruction is not executed.

1. If the “mode-based execute control for EPT” VM-execution control is 1, software should use the INVEPT instruction after changing privilege bit 10 from 1 to 0.
2. If the “mode-based execute control for EPT” VM-execution control is 1, software may use the INVEPT instruction after modifying a present EPT paging-structure entry to change privilege bit 10 from 0 to 1.
VMX SUPPORT FOR ADDRESS TRANSLATION

- Because a logical processor does not cache any information derived from EPT paging-structure entries that are not present (see Section 29.3.2) or misconfigured (see Section 29.3.3.1), it is not necessary to execute INVEPT following modification of an EPT paging-structure entry that had been not present or misconfigured.

- As detailed in Section 30.4.5, an access to the APIC-access page might not cause an APIC-access VM exit if software does not properly invalidate information that may be cached from the EPT paging structures. If EPT was in use on a logical processor at one time with EPTP X, it is recommended that software use the INVEPT instruction with the "single-context" INVEPT type and with EPTP X in the INVEPT descriptor before a VM entry on the same logical processor that enables EPT with EPTP X and either (a) the "virtualize APIC accesses" VM-execution control was changed from 0 to 1; or (b) the value of the APIC-access address was changed.

- Software can use the INVEPT instruction with the "all-context" INVEPT type immediately after execution of the VMXON instruction or immediately prior to execution of the VMXOFF instruction. Either prevents potentially undesired retention of information cached from EPT paging structures between separate uses of VMX operation.

In a system containing more than one logical processor, software must account for the fact that information from an EPT paging-structure entry may be cached on logical processors other than the one that modifies that entry. The process of propagating the changes to a paging-structure entry is commonly referred to as "TLB shootdown." A discussion of TLB shootdown appears in Section 4.10.5, "Propagation of Paging-Structure Changes to Multiple Processors," in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
The VMCS includes controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

When these controls are used, the processor will emulate many accesses to the APIC, track the state of the virtual APIC, and deliver virtual interrupts — all in VMX non-root operation with out a VM exit.1

The processor tracks the state of the virtual APIC using a virtual-APIC page identified by the virtual-machine monitor (VMM). Section 30.1 discusses the virtual-APIC page and how the processor uses it to track the state of the virtual APIC.

The following are the VM-execution controls relevant to APIC virtualization and virtual interrupts (see Section 25.6 for information about the locations of these controls):

• **Virtual-interrupt delivery.** This control enables the evaluation and delivery of pending virtual interrupts (Section 30.2). It also enables the emulation of writes (memory-mapped or MSR-based, as enabled) to the APIC registers that control interrupt prioritization.

• **Use TPR shadow.** This control enables emulation of accesses to the APIC’s task-priority register (TPR) via CR8 (Section 30.3) and, if enabled, via the memory-mapped or MSR-based interfaces.

• **Virtualize APIC accesses.** This control enables virtualization of memory-mapped accesses to the APIC (Section 30.4) by causing VM exits on accesses to a VMM-specified APIC-access page. Some of the other controls, if set, may cause some of these accesses to be emulated rather than causing VM exits.

• **Virtualize x2APIC mode.** This control enables virtualization of MSR-based accesses to the APIC (Section 30.5).

• **APIC-register virtualization.** This control allows memory-mapped and MSR-based reads of most APIC registers (as enabled) by satisfying them from the virtual-APIC page. It directs memory-mapped writes to the APIC-access page to the virtual-APIC page, following them by VM exits for VMM emulation.

• **Process posted interrupts.** This control allows software to post virtual interrupts in a data structure and send a notification to another logical processor; upon receipt of the notification, the target processor will process the posted interrupts by copying them into the virtual-APIC page (Section 30.6).

• **IPI virtualization.** This control enables the virtualization of interprocessor interrupts (Section 30.1.6).

“Virtualize APIC accesses”, “virtualize x2APIC mode”, “virtual-interrupt delivery”, and “APIC-register virtualization” are all secondary processor-based VM-execution controls; if bit 31 of the primary processor-based VM-execution controls is 0, the processor operates as if these controls were all 0. “IPI virtualization” is a tertiary processor-based VM-execution control; if bit 17 of the primary processor-based VM-execution controls is 0, the processor operates as if “IPI virtualization” were 0. See Section 25.6.2.

### 30.1 VIRTUAL APIC STATE

The **virtual-APIC page** is a 4-KByte region of memory that the processor uses to virtualize certain accesses to APIC registers and to manage virtual interrupts. The physical address of the virtual-APIC page is the **virtual-APIC address**, a 64-bit VM-execution control field in the VMCS (see Section 25.6.8).

Depending on the settings of certain VM-execution controls, the processor may virtualize certain fields on the virtual-APIC page with functionality analogous to that performed by the local APIC. Section 30.1.1 identifies and defines these fields. Section 30.1.2, Section 30.1.3, Section 30.1.4, and Section 30.1.5 detail the actions taken to virtualize updates to some of these fields.

With the exception of fields corresponding to virtualized APIC registers (defined in Section 30.1.1), software may modify the virtual-APIC page referenced by the current VMCS of a logical processor in VMX non-root operation. (This is an exception to the general requirement given in Section 25.11.4.)

---

1. In most cases, it is not necessary for a virtual-machine monitor (VMM) to inject virtual interrupts as part of VM entry.
30.1.1 Virtualized APIC Registers

Depending on the setting of certain VM-execution controls, a logical processor may virtualize certain accesses to APIC registers using the following fields on the virtual-APIC page:

- **Virtual task-priority register (VTPR)**: the 32-bit field located at offset 080H on the virtual-APIC page.
- **Virtual processor-priority register (VPPR)**: the 32-bit field located at offset 0A0H on the virtual-APIC page.
- **Virtual end-of-interrupt register (VEOI)**: the 32-bit field located at offset 0B0H on the virtual-APIC page.
- **Virtual interrupt-service register (VISR)**: the 256-bit value comprising eight non-contiguous 32-bit fields at offsets 100H, 110H, 120H, 130H, 140H, 150H, 160H, and 170H on the virtual-APIC page. Bit x of the VISR is at bit position \((x \& 1FH)\) at offset \((100H | ((x \& E0H) \gg 1))\). The processor uses only the low 4 bytes of each of the 16-byte fields at offsets 100H, 110H, 120H, 130H, 140H, 150H, 160H, and 170H.
- **Virtual interrupt-request register (VIRR)**: the 256-bit value comprising eight non-contiguous 32-bit fields at offsets 200H, 210H, 220H, 230H, 240H, 250H, 260H, and 270H on the virtual-APIC page. Bit x of the VIRR is at bit position \((x \& 1FH)\) at offset \((200H | ((x \& E0H) \gg 1))\). The processor uses only the low 4 bytes of each of the 16-byte fields at offsets 200H, 210H, 220H, 230H, 240H, 250H, 260H, and 270H.
- **Virtual interrupt-command register (VICR_LO)**: the 32-bit field located at offset 300H on the virtual-APIC page.
- **Virtual interrupt-command register (VICR_HI)**: the 32-bit field located at offset 310H on the virtual-APIC page.

The VTPR field virtualizes the TPR whenever the "use TPR shadow" VM-execution control is 1. The other fields indicated above virtualize the corresponding APIC registers whenever the "virtual-interrupt delivery" VM-execution control is 1. (VICR_LO and VICR_HI also virtualize the ICR when the "IPI virtualization" VM-execution control is 1.)

30.1.2 TPR Virtualization

The processor performs **TPR virtualization** in response to the following operations: (1) virtualization of the MOV to CR8 instruction; (2) virtualization of a write to offset 080H on the APIC-access page; and (3) virtualization of the WRMSR instruction with ECX = 808H. See Section 30.3, Section 30.4.3, and Section 30.5 for details of when TPR virtualization is performed.

The following pseudocode details the behavior of TPR virtualization:

```plaintext
IF "virtual-interrupt delivery" is 0
    THEN
        IF VTPR[7:4] < TPR threshold (see Section 25.6.8)
            THEN cause VM exit due to TPR below threshold;
            FI;
        ELSE
            perform PPR virtualization (see Section 30.1.3);
            evaluate pending virtual interrupts (see Section 30.2.1);
            FI;
FI;
```

Any VM exit caused by TPR virtualization is trap-like: the instruction causing TPR virtualization completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

30.1.3 PPR Virtualization

The processor performs **PPR virtualization** in response to the following operations: (1) VM entry; (2) TPR virtualization; and (3) EOI virtualization. See Section 27.3.2.5, Section 30.1.2, and Section 30.1.4 for details of when PPR virtualization is performed.

PPR virtualization uses the guest interrupt status (specifically, SVI; see Section 25.4.2) and VTPR. The following pseudocode details the behavior of PPR virtualization:

```plaintext
```
PPR virtualization always clears bytes 3:1 of VPPR.

PPR virtualization is caused only by TPR virtualization, EOI virtualization, and VM entry. Delivery of a virtual interrupt also modifies VPPR, but in a different way (see Section 30.2.2). No other operations modify VPPR, even if they modify SVI, VISR, or VTPR.

30.1.4 EOI Virtualization

The processor performs EOI virtualization in response to the following operations: (1) virtualization of a write to offset 0B0H on the APIC-access page; and (2) virtualization of the WRMSR instruction with ECX = 80BH. See Section 30.4.3 and Section 30.5 for details of when EOI virtualization is performed. EOI virtualization occurs only if the "virtual-interrupt delivery" VM-execution control is 1.

EOI virtualization uses and updates the guest interrupt status (specifically, SVI; see Section 25.4.2). The following pseudocode details the behavior of EOI virtualization:

Vector := SVI;
VISR[Vector] := 0; (see Section 30.1.1 for definition of VISR)
IF any bits set in VISR
   THEN SVI := highest index of bit set in VISR
   ELSE SVI := 0;
FI;
perform PPR virtualization (see Section 30.1.3);
IF EOI_exit_bitmap[Vector] = 1 (see Section 25.6.8 for definition of EOI_exit_bitmap)
   THEN cause EOI-induced VM exit with Vector as exit qualification;
   ELSE evaluate pending virtual interrupts; (see Section 30.2.1)
FI;

Any VM exit caused by EOI virtualization is trap-like: the instruction causing EOI virtualization completes before the VM exit occurs (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

30.1.5 Self-IPI Virtualization

The processor performs self-IPI virtualization in response to the following operations: (1) virtualization of a write to offset 300H on the APIC-access page; and (2) virtualization of the WRMSR instruction with ECX = 83FH. See Section 30.4.3 and Section 30.5 for details of when self-IPI virtualization is performed. Self-IPI virtualization occurs only if the "IPI virtualization" VM-execution control is 1.

Each operation that leads to self-IPI virtualization provides an 8-bit vector (see Section 30.4.3 and Section 30.5). Self-IPI virtualization updates the guest interrupt status (specifically, RVI; see Section 25.4.2). The following pseudocode details the behavior of self-IPI virtualization:

VIRR[Vector] := 1; (see Section 30.1.1 for definition of VIRR)
RVI := max{RVI,Vector};
evaluate pending virtual interrupts; (see Section 30.2.1)

30.1.6 IPI Virtualization

The processor performs IPI virtualization in response to the following operations: (1) virtualization of a write to offset 300H on the APIC-access page (Section 30.4.3); (2) virtualization of the WRMSR instruction with ECX = 830H (Section 30.5); and (3) virtualization of some executions of SENDUIPI (Section 30.7). IPI virtualization occurs only if the "IPI virtualization" VM-execution control is 1.
APIC VIRTUALIZATION AND VIRTUAL INTERRUPTS

Each operation that leads to IPI virtualization provides an 8-bit virtual vector V and a 32-bit virtual APIC ID T. IPI virtualization uses those values to initiate the indicated virtual IPI using the PID-pointer table. (Recall that the PID-pointer table address is a field in the VMCS, as is the last PID-pointer index.) Specifically, the CPU uses the virtual APIC ID to select an entry from the PID-pointer table. It uses the address in that entry to locate a posted-interrupt descriptor (PID) and then posts a virtual interrupt with vector V in that PID. The following pseudocode details the behavior of self-IPI virtualization:

```
IF V < 16
    THEN APIC-write VM exit; // illegal vector
ELSE IF T ≤ last PID-pointer index
    THEN
        PID_ADDR := 8 bytes at (PID-pointer table address + (T << 3));
        IF PID_ADDR sets bits beyond the processor's physical-address width OR
            PID_ADDR[5:0] ≠ 000001b // PID pointer not valid or reserved bits set
            THEN APIC-write VM exit; // See Section 30.4.3.3
        ELSE
            PIR := 32 bytes at PID_ADDR;
            PIR[V] := 1;
            store PIR at PID_ADDR;
            NotifyInfo := 8 bytes at PID_ADDR + 32;
            IF NotifyInfo.ON = 0 AND NotifyInfo.SN = 0
                THEN
                    NotifyInfo.ON := 1;
                    SendNotify := 1;
                    ELSE SendNotify := 0;
                    store NotifyInfo at PID_ADDR + 32; // release lock
                    IF SendNotify = 1
                        THEN send an IPI specified by NotifyInfo.NDST and NotifyInfo.NV;
                    FI;
            ELSE APIC-write VM exit; // virtual APIC ID beyond end of tables
        FI;
ELSE
    THEN APIC-write VM exit; // illegal vector
FI;
```

The sending of the notification IPI is indicated by fields in the selected PID: NDST (PID[319:288]) and NV (PID[279:272]):

- If the local APIC is in xAPIC mode, this is the IPI that would be generated by writing NDST[15:8] (PID[303:296]) to ICR_HI[31:24] (offset 310H from IA32_APIC_BASE) and then writing NV to ICR_LO (offset 300H from IA32_APIC_BASE).
- If the local APIC is in x2APIC mode, this is the IPI that would be generated by executing WRMSR with ECX = 830H (ICR), EAX = NV, and EDX = NDST.

If the pseudocode specifies an APIC-write VM exit, this VM exit occurs as if there had been a write access to page offset 300H on the APIC-access page (see Section 30.4.3.3).

30.2 EVALUATION AND DELIVERY OF VIRTUAL INTERRUPTS

If the "virtual-interrupt delivery" VM-execution control is 1, certain actions in VMX non-root operation or during VM entry cause the processor to evaluate and deliver virtual interrupts.

Evaluation of virtual interrupts is triggered by certain actions change the state of the virtual-APIC page and is described in Section 30.2.1. This evaluation may result in recognition of a virtual interrupt. Once a virtual interrupt is recognized, the processor may deliver it within VMX non-root operation without a VM exit. Virtual-interrupt delivery is described in Section 30.2.2.

30.2.1 Evaluation of Pending Virtual Interrupts

If the "virtual-interrupt delivery" VM-execution control is 1, certain actions cause a logical processor to evaluate pending virtual interrupts.

The following actions cause the evaluation of pending virtual interrupts: VM entry; TPR virtualization; EOI virtualization; self-IPI virtualization; and posted-interrupt processing. See Section 27.3.2.5, Section 30.1.2, Section
30.1.4, Section 30.1.5, and Section 30.6 for details of when evaluation of pending virtual interrupts is performed. No other operations cause the evaluation of pending virtual interrupts, even if they modify RVI or VPPR.

Evaluation of pending virtual interrupts uses the guest interrupt status (specifically, RVI; see Section 25.4.2). The following pseudocode details the evaluation of pending virtual interrupts:

```plaintext
IF "interrupt-window exiting" is 0 AND
   RVI[7:4] > VPPR[7:4] (see Section 30.1.1 for definition of VPPR)
   THEN recognize a pending virtual interrupt;
ELSE
   do not recognize a pending virtual interrupt;
FI;
```

Once recognized, a virtual interrupt may be delivered in VMX non-root operation; see Section 30.2.2.

Evaluation of pending virtual interrupts is caused only by VM entry, TPR virtualization, EOI virtualization, self-IPI virtualization, and posted-interrupt processing. No other operations do so, even if they modify RVI or VPPR. The logical processor ceases recognition of a pending virtual interrupt following the delivery of a virtual interrupt.

### 30.2.2 Virtual-Interrupt Delivery

If a virtual interrupt has been recognized (see Section 30.2.1), it is delivered at an instruction boundary when the following conditions all hold: (1) RFLAGS.IF = 1; (2) there is no blocking by STI; (3) there is no blocking by MOV SS or by POP SS; and (4) the "interrupt-window exiting" VM-execution control is 0.

Virtual-interrupt delivery has the same priority as that of VM exits due to the 1-setting of the "interrupt-window exiting" VM-execution control.\(^2\) Thus, non-maskable interrupts (NMIs) and higher priority events take priority over delivery of a virtual interrupt; delivery of a virtual interrupt takes priority over external interrupts and lower priority events.

Virtual-interrupt delivery wakes a logical processor from the same inactive activity states as would an external interrupt. Specifically, it wakes a logical processor from the states entered using the HLT and MWAIT instructions. It does not wake a logical processor in the shutdown state or in the wait-for-SIPI state.

Virtual-interrupt delivery updates the guest interrupt status (both RVI and SVI; see Section 25.4.2) and delivers an event within VMX non-root operation without a VM exit. The following pseudocode details the behavior of virtual-interrupt delivery (see Section 30.1.1 for definition of VISR, VIRR, and VPPR):

```plaintext
Vector := RVI;
VISR[Vector] := 1;
SVI := Vector;
VPPR := Vector & F0H;
VIRR[Vector] := 0;
IF any bits set in VIRR
   THEN RVI := highest index of bit set in VIRR
   ELSE RVI := 0;
FI;
```

cease recognition of any pending virtual interrupt;

```plaintext
IF transactional execution is in effect
   THEN abort transactional execution and transition to a non-transactional execution;
FI;
```

```plaintext
IF logical processor is in enclave mode
   THEN cause an Asynchronous Enclave Exit (AEX) (see Chapter 37, “Enclave Exiting Events”)
FI;
```

```plaintext
IF CR4.UINTR = 1 AND IA32_EFER.LMA = 1 AND Vector = UINV
   THEN virtualize user-interrupt notification identification and processing (see Section 30.2.3)
   ELSE deliver interrupt with Vector through IDT;
```

\(^2\) A logical processor never recognizes or delivers a virtual interrupt if the "interrupt-window exiting" VM-execution control is 1. Because of this, the relative priority of virtual-interrupt delivery and VM exits due to the 1-setting of that control is not defined.
30.2.3 Virtualizing User-Interrupt Notifications

Section 7.5 describes the process of user-interrupt notification identification and processing. If the “virtual-interrupt delivery” VM-execution control is 1, this process is modified as described in the following paragraphs.

The virtualized form of user-interrupt notification identification begins as described in Section 30.2.2. Following this, instead of writing zero to the EOI register in the local APIC, the logical processor performs the initial steps of EOI virtualization:

\[
\text{VISR}[V] := 0; \\
\text{IF any bit is set in VISR} \\
\quad \text{THEN SVI := highest index of bit set in VISR} \\
\quad \text{ELSE SVI := 0;} \\
\text{FI;}
\]

perform PPR virtualization (Section 30.1.3);

Unlike EOI virtualization resulting from a guest write to the EOI register (as defined for virtual-interrupt delivery), the logical processor does not check the EOI-exit bitmap as part of this modified form of user-interrupt notification identification, and the corresponding VM exits cannot occur.

Following this modified form of user-interrupt notification identification, the logical processor then performs user-interrupt notification processing as specified in Section 7.5.2.

A logical processor is not interruptible during this modified form of user-interrupt notification identification or between it and any subsequent user-interrupt notification processing.

If the user-interrupt notification identification that precedes user-interrupt notification processing occurred while the logical processor was in the HLT state, the logical processor returns to the HLT state following user-interrupt notification processing.

30.3 VIRTUALIZING CR8-BASED TPR ACCESSES

In 64-bit mode, software can access the local APIC’s task-priority register (TPR) through CR8. Specifically, software uses the MOV from CR8 and MOV to CR8 instructions (see Section 11.8.6, “Task Priority in IA-32e Mode”). This section describes how these accesses can be virtualized.

A virtual-machine monitor can virtualize these CR8-based APIC accesses by setting the “CR8-load exiting” and “CR8-store exiting” VM-execution controls, ensuring that the accesses cause VM exits (see Section 26.1.3). Alternatively, there are methods for virtualizing some CR8-based APIC accesses without VM exits.

Normally, an execution of MOV from CR8 or MOV to CR8 that does not fault or cause a VM exit accesses the APIC’s TPR. However, such an execution are treated specially if the “use TPR shadow” VM-execution control is 1. The following items provide details:

- **MOV from CR8.** The instruction loads bits 3:0 of its destination operand with bits 7:4 of VTPR (see Section 30.1.1). Bits 63:4 of the destination operand are cleared.
- **MOV to CR8.** The instruction stores bits 3:0 of its source operand into bits 7:4 of VTPR; the remainder of VTPR (bits 3:0 and bits 31:8) are cleared. Following this, the processor performs TPR virtualization (see Section 30.1.2).

30.4 VIRTUALIZING MEMORY-MAPPED APIC ACCESSES

When the local APIC is in xAPIC mode, software accesses the local APIC’s control registers using a memory-mapped interface. Specifically, software uses linear addresses that translate to physical addresses on page frame indicated by the base address in the IA32_APIC_BASE MSR (see Section 11.4.4, “Local APIC Status and Location”). This section describes how these accesses can be virtualized.
A virtual-machine monitor (VMM) can virtualize these memory-mapped APIC accesses by ensuring that any access to a linear address that would access the local APIC instead causes a VM exit. This could be done using paging or the extended page-table mechanism (EPT). Another way is by using the 1-setting of the “virtualize APIC accesses” VM-execution control.

If the "virtualize APIC accesses" VM-execution control is 1, the logical processor treats specially memory accesses using linear addresses that translate to physical addresses in the 4-KByte APIC-access page.3,4 (The APIC-access page is identified by the APIC-access address, a field in the VMCS; see Section 25.6.8.)

In general, an access to the APIC-access page causes an APIC-access VM exit. APIC-access VM exits provide a VMM with information about the access causing the VM exit. Section 30.4.1 discusses the priority of APIC-access VM exits.

Certain VM-execution controls enable the processor to virtualize certain accesses to the APIC-access page without a VM exit. In general, this virtualization causes these accesses to be made to the virtual-APIC page instead of the APIC-access page.

NOTES

Unless stated otherwise, this section characterizes only linear accesses to the APIC-access page; an access to the APIC-access page is a linear access if (1) it results from a memory access using a linear address; and (2) the access’s physical address is the translation of that linear address. Section 30.4.6 discusses accesses to the APIC-access page that are not linear accesses.

The distinction between the APIC-access page and the virtual-APIC page allows a VMM to share paging structures or EPT paging structures among the virtual processors of a virtual machine (the shared paging structures referencing the same APIC-access address, which appears in the VMCS of all the virtual processors) while giving each virtual processor its own virtual APIC (the VMCS of each virtual processor will have a unique virtual-APIC address).

Section 30.4.2 discusses when and how the processor may virtualize read accesses from the APIC-access page. Section 30.4.3 does the same for write accesses. When virtualizing a write to the APIC-access page, the processor typically takes actions in addition to passing the write through to the virtual-APIC page.

The discussion in those sections uses the concept of an operation within which these memory accesses may occur. For those discussions, an “operation” can be an iteration of a REP-prefixed string instruction, an execution of any other instruction, or delivery of an event through the IDT.

The 1-setting of the “virtualize APIC accesses” VM-execution control may also affect accesses to the APIC-access page that do not result directly from linear addresses. This is discussed in Section 30.4.6.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 39.5.3 for details.

30.4.1 Priority of APIC-Access VM Exits

The following items specify the priority of APIC-access VM exits relative to other events.

- The priority of an APIC-access VM exit due to a memory access is below that of any page fault or EPT violation that that access may incur. That is, an access does not cause an APIC-access VM exit if it would cause a page fault or an EPT violation.

3. Even when addresses are translated using EPT (see Section 29.3), the determination of whether an APIC-access VM exit occurs depends on an access’s physical address, not its guest-physical address. Even when CR0.PG = 0, ordinary memory accesses by software use linear addresses; the fact that CR0.PG = 0 means only that the identity translation is used to convert linear addresses to physical (or guest-physical) addresses.

4. If EPT is enabled and there is write to a guest-physical address that translates to an address on the APIC-access page that is eligible for sub-page write permissions (see Section 29.3.4.1), the processor may treat the write as if the “virtualize APIC accesses” VM-execution control were 0 (and not apply the treatment specified in this section). For that reason, it is recommended that software not configure any guest-physical address that translates to an address on the APIC-access page to be eligible for sub-page write permissions.
• A memory access does not cause an APIC-access VM exit until after the accessed flags are set in the paging structures (including EPT paging structures, if enabled).

• A write access does not cause an APIC-access VM exit until after the dirty flags are set in the appropriate paging structure and EPT paging structure (if enabled).

• With respect to all other events, any APIC-access VM exit due to a memory access has the same priority as any page fault or EPT violation that the access could cause. (This item applies to other events that the access may generate as well as events that may be generated by other accesses by the same operation.)

These principles imply, among other things, that an APIC-access VM exit may occur during the execution of a repeated string instruction (including INS and OUTS). Suppose, for example, that the first $n$ iterations ($n$ may be 0) of such an instruction do not access the APIC-access page and that the next iteration does access that page. As a result, the first $n$ iterations may complete and be followed by an APIC-access VM exit. The instruction pointer saved in the VMCS references the repeated string instruction and the values of the general-purpose registers reflect the completion of $n$ iterations.

### 30.4.2 Virtualizing Reads from the APIC-Access Page

A read access from the APIC-access page causes an APIC-access VM exit if any of the following are true:

• The “use TPR shadow” VM-execution control is 0.

• The access is for an instruction fetch.

• The access is more than 32 bits in size.

• The access is part of an operation for which the processor has already virtualized a write to the APIC-access page.

• The access is not entirely contained within the low 4 bytes of a naturally aligned 16-byte region. That is, bits 3:2 of the access's address are 0, and the same is true of the address of the highest byte accessed.

If none of the above are true, whether a read access is virtualized depends on the setting of the “APIC-register virtualization” and “virtual-interrupt delivery” VM-execution controls:

• A read access is virtualized if its page offset is 08OH (task priority) regardless of the settings of the “APIC-register virtualization” and “virtual-interrupt delivery” VM-execution controls.

• If the “virtual-interrupt delivery” VM-execution control is 1, a read access is virtualized if its page offset is 0B0H (end of interrupt) or 300H (interrupt command — low).

• If "APIC-register virtualization" is 1, a read access is virtualized if it is entirely within one the following ranges of offsets:
  - 020H–023H (local APIC ID);
  - 030H–033H (local APIC version);
  - 080H–083H (task priority);
  - 080H–0B3H (end of interrupt);
  - 0D0H–0D3H (logical destination);
  - 0E0H–0E3H (destination format);
  - 0F0H–0F3H (spurious-interrupt vector);
  - 180H–183H, 190H–193H, 1A0H–1A3H, 1B0H–1B3H, 1C0H–1C3H, 1D0H–1D3H, 1E0H–1E3H, or 1F0H–1F3H (trigger mode);
  - 280H–283H (error status);
  - 300H–303H or 310H–313H (interrupt command);
— 380H–383H (initial count); or
— 3E0H–3E3H (divide configuration).

In all other cases, the access causes an APIC-access VM exit.

A read access from the APIC-access page that is virtualized returns data from the corresponding page offset on the virtual-APIC page.5

### 30.4.3 Virtualizing Writes to the APIC-Access Page

Whether a write access to the APIC-access page is virtualized depends on the settings of the VM-execution controls and the page offset of the access. Section 30.4.3.1 details when APIC-write virtualization occurs.

Unlike reads, writes to the local APIC have side effects; because of this, virtualization of writes to the APIC-access page may require emulation specific to the access’s page offset (which identifies the APIC register being accessed). Section 30.4.3.2 describes this APIC-write emulation.

For some page offsets, it is necessary for software to complete the virtualization after a write completes. In these cases, the processor causes an APIC-write VM exit to invoke VMM software. Section 30.4.3.3 discusses APIC-write VM exits.

#### 30.4.3.1 Determining Whether a Write Access is Virtualized

A write access to the APIC-access page causes an APIC-access VM exit if any of the following are true:

- The “use TPR shadow” VM-execution control is 0.
- The access is more than 32 bits in size.
- The access is part of an operation for which the processor has already virtualized a write (with a different page offset or a different size) to the APIC-access page.
- The access is not entirely contained within the low 4 bytes of a naturally aligned 16-byte region. That is, bits 3:2 of the access’s address are 0, and the same is true of the address of the highest byte accessed.

If none of the above are true, whether a write access is virtualized depends on the settings of the “APIC-register virtualization”, “virtual-interrupt delivery”, and “IPI virtualization” VM-execution controls:

- A write access is virtualized if its page offset is 080H (task priority) regardless of the settings of the “APIC-register virtualization” and “virtual-interrupt delivery” VM-execution controls.
- If the “virtual-interrupt delivery” VM-execution control is 1, a write access is virtualized if its page offset is 0B0H (end of interrupt) or 300H (interrupt command — low).
- If the “IPI virtualization” VM-execution control is 1, a write access is virtualized if its page offset is 300H.
- If the “APIC-register virtualization” VM-execution control is 1, a write access is virtualized if it is entirely within one of the following ranges of offsets:
  - 020H–023H (local APIC ID);
  - 080H–083H (task priority);
  - 080H–0B3H (end of interrupt);
  - 0D0H–0D3H (logical destination);
  - 0E0H–0E3H (destination format);
  - 0F0H–0F3H (spurious-interrupt vector);
  - 280H–283H (error status);
  - 300H–303H or 310H–313H (interrupt command);

---

5. The memory type used for accesses that read from the virtual-APIC page is reported in bits 53:50 of the IA32_VMX_BASIC MSR (see Appendix A.1).
— 380H–383H (initial count); or
— 3E0H–3E3H (divide configuration).

In all other cases, the access causes an APIC-access VM exit.

The processor virtualizes a write access to the APIC-access page by writing data to the corresponding page offset on the virtual-APIC page. Following this, the processor performs certain actions after completion of the operation of which the access was a part. APIC-write emulation is described in Section 30.4.3.2.

### 30.4.3.2 APIC-Write Emulation

If the processor virtualizes a write access to the APIC-access page, it performs additional actions after completion of an operation of which the access was a part. These actions are called **APIC-write emulation**.

The details of APIC-write emulation depend upon the page offset of the virtualized write access:

- **080H (task priority).** The processor clears bytes 3:1 of VTPR and then causes TPR virtualization (Section 30.1.2).
- **0B0H (end of interrupt).** If the "virtual-interrupt delivery" VM-execution control is 1, the processor clears VEOI and then causes EOI virtualization (Section 30.1.4); otherwise, the processor causes an APIC-write VM exit (Section 30.4.3.3).
- **300H (interrupt command — low).** If the "virtual-interrupt delivery" VM-execution control is 1, the processor checks the value of VICR_LO to determine whether the following are all true:
  - Reserved bits (31:20, 17:16, 13) and bit 12 (delivery status) are all 0.
  - Bits 19:18 (destination shorthand) are 01B (self).
  - Bit 15 (trigger mode) is 0 (edge).
  - Bits 10:8 (delivery mode) are 000B (fixed).
  - Bits 7:4 (the upper half of the vector) are not 0000B.

If all of the items above are true, the processor performs self-IPI virtualization using the 8-bit vector in byte 0 of VICR_LO (Section 30.1.5).

If the "virtual-interrupt delivery" VM-execution control is 0, or if any of the items above are false, behavior depends on the setting of the "IPI virtualization" VM-execution control:

- If the "IPI virtualization" VM-execution control is 1, the processor checks the value of VICR_LO to determine whether the following are all true:
  - Reserved bits (31:20, 17:16, 13) and bit 12 (delivery status) are all 0.
  - Bits 19:18 (destination shorthand) are 00B (no shorthand).
  - Bit 15 (trigger mode) is 0 (edge).
  - Bit 11 (destination mode) is 0 (physical).
  - Bits 10:8 (delivery mode) are 000B (fixed).

If all of the items above are true, the processor performs IPI virtualization using the 8-bit vector in byte 0 of VICR_LO and the 8-bit APIC ID in VICR_HI[31:24] (Section 30.1.6); otherwise, the processor causes an APIC-write VM exit.

- If the "IPI virtualization" VM-execution control is 0, the processor causes an APIC-write VM exit.

---

6. The memory type used for accesses that write to the virtual-APIC page is reported in bits 53:50 of the IA32_VMX_BASIC MSR (see Appendix A.1).
7. Recall that, for the purposes of this discussion, an operation is an iteration of a REP-prefixed string instruction, an execution of any other instruction, or delivery of an event through the IDT.
8. For any operation, there can be only one page offset for which a write access was virtualized. This is because a write access is not virtualized if the processor has already virtualized a write access for the same operation with a different page offset.
• 310H–313H (interrupt command — high). The processor clears bytes 2:0 of VICR_HI. No other virtualization or VM exit occurs.
• Any other page offset. The processor causes an APIC-write VM exit.

APIC-write emulation takes priority over system-management interrupts (SMIs), INIT signals, and lower priority events. APIC-write emulation is not blocked if RFLAGS.IF = 0 or by the MOV SS, POP SS, or STI instructions.

If an operation causes a fault after a write access to the APIC-access page and before APIC-write emulation, and that fault is delivered without a VM exit, APIC-write emulation occurs after the fault is delivered and before the fault handler can execute. If an operation causes a VM exit (perhaps due to a fault) after a write access to the APIC-access page and before APIC-write emulation, the APIC-write emulation does not occur.

30.4.3.3 APIC-Write VM Exits

In certain cases, VMM software must be invoked to complete the virtualization of a write access to the APIC-access page. In this case, APIC-write emulation causes an APIC-write VM exit. (Section 30.4.3.2 details the cases that causes APIC-write VM exits.)

APIC-write VM exits are invoked by APIC-write emulation, and APIC-write emulation occurs after an operation that performs a write access to the APIC-access page. Because of this, every APIC-write VM exit is trap-like: it occurs after completion of the operation containing the write access that caused the VM exit (for example, the value of CS:RIP saved in the guest-state area of the VMCS references the next instruction).

The basic exit reason for an APIC-write VM exit is “APIC write.” The exit qualification is the page offset of the write access that led to the VM exit.

As noted in Section 30.5, execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit if the “virtual-interrupt delivery” VM-execution control is 1; the exit qualification for the APIC-write VM exit is 3F0H. As noted in Section 30.1.6 and in Section 30.7, IPI virtualization and execution of SENDUIPI may lead to APIC-write VM exits; these VM exits produce an exit qualification of 300H.

30.4.4 Instruction-Specific Considerations

Certain instructions that use linear address may cause page faults even though they do not use those addresses to access memory. The APIC-virtualization features may affect these instructions as well:

• CLFLUSH, CLFLUSHOPT. With regard to faulting, the processor operates as if each of these instructions reads from the linear address in its source operand. If that address translates to one on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will flush the corresponding cache line on the virtual-APIC page instead of the APIC-access page.
• ENTER. With regard to faulting, the processor operates if ENTER writes to the byte referenced by the final value of the stack pointer (even though it does not if its size operand is non-zero). If that value translates to an address on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will cause the APIC-write emulation appropriate to the address’s page offset.
• MASKMOVQ and MASKMOVQ. Even if the instruction’s mask is zero, the processor may operate with regard to faulting as if MASKMOVQ or MASKMOVQ writes to memory (the behavior is implementation-specific). In such a situation, an APIC-access VM exit may occur.
• MONITOR. With regard to faulting, the processor operates as if MONITOR reads from the effective address in RAX. If the resulting linear address translates to one on the APIC-access page, the instruction may cause an APIC-access VM exit. If it does not, it will monitor the corresponding address on the virtual-APIC page instead of the APIC-access page.
• PREFETCH. An execution of the PREFETCH instruction that would result in an access to the APIC-access page does not cause an APIC-access VM exit. Such an access may prefetch data; if so, it is from the corresponding address on the virtual-APIC page.

9. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For IA-32 processors, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
Virtualization of accesses to the APIC-access page is principally intended for basic instructions such as AND, MOV, OR, TEST, XCHG, and XOR. Use of an instruction that normally operates on floating-point, SSE, AVX, or AVX-512 registers may cause an APIC-access VM exit unconditionally regardless of the page offset it accesses on the APIC-access page.

### 30.4.5 Issues Pertaining to Page Size and TLB Management

The 1-setting of the "virtualize APIC accesses" VM-execution is guaranteed to apply only if translations to the APIC-access address use a 4-KByte page. The following items provide details:

- **If EPT is not in use, any linear address that translates to an address on the APIC-access page should use a 4-KByte page. Any access to a linear address that translates to the APIC-access page using a larger page may operate as if the "virtualize APIC accesses" VM-execution control were 0.**

- **If EPT is in use, any guest-physical address that translates to an address on the APIC-access page should use a 4-KByte page. Any access to a linear address that translates to a guest-physical address that in turn translates to the APIC-access page using a larger page may operate as if the "virtualize APIC accesses" VM-execution control were 0. (This is true also for guest-physical accesses to the APIC-access page; see Section 30.4.6.1.)**

In addition, software should perform appropriate TLB invalidation when making changes that may affect APIC-virtualization. The specifics depend on whether VPIDs or EPT is being used:

- **VPIDs being used but EPT not being used.** Suppose that there is a VPID that has been used before and that software has since made either of the following changes: (1) set the "virtualize APIC accesses" VM-execution control when it had previously been 0; or (2) changed the paging structures so that some linear address translates to the APIC-access address when it previously did not. In that case, software should execute INVPVID (see "INVPVID— Invalidate Translations Based on VPID" in Section 31.3) before performing on the same logical processor and with the same VPID.10

- **EPT being used.** Suppose that there is an EPTP value that has been used before and that software has since made either of the following changes: (1) set the "virtualize APIC accesses" VM-execution control when it had previously been 0; or (2) changed the EPT paging structures so that some guest-physical address translates to the APIC-access address when it previously did not. In that case, software should execute INVEPT (see "INVEPT— Invalidate Translations Derived from EPT" in Section 31.3) before performing on the same logical processor and with the same EPTP value.11

- **Neither VPIDs nor EPT being used.** No invalidation is required.

Failure to perform the appropriate TLB invalidation may result in the logical processor operating as if the "virtualize APIC accesses" VM-execution control were 0 in responses to accesses to the affected address. (No invalidation is necessary if neither VPIDs nor EPT is being used.)

### 30.4.6 APIC Accesses Not Directly Resulting From Linear Addresses

Section 30.4 has described the treatment of accesses that use linear addresses that translate to addresses on the APIC-access page. This section considers memory accesses that do not result directly from linear addresses.

- An access is called a **guest-physical access** if (1) CR0.PG = 1;12 (2) the “enable EPT” VM-execution control is 1;13 (3) the access’s physical address is the result of an EPT translation; and (4) either (a) the access was not generated by a linear address; or (b) the access’s guest-physical address is not the translation of the access’s linear address. Section 30.4.6.1 discusses the treatment of guest-physical accesses to the APIC-access page.

10. INVPVID should use either (1) the all-contexts INVPVID type; (2) the single-context INVPVID type with the VPID in the INVPVID descriptor; or (3) the individual-address INVPVID type with the linear address and the VPID in the INVPVID descriptor.

11. INVEPT should use either (1) the global INVEPT type; or (2) the single-context INVEPT type with the EPTP value in the INVEPT descriptor.

12. If the capability MSR IA32_VMX_CRO_Fixed0 reports that CR0.PG must be 1 in VMX operation, CR0.PG must be 1 unless the “unrestricted guest” VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

13. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "enable EPT" VM-execution control were 0. See Section 25.6.2.
• An access is called a **physical access** if (1) either (a) the "enable EPT" VM-execution control is 0; or (b) the access's physical address is not the result of a translation through the EPT paging structures; and (2) either (a) the access is not generated by a linear address; or (b) the access’s physical address is not the translation of its linear address. Section 30.4.6.2 discusses the treatment of physical accesses to the APIC-access page.

### 30.4.6.1 Guest-Physical Accesses to the APIC-Access Page

Guest-physical accesses include the following when guest-physical addresses are being translated using EPT:

• Reads from the guest paging structures when translating a linear address (such an access uses a guest-physical address that is not the translation of that linear address).

• Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging (see Section 4.4).

• Updates to the accessed and dirty flags in the guest paging structures when using a linear address (such an access uses a guest-physical address that is not the translation of that linear address).

• Memory accesses by Intel Processor Trace when the “Intel PT uses guest physical addresses” VM-execution control is 1 (see Section 26.5.4).

Every guest-physical access using a guest-physical address that translates to an address on the APIC-access page causes an APIC-access VM exit. Such accesses are never virtualized regardless of the page offset.

The following items specify the priority relative to other events of APIC-access VM exits caused by guest-physical accesses to the APIC-access page.

• The priority of an APIC-access VM exit caused by a guest-physical access to memory is below that of any EPT violation that that access may incur. That is, a guest-physical access does not cause an APIC-access VM exit if it would cause an EPT violation.

• With respect to all other events, any APIC-access VM exit caused by a guest-physical access has the same priority as any EPT violation that the guest-physical access could cause.

### 30.4.6.2 Physical Accesses to the APIC-Access Page

Physical accesses include the following:

• If the “enable EPT” VM-execution control is 0:
  
  — Reads from the paging structures when translating a linear address.
  
  — Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging (see Section 4.4).
  
  — Updates to the accessed and dirty flags in the paging structures.

• If the “enable EPT” VM-execution control is 1, accesses to the EPT paging structures (including updates to the accessed and dirty flags for EPT).

• Any of the following accesses made by the processor to support VMX non-root operation:
  
  — Accesses to the VMCS region.
  
  — Accesses to data structures referenced (directly or indirectly) by physical addresses in VM-execution control fields in the VMCS. These include the I/O bitmaps, the MSR bitmaps, and the virtual-APIC page.

• Accesses that effect transitions into and out of SMM.\(^\text{14}\) These include the following:
  
  — Accesses to SMRAM during SMI delivery and during execution of RSM.
  
  — Accesses during SMM VM exits (including accesses to MSEG) and during VM entries that return from SMM.

A physical access to the APIC-access page may or may not cause an APIC-access VM exit. If it does not cause an APIC-access VM exit, it may access the APIC-access page or the virtual-APIC page. Physical write accesses to the APIC-access page may or may not cause APIC-write emulation or APIC-write VM exits.

\(^{14}\) Technically, these accesses do not occur in VMX non-root operation. They are included here for clarity.
APIC VIRTUALIZATION AND VIRTUAL INTERRUPTS

The priority of an APIC-access VM exit caused by physical access is not defined relative to other events that the access may cause.

It is recommended that software not set the APIC-access address to any of the addresses used by physical memory accesses (identified above). For example, it should not set the APIC-access address to the physical address of any of the active paging structures if the “enable EPT” VM-execution control is 0.

30.5 VIRTUALIZING MSR-BASED APIC ACCESSES

When the local APIC is in x2APIC mode, software accesses the local APIC’s control registers using the MSR interface. Specifically, software uses the RDMSR and WRMSR instructions, setting ECX (identifying the MSR being accessed) to values in the range 800H–8FFH (see Section 11.12, "Extended XAPIC (x2APIC)"). This section describes how these accesses can be virtualized.

A virtual-machine monitor can virtualize these MSR-based APIC accesses by configuring the MSR bitmaps (see Section 25.6.9) to ensure that the accesses cause VM exits (see Section 26.1.3). Alternatively, there are methods for virtualizing some MSR-based APIC accesses without VM exits.

Normally, an execution of RDMSR or WRMSR that does not fault or cause a VM exit accesses the MSR indicated in ECX. However, such an execution treats some values of ECX in the range 800H–8FFH specially if the “virtualize x2APIC mode” VM-execution control is 1. The following items provide details:

• **RDMSR.** The instruction’s behavior depends on the setting of the “APIC-register virtualization” VM-execution control.
  - If the “APIC-register virtualization” VM-execution control is 0, behavior depends upon the value of ECX.
    - If ECX contains 808H (indicating the TPR MSR), the instruction reads the 8 bytes from offset 080H on the virtual-APIC page (VTPR and the 4 bytes above it) into EDX:EAX. This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not x2APIC mode).
    - If ECX contains any other value in the range 800H–8FFH, the instruction operates normally. If the local APIC is in x2APIC mode and ECX indicates a readable APIC register, EDX and EAX are loaded with the value of that register. If the local APIC is not in x2APIC mode or ECX does not indicate a readable APIC register, a general-protection fault occurs.
  - If “APIC-register virtualization” is 1 and ECX contains a value in the range 800H–8FFH, the instruction reads the 8 bytes from offset X on the virtual-APIC page into EDX:EAX, where X = (ECX & FFH) « 4. This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not in x2APIC mode).

• **WRMSR.** The instruction’s behavior depends on the value of ECX and the setting of the “virtual-interrupt delivery” and “IPI virtualization” VM-execution controls.

Special processing applies in the following cases: (1) ECX contains 808H (indicating the TPR MSR); (2) ECX contains 80BH (indicating the EOI MSR) and the “virtual-interrupt delivery” VM-execution control is 1; (3) ECX contains 83FH (indicating the self-IPI MSR) and the “virtual-interrupt delivery” VM-execution control is 1; and (4) ECX contains 830H (indicating the ICR MSR) and the “IPI virtualization” VM-execution control is 1.

If special processing applies, no general-protection exception is produced due to the fact that the local APIC is in xAPIC mode. However, WRMSR does perform the normal reserved-bit checking:
  - If ECX contains 808H or 83FH, a general-protection fault occurs if either EDX or EAX[31:8] is non-zero.
  - If ECX contains 80BH, a general-protection fault occurs if either EDX or EAX is non-zero.
  - If ECX contains 830H, a general-protection fault occurs if any of bits 31:20, 17:16, or 13 of EAX is non-zero.

If there is no fault, WRMSR stores EDX:EAX at offset X on the virtual-APIC page, where X = (ECX & FFH) « 4. Following this, the processor performs an operation depending on the value of ECX:
  - If ECX contains 808H, the processor performs TPR virtualization (see Section 30.1.2).
  - If ECX contains 80BH, the processor performs EOI virtualization (see Section 30.1.4).
  - If ECX contains 83FH, the processor then checks the value of EAX[7:4] and proceeds as follows:
• If the value is non-zero, the logical processor performs self-IPI virtualization with the 8-bit vector in EAX[7:0] (see Section 30.1.5).
• If the value is zero, the logical processor causes an APIC-write VM exit as if there had been a write access to page offset 3F0H on the APIC-access page (see Section 30.4.3.3).
  — If ECX contains 830H, the processor then checks the value of VICR to determine whether the following are all true:
    • Bits 19:18 (destination shorthand) are 00B (no shorthand).
    • Bit 15 (trigger mode) is 0 (edge).
    • Bit 12 (unused) is 0.
    • Bit 11 (destination mode) is 0 (physical).
    • Bits 10:8 (delivery mode) are 000B (fixed).
    If all of the items above are true, the processor performs IPI virtualization using the 8-bit vector in byte 0 of VICR and the 32-bit APIC ID in VICR[63:32] (see Section 30.1.6). Otherwise, the logical processor causes an APIC-write VM exit (see Section 30.4.3.3).

If special processing does not apply, the instruction operates normally. If the local APIC is in x2APIC mode and ECX indicates a writable APIC register, the value in EDX:EAX is written to that register. If the local APIC is not in x2APIC mode or ECX does not indicate a writable APIC register, a general-protection fault occurs.

### 30.6 POSTED-INTERRUPT PROCESSING

Posted-interrupt processing is a feature by which a processor processes the virtual interrupts by recording them as pending on the virtual-APIC page.

Posted-interrupt processing is enabled by setting the "process posted interrupts" VM-execution control. The processing is performed in response to the arrival of an interrupt with the posted-interrupt notification vector.

In response to such an interrupt, the processor processes virtual interrupts recorded in a data structure called a posted-interrupt descriptor. The posted-interrupt notification vector and the address of the posted-interrupt descriptor are fields in the VMCS; see Section 25.6.8.

If the "process posted interrupts" VM-execution control is 1, a logical processor uses a 64-byte posted-interrupt descriptor located at the posted-interrupt descriptor address. The posted-interrupt descriptor has the following format:

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>255:0</td>
<td>Posted-interrupt requests</td>
<td>One bit for each interrupt vector. There is a posted-interrupt request for a vector if the corresponding bit is 1.</td>
</tr>
<tr>
<td>256</td>
<td>Outstanding notification</td>
<td>If this bit is set, there is a notification outstanding for one or more posted interrupts in bits 255:0.</td>
</tr>
<tr>
<td>511:257</td>
<td>Reserved for software and other agents</td>
<td>These bits may be used by software and by other agents in the system (e.g., chipset). The processor does not modify these bits.</td>
</tr>
</tbody>
</table>

The notation PIR (posted-interrupt requests) refers to the 256 posted-interrupt bits in the posted-interrupt descriptor.

Use of the posted-interrupt descriptor differs from that of other data structures that are referenced by pointers in a VMCS. There is a general requirement that software ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation. That requirement does not apply to the posted-interrupt descriptor. There is a requirement, however, that such modifications be done using locked read-modify-write instructions.
If the “external-interrupt exiting” VM-execution control is 1, any unmasked external interrupt causes a VM exit (see Section 26.2). If the “process posted interrupts” VM-execution control is also 1, this behavior is changed and the processor handles an external interrupt as follows:15

1. The local APIC is acknowledged; this provides the processor core with an interrupt vector, called here the **physical vector**.

2. If the physical vector equals the posted-interrupt notification vector, the logical processor continues to the next step. Otherwise, a VM exit occurs as it would normally due to an external interrupt; the vector is saved in the VM-exit interruption-information field.

3. The processor clears the outstanding-notification bit in the posted-interrupt descriptor. This is done atomically so as to leave the remainder of the descriptor unmodified (e.g., with a locked AND operation).

4. The processor writes zero to the EOI register in the local APIC; this dismisses the interrupt with the posted-interrupt notification vector from the local APIC.

5. The logical processor performs a logical-OR of PIR into VIRR and clears PIR. No other agent can read or write a PIR bit (or group of bits) between the time it is read (to determine what to OR into VIRR) and when it is cleared.

6. The logical processor sets RVI to be the maximum of the old value of RVI and the highest index of all bits that were set in PIR; if no bit was set in PIR, RVI is left unmodified.

7. The logical processor evaluates pending virtual interrupts as described in Section 30.2.1.

The logical processor performs the steps above in an uninterruptible manner. If step #7 leads to recognition of a virtual interrupt, the processor may deliver that interrupt immediately.

Steps #1 to #7 above occur when the interrupt controller delivers an unmasked external interrupt to the CPU core. The following items consider certain cases of interrupt delivery:

- Interrupt delivery can occur between iterations of a REP-prefixed instruction (after at least one iteration has completed but before all iterations have completed). If this occurs, the following items characterize processor state after posted-interrupt processing completes and before guest execution resumes:
  - RIP references the REP-prefixed instruction;
  - RCX, RSI, and RDI are updated to reflect the iterations completed; and
  - RFLAGS.RF = 1.

- Interrupt delivery can occur when the logical processor is in the active, HLT, or MWAIT states. If the logical processor had been in the active or MWAIT state before the arrival of the interrupt, it is in the active state following completion of step #7; if it had been in the HLT state, it returns to the HLT state after step #7 (if a pending virtual interrupt was recognized, the logical processor may immediately wake from the HLT state).

- Interrupt delivery can occur while the logical processor is in enclave mode. If the logical processor had been in enclave mode before the arrival of the interrupt, an Asynchronous Enclave Exit (AEX) may occur before the steps #1 to #7 (see Chapter 37, “Enclave Exiting Events”). If no AEX occurs before step #1 and a VM exit occurs at step #2, an AEX occurs before the VM exit is delivered.

### 30.7 VIRTUALIZING SENDUIPI

The user-interrupt feature includes the SENDUIPI instruction that software operating with CPL = 3 can use to send user interrupts to another software thread (“user IPIs”). The SENDUIPI instruction has the following high-level operation:

- read selected entry from user-interrupt target table;
- use address in entry to read the referenced user posted-interrupt descriptor (UPID);
- update certain fields in UPID;
- if necessary, send ordinary IPI indicated in UPID’s notification information;

15. VM entry ensures that the “process posted interrupts” VM-execution control is 1 only if the “external-interrupt exiting” VM-execution control is also 1. See Section 27.2.1.1.
The last step uses two fields in the UPID: an 8-bit notification vector (UPID.NV) and a 32-bit notification destination (an APIC ID, UPID.NDST). Outside of VMX non-root operation, the processor implements the last step as follows:

- If the local APIC is in xAPIC mode, it writes UPID.NDST[15:8] to ICR_HI[31:24] (offset 310H from IA32_APIC_BASE) and then writes UPID.NV to ICR_LO (offset 300H).
- If the local APIC is in x2APIC mode, it performs the control-register write that would be done by an execution of WRMSR with ECX = 310H (ICR), EAX = UPID.NV, and EDX = UPID.NDST.

In VMX non-root operation, implementation of the step depends on the settings of the "use TPR shadow,” “virtualize APIC accesses,” and "IPI virtualization” VM-execution controls:

1. If the “use TPR shadow” VM-execution control is 0, the behavior is not modified: the logical processor sends the specified IPI by writing to the local APIC’s ICR as specified above (based on the current mode of the local APIC).
2. If the “use TPR shadow” VM-execution control is 1 and the “virtualize APIC accesses” VM-execution control is 0, the logical processor virtualizes the sending of an x2APIC-mode IPI with the following steps:
   a. The 64-bit value Z is written to offset 300H on the virtual-APIC page (VICR), where Z[7:0] = UPID.NV (the 8-bit virtual vector), Z[63:32] = UPID.NDST (the 32-bit virtual APIC ID) and Z[31:8] = 000000H (indicating a physically addressed fixed-mode IPI).
   b. If the “IPI virtualization” VM-execution control is 1, IPI virtualization (Section 30.1.6) is performed using the vector UPID.NV and the 32-bit virtual APIC ID UPID.NDST.
3. If the “use TPR shadow” and “virtualize APIC accesses” VM-execution controls are both 1, the logical processor virtualizes the sending of an xAPIC-mode IPI by performing the following steps:
   a. The 32-bit value X is written to offset 310H on the virtual-APIC page (VICR_HI), where X[31:24] = UPID.NDST[15:8] (the 8-bit virtual APIC ID) and X[23:0] = 000000H.\(^\text{17}\)
   b. The 32-bit value Y is written to offset 300H on the virtual-APIC page (VICR_LO), where Y[7:0] = UPID.NV (the 8-bit virtual vector) and Y[31:8] = 0000000H (indicating a physically addressed fixed-mode IPI).
   c. If the “IPI virtualization” VM-execution control is 1, IPI virtualization is performed using the vector UPID.NV and the APIC ID UPID.NDST[15:8]. IPI virtualization will use only the 8-bit APIC ID from bits 15:8 of the UPID’s destination field (the 8-bit value written earlier to bits 31:24 of VICR_HI).
4. If the “use TPR shadow” VM-execution control is 1 and the “IPI virtualization” VM-execution control is 0, an APIC-write VM exit occurs as if there had been a write access to page offset 300H on the APIC-access page (see Section 30.4.3.3).

---

16. The setting of the “virtualize x2APIC mode” VM-execution control does not affect this operation.
17. For xAPIC mode (which is virtualized if the “virtualize APIC accesses” VM-execution control is 1), the destination APIC ID is in byte 1 (not byte 0) of the UPID’s 4-byte NDST field.
CHAPTER 31
VMX INSTRUCTION REFERENCE

31.1 OVERVIEW

This chapter describes the virtual-machine extensions (VMX) for the Intel 64 and IA-32 architectures. VMX is intended to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments. The virtual-machine extensions (VMX) includes five instructions that manage the virtual-machine control structure (VMCS), four instructions that manage VMX operation, two TLB-management instructions, and two instructions for use by guest software. Additional details of VMX are described in Chapter 24 through Chapter 30.

The behavior of the VMCS-maintenance instructions is summarized below:

- **VMPTRLD** — This instruction takes a single 64-bit source operand that is in memory. It makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the current VMCS based on the contents of VMCS-data area in the referenced VMCS region. Because this makes the referenced VMCS active, a logical processor may start maintaining on the processor some of the VMCS data for the VMCS.

- **VMPTRST** — This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.

- **VMCLEAR** — This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to "clear", renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.

- **VMREAD** — This instruction reads a component from a VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.

- **VMWRITE** — This instruction writes a component to a VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.

The behavior of the VMX management instructions is summarized below:

- **VMLAUNCH** — This instruction launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.

- **VMRESUME** — This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.

- **VMXOFF** — This instruction causes the processor to leave VMX operation.

- **VMXON** — This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

The behavior of the VMX-specific TLB-management instructions is summarized below:

- **INVEPT** — This instruction invalidates entries in the TLBs and paging-structure caches that were derived from extended page tables (EPT).

- **INVVPID** — This instruction invalidates entries in the TLBs and paging-structure caches based on a Virtual-Processor Identifier (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:

- **VMCALL** — This instruction allows software in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.

- **VMFUNC** — This instruction allows software in VMX non-root operation to invoke a VM function (processor functionality enabled and configured by software in VMX root operation) without a VM exit.
31.2 CONVENTIONS

The operation sections for the VMX instructions in Section 31.3 use the pseudo-function VMexit, which indicates that the logical processor performs a VM exit.

The operation sections also use the pseudo-functions VMsucceed, VMfail, VMfailInvalid, and VMfailValid. These pseudo-functions signal instruction success or failure by setting or clearing bits in RFLAGS and, in some cases, by writing the VM-instruction error field. The following pseudocode fragments detail these functions:

**VMsucceed:**
- CF := 0;
- PF := 0;
- AF := 0;
- ZF := 0;
- SF := 0;
- OF := 0;

**VMfail(ErrorNumber):**
- IF VMCS pointer is valid
  - THEN VMfailValid(ErrorNumber);
  - ELSE VMfailInvalid;
- FI;

**VMfailInvalid:**
- CF := 1;
- PF := 0;
- AF := 0;
- ZF := 0;
- SF := 0;
- OF := 0;

**VMfailValid(ErrorNumber):**
- // executed only if there is a current VMCS
- CF := 0;
- PF := 0;
- AF := 0;
- ZF := 1;
- SF := 0;
- OF := 0;
- Set the VM-instruction error field to ErrorNumber;

The different VM-instruction error numbers are enumerated in Section 31.4, "VM Instruction Error Numbers."

31.3 VMX INSTRUCTIONS

This section provides detailed descriptions of the VMX instructions.
INVEPT— Invalidate Translations Derived from EPT

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 80</td>
<td>RM</td>
<td>Invalidates EPT-derived entries in the TLBs and paging-structure caches (in 64-bit mode).</td>
</tr>
<tr>
<td>INVEPT r64, m128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66 0F 38 80</td>
<td>RM</td>
<td>Invalidates EPT-derived entries in the TLBs and paging-structure caches (outside 64-bit mode).</td>
</tr>
<tr>
<td>INVEPT r32, m128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches that were derived from extended page tables (EPT). (See Chapter 29, "VMX Support for Address Translation."). Invalidation is based on the INVEPT type specified in the register operand and the INVEPT descriptor specified in the memory operand.

Outside IA-32e mode, the register operand is always 32 bits, regardless of the value of CS.D; in 64-bit mode, the register operand has 64 bits (the instruction cannot be executed in compatibility mode).

The INVEPT types supported by a logical processors are reported in the IA32_VMX_EPT_VPID_CAP MSR (see Appendix A, "VMX Capability Reporting Facility"). There are two INVEPT types currently defined:

- Single-context invalidation: If the INVEPT type is 1, the logical processor invalidates all mappings associated with bits 51:12 of the EPT pointer (EPTP) specified in the INVEPT descriptor. It may invalidate other mappings as well.
- Global invalidation: If the INVEPT type is 2, the logical processor invalidates mappings associated with all EPTPs.

If an unsupported INVEPT type is specified, the instruction fails.

INVEPT invalidates all the specified mappings for the indicated EPTP(s) regardless of the VPID and PCID values with which those mappings may be associated.

The INVEPT descriptor comprises 128 bits and contains a 64-bit EPTP value in bits 63:0 (see Figure 31-1).

![Figure 31-1. INVEPT Descriptor](image)

Operation

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
ELSE
INVEPT_TYPE := value of register operand;
IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support INVEPT_TYPE
THEN VMfail(Invalid operand to INVEPT/INVVPID);
ELSE  // INVEPT_TYPE must be 1 or 2
    INVEPT_DESC := value of memory operand;
    EPTP := INVEPT_DESC[63:0];
    CASE INVEPT_TYPE OF
    1:  // single-context invalidation
        IF VM entry with the "enable EPT" VM execution control set to 1
            would fail due to the EPTP value
        THEN VMfail(Invalid operand to INVEPT/INVVPID);
        ELSE
            Invalidate mappings associated with EPTP[51:12];
            VMsucceed;
        FI;
    BREAK;
    2:  // global invalidation
        Invalidate mappings associated with all EPTPs;
        VMsucceed;
    BREAK;
    ESAC;
FI;
FI;

Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions
#GP(0) If the current privilege level is not 0.
    If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register contains an unusable segment.
    If the source operand is located in an execute-only code segment.
#PF(fault-code) If a page fault occurs in accessing the memory operand.
#SS(0) If the memory operand effective address is outside the SS segment limit.
    If the SS register contains an unusable segment.
#UD If not in VMX operation.
    If the logical processor does not support EPT (IA32_VMX_PROCBASED_CTLS2[33]=0).
    If the logical processor supports EPT (IA32_VMX_PROCBASED_CTLS2[33]=1) but does not
    support the INVEPT instruction (IA32_VMX_EPT_VPID_CAP[20]=0).

Real-Address Mode Exceptions
#UD The INVEPT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The INVEPT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The INVEPT instruction is not recognized in compatibility mode.
64-Bit Mode Exceptions

#GP(0)  If the current privilege level is not 0.
       If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is
       in a non-canonical form.

#PF(fault-code)  If a page fault occurs in accessing the memory operand.

#SS(0)  If the memory operand is in the SS segment and the memory address is in a non-canonical
       form.

#UD  If not in VMX operation.
       If the logical processor does not support EPT (IA32_VMX_PROCBASED_CTLS2[33]=0).
       If the logical processor supports EPT (IA32_VMX_PROCBASED_CTLS2[33]=1) but does not
       support the INVEPT instruction (IA32_VMX_EPT_VPID_CAP[20]=0).
VMX INSTRUCTION REFERENCE

INVVPID— Invalidate Translations Based on VPID

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 38 B1 INVVPID r64, m128</td>
<td>RM</td>
<td>Invalidates entries in the TLBs and paging-structure caches based on VPID (in 64-bit mode).</td>
</tr>
<tr>
<td>66 0F 38 B1 INVVPID r32, m128</td>
<td>RM</td>
<td>Invalidates entries in the TLBs and paging-structure caches based on VPID (outside 64-bit mode).</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on virtual-processor identifier (VPID). (See Chapter 29, "VMX Support for Address Translation.") Invalidation is based on the INVVPID type specified in the register operand and the INVVPID descriptor specified in the memory operand.

Outside IA-32e mode, the register operand is always 32 bits, regardless of the value of CS.D; in 64-bit mode, the register operand has 64 bits (the instruction cannot be executed in compatibility mode).

The INVVPID types supported by a logical processors are reported in the IA32_VMX_EPT_VPID_CAP MSR (see Appendix A, “VMX Capability Reporting Facility”). There are four INVVPID types currently defined:

- Individual-address invalidation: If the INVVPID type is 0, the logical processor invalidates mappings for the linear address and VPID specified in the INVVPID descriptor. In some cases, it may invalidate mappings for other linear addresses (or other VPIDs) as well.
- Single-context invalidation: If the INVVPID type is 1, the logical processor invalidates all mappings tagged with the VPID specified in the INVVPID descriptor. In some cases, it may invalidate mappings for other VPIDs as well.
- All-contexts invalidation: If the INVVPID type is 2, the logical processor invalidates all mappings tagged with all VPIDs except VPID 0000H. In some cases, it may invalidate translations with VPID 0000H as well.
- Single-context invalidation, retaining global translations: If the INVVPID type is 3, the logical processor invalidates all mappings tagged with the VPID specified in the INVVPID descriptor except global translations. In some cases, it may invalidate global translations (and mappings with other VPIDs) as well. See the "Caching Translation Information" section in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for information about global translations.

If an unsupported INVVPID type is specified, the instruction fails.

INVVPID invalidates all the specified mappings for the indicated VPID(s) regardless of the EPTP and PCID values with which those mappings may be associated.

The INVVPID descriptor comprises 128 bits and consists of a VPID and a linear address as shown in Figure 31-2.

![Figure 31-2. INVVPID Descriptor](image-url)
Operation

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
ELSE
    INVVPID_TYPE := value of register operand;
    IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support
        INVVPID_TYPE
        THEN VMfail(Invalid operand to INVEPT/INVVPID);
        ELSE // INVVPID_TYPE must be in the range 0–3
            INVVPID_DESC := value of memory operand;
            IF INVVPID_DESC[63:16] ≠ 0
                THEN VMfail(Invalid operand to INVEPT/INVVPID);
                ELSE
                    CASE INVVPID_TYPE OF
                        0: // individual-address invalidation
                            VPID := INVVPID_DESC[15:0];
                            IF VPID = 0
                                THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                ELSE
                                    GL_ADDR := INVVPID_DESC[127:64];
                                    IF (GL_ADDR is not in a canonical form)
                                        THEN
                                            VMfail(Invalid operand to INVEPT/INVVPID);
                                        ELSE
                                            Invalidate mappings for GL_ADDR tagged with VPID;
                                            VMsucceed;
                                    FI;
                            FI;
                        BREAK;
                        1: // single-context invalidation
                            VPID := INVVPID_DESC[15:0];
                            IF VPID = 0
                                THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                ELSE
                                    Invalidate all mappings tagged with VPID;
                                    VMsucceed;
                            FI;
                        BREAK;
                        2: // all-context invalidation
                            Invalidate all mappings tagged with all non-zero VPIDs;
                            VMsucceed;
                        BREAK;
                        3: // single-context invalidation retaining globals
                            VPID := INVVPID_DESC[15:0];
                            IF VPID = 0
                                THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                ELSE
                                    Invalidate all mappings tagged with VPID except global translations;
                                    VMsucceed;
Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- If the DS, ES, FS, or GS register contains an unusable segment.
- If the source operand is located in an execute-only code segment.
- #PF(fault-code) If a page fault occurs in accessing the memory operand.
- #SS(0) If the memory operand effective address is outside the SS segment limit.
- If the SS register contains an unusable segment.
- #UD If not in VMX operation.
- If the logical processor does not support VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=0).
- If the logical processor supports VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=1) but does not support the INVVPID instruction (IA32_VMX_EPT_VPID_CAP[32]=0).

Real-Address Mode Exceptions

- #UD The INVVPID instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

- #UD The INVVPID instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

- #UD The INVVPID instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs in accessing the memory operand.
- #SS(0) If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
- #UD If not in VMX operation.
- If the logical processor does not support VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=0).
- If the logical processor supports VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=1) but does not support the INVVPID instruction (IA32_VMX_EPT_VPID_CAP[32]=0).
VMCALL—Call to VM Monitor

**Opcode/Instruction** | **Op/En** | **Description**
--- | --- | ---
0F 01 C1 VMCALL | ZO | Call to VM monitor by causing VM exit.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction allows guest software to make a call for service into an underlying VM monitor. The details of the programming interface for such calls are VMM-specific; this instruction does nothing more than cause a VM exit, registering the appropriate exit reason.

Use of this instruction in VMX root operation invokes an SMM monitor (see Section 32.15.2). This invocation will activate the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM) if it is not already active (see Section 32.15.6).

**Operation**

IF not in VMX operation
  THEN #UD;
ELSIF in VMX non-root operation
  THEN VM exit;
ELSIF (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
  THEN #UD;
ELSIF CPL > 0
  THEN #GP(0);
ELSIF in SMM or the logical processor does not support the dual-monitor treatment of SMIs and SMM or the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
  THEN VMfail (VMCALL executed in VMX root operation);
ELSIF dual-monitor treatment of SMIs and SMM is active
  THEN perform an SMM VM exit (see Section 32.15.2);
ELSIF current-VMCS pointer is not valid
  THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
  THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields are not valid (see Section 32.15.6.1)
  THEN VMfailValid(VMCALL with invalid VM-exit control fields);
ELSE
  enter SMM;
  read revision identifier in MSEG;
  IF revision identifier does not match that supported by processor
    THEN
      leave SMM;
      VMfailValid(VMCALL with incorrect MSEG revision identifier);
    ELSE
      read SMM-monitor features field in MSEG (see Section 32.15.6.1);
      IF features field is invalid
        THEN
          leave SMM;
VMX failValid(VMCALL with invalid SMM-monitor features);
ELSE activate dual-monitor treatment of SMIs and SMM (see Section 32.15.6);
FI;
FI;
FI;

Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions
#GP(0) If the current privilege level is not 0 and the logical processor is in VMX root operation.
#UD If executed outside VMX operation.

Real-Address Mode Exceptions
#UD If executed outside VMX operation.

Virtual-8086 Mode Exceptions
#UD If executed outside VMX non-root operation.

Compatibility Mode Exceptions
#UD If executed outside VMX non-root operation.

64-Bit Mode Exceptions
#UD If executed outside VMX operation.
VMCLEAR—Clear Virtual-Machine Control Structure

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 OF C7 /6 VMCLEAR m64</td>
<td>M</td>
<td>Copy VMCS data to VMCS region in memory.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

This instruction applies to the VMCS whose VMCS region resides at the physical address contained in the instruction operand. The instruction ensures that VMCS data for that VMCS (some of these data may be currently maintained on the processor) are copied to the VMCS region in memory. It also initializes parts of the VMCS region (for example, it sets the launch state of that VMCS to clear). See Chapter 25, "Virtual Machine Control Structures."

The operand of this instruction is always 64 bits and is always in memory. If the operand is the current-VMCS pointer, then that pointer is made invalid (set to `FFFFFFFF_FFFFFFFF`).

Note that the VMCLEAR instruction might not explicitly write any VMCS data to memory; the data may be already resident in memory before the VMCLEAR is executed.

### Operation

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0) THEN #UD;
ELSIF in VMX non-root operation THEN VM exit;
ELSIF CPL > 0 THEN #GP(0);
ELSE
addr := contents of 64-bit in-memory operand;
IF addr is not 4KB-aligned OR
addr sets any bits beyond the physical-address width\(^1\)
THEN VMfail(VMCLEAR with invalid physical address);
ELSIF addr = VMXON pointer
THEN VMfail(VMCLEAR with VMXON pointer);
ELSE
ensure that data for VMCS referenced by the operand is in memory;
initialize implementation-specific data in VMCS region;
launch state of VMCS referenced by the operand := "clear"
IF operand addr = current-VMCS pointer
THEN current-VMCS pointer := `FFFFFFFF_FFFFFFFF';
FI;
VMSucceed;
FI;
FI;

### Flags Affected

See the operation section and Section 31.2.

---

1. If IA32_VMX_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix A.1.
Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the operand is located in an execute-only code segment.

#PF(fault-code) If a page fault occurs in accessing the memory operand.
#SS(0) If the memory operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.
#UD If operand is a register.
If not in VMX operation.

Real-Address Mode Exceptions

#UD The VMCLEAR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The VMCLEAR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The VMCLEAR instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is
in a non-canonical form.

#PF(fault-code) If a page fault occurs in accessing the memory operand.
#SS(0) If the source operand is in the SS segment and the memory address is in a non-canonical
form.
#UD If operand is a register.
If not in VMX operation.
VMFUNC—Invoke VM function

### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D4 VMFUNC</td>
<td>ZO</td>
<td>Invoke VM function specified in EAX.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. The value of EAX selects the specific VM function being invoked.

The behavior of each VM function (including any additional fault checking) is specified in Section 26.5.6, “VM Functions.”

### Operation

Perform functionality of the VM function specified in EAX;

### Flags Affected

Depends on the VM function specified in EAX. See Section 26.5.6, “VM Functions.”

### Protected Mode Exceptions (not including those defined by specific VM functions)

- **#UD**
  - If executed outside VMX non-root operation.
  - If “enable VM functions” VM-execution control is 0.
  - If EAX ≥ 64.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.
VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 C2 VMLAUNCH</td>
<td>ZO</td>
<td>Launch virtual machine managed by current VMCS.</td>
</tr>
<tr>
<td>0F 01 C3 VMRESUME</td>
<td>ZO</td>
<td>Resume virtual machine managed by current VMCS.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Effects a VM entry managed by the current VMCS.

- **VMLAUNCH** fails if the launch state of current VMCS is not "clear". If the instruction is successful, it sets the launch state to "launched."
- **VMRESUME** fails if the launch state of the current VMCS is not "launched."

If VM entry is attempted, the logical processor performs a series of consistency checks as detailed in Chapter 27, "VM Entries." Failure to pass checks on the VMX controls or on the host-state area passes control to the instruction following the VMLAUNCH or VMRESUME instruction. If these pass but checks on the guest-state area fail, the logical processor loads state from the host-state area of the VMCS, passing control to the instruction referenced by the RIP field in the host-state area.

VM entry is not allowed when events are blocked by MOV SS or POP SS. Neither VMLAUNCH nor VMRESUME should be used immediately after either MOV to SS or POP to SS.

**Operation**

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
THEN #UD;
ELSIF in VMX non-root operation
THEN VMexit;
ELSIF CPL > 0
THEN #GP(0);
ELSIF current-VMCS pointer is not valid
THEN VMfailInvalid;
ELSIF events are being blocked by MOV SS
THEN VMfailValid(VM entry with events blocked by MOV SS);
ELSIF (VMLAUNCH and launch state of current VMCS is not "clear")
THEN VMfailValid(VMLAUNCH with non-clear VMCS);
ELSIF (VMRESUME and launch state of current VMCS is not "launched")
THEN VMfailValid(VMRESUME with non-launched VMCS);
ELSE
Check settings of VMX controls and host-state area;
IF invalid settings
THEN VMfailValid(VM entry with invalid VMX-control field(s)) or
VMfailValid(VM entry with invalid host-state field(s)) or
VMfailValid(VM entry with invalid executive-VMCS pointer)) or
VMfailValid(VM entry with non-launched executive VMCS) or
VMfailValid(VM entry with executive-VMCS pointer not VMXON pointer) or

31-14 Vol. 3C
VMfailValid(VM entry with invalid VM-execution control fields in executive VMCS) as appropriate;
ELSE
   Attempt to load guest state and PDPTRs as appropriate;
   clear address-range monitoring;
   IF failure in checking guest state or PDPTRs
   THEN VM entry fails (see Section 27.8);
   ELSE
      Attempt to load MSRs from VM-entry MSR-load area;
      IF failure
      THEN VM entry fails
      (see Section 27.8);
      ELSE
         IF VMLAUNCH
            THEN launch state of VMCS := “launched”;
         FI;
         IF in SMM and "entry to SMM" VM-entry control is 0
         THEN
            IF "deactivate dual-monitor treatment" VM-entry control is 0
               THEN SMM-transfer VMCS pointer := current-VMCS pointer;
               FI;
            IF executive-VMCS pointer is VMXON pointer
               THEN current-VMCS pointer := VMCS-link pointer;
            ELSE current-VMCS pointer := executive-VMCS pointer;
         FI;
         leave SMM;
      FI;
      VM entry succeeds;
   FI;
FI;
Further details of the operation of the VM-entry appear in Chapter 27.

Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions
#GP(0) If the current privilege level is not 0.
#UD If executed outside VMX operation.

Real-Address Mode Exceptions
#UD The VMLAUNCH and VMRESUME instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The VMLAUNCH and VMRESUME instructions are not recognized in virtual-8086 mode.
Compatibility Mode Exceptions

#UD The VMLAUNCH and VMRESUME instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD If executed outside VMX operation.
VMPTRLD—Load Pointer to Virtual-Machine Control Structure

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C7 /6 VMPTRLD m64</td>
<td>M</td>
<td>Loads the current VMCS pointer from memory.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Marks the current-VMCS pointer valid and loads it with the physical address in the instruction operand. The instruction fails if its operand is not properly aligned, sets unsupported physical-address bits, or is equal to the VMXON pointer. In addition, the instruction fails if the 32 bits in memory referenced by the operand do not match the VMCS revision identifier supported by this processor.¹

The operand of this instruction is always 64 bits and is always in memory.

### Operation

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
THEN #UD;
ELSIF in VMX non-root operation
  THEN VMexit;
ELSIF CPL > 0
  THEN #GP(0);
ELSE
  addr := contents of 64-bit in-memory source operand;
  IF addr is not 4KB-aligned OR
    addr sets any bits beyond the physical-address width²
    THEN VMfail(VMPTRLD with invalid physical address);
  ELSIF addr = VMXON pointer
    THEN VMfail(VMPTRLD with VMXON pointer);
  ELSE
    rev := 32 bits located at physical address addr;
    IF rev[30:0] ≠ VMCS revision identifier supported by processor OR
      rev[31] = 1 AND processor does not support 1-setting of "VMCS shadowing"
      THEN VMfail(VMPTRLD with incorrect VMCS revision identifier);
    ELSE
      current-VMCS pointer := addr;
      VMsucceed;
  FI;
FI;

### Flags Affected

See the operation section and Section 31.2.

¹ Software should consult the VMX capability MSR VMX_BASIC to discover the VMCS revision identifier supported by this processor (see Appendix A, “VMX Capability Reporting Facility”).
² If IA32_VMX_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix A.1.
VMX INSTRUCTION REFERENCE

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the source operand is located in an execute-only code segment.

#PF(fault-code) If a page fault occurs in accessing the memory source operand.
#SS(0) If the memory source operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.

#UD
If operand is a register.
If not in VMX operation.

Real-Address Mode Exceptions

#UD The VMPTRLD instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The VMPTRLD instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The VMPTRLD instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.
If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.

#PF(fault-code) If a page fault occurs in accessing the memory source operand.
#SS(0) If the source operand is in the SS segment and the memory address is in a non-canonical form.

#UD
If operand is a register.
If not in VMX operation.
VMPTRST—Store Pointer to Virtual-Machine Control Structure

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F C7 /7</td>
<td>M</td>
<td>Stores the current VMCS pointer into memory.</td>
</tr>
<tr>
<td>VMPTRST m64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**
Stores the current-VMCS pointer into a specified memory address. The operand of this instruction is always 64 bits and is always in memory.

**Operation**

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
THEN #UD;
ELSIF in VMX non-root operation
THEN VMexit;
ELSIF CPL > 0
THEN #GP(0);
ELSE
64-bit in-memory destination operand := current-VMCS pointer;
VMsucceed;
FI;

**Flags Affected**
See the operation section and Section 31.2.

**Protected Mode Exceptions**

#GP(0) If the current privilege level is not 0.
If the memory destination operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the destination operand is located in a read-only data segment or any code segment.
#PF(fault-code) If a page fault occurs in accessing the memory destination operand.
#SS(0) If the memory destination operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.
#UD If operand is a register.
If not in VMX operation.

**Real-Address Mode Exceptions**

#UD The VMPTRST instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The VMPTRST instruction is not recognized in virtual-8086 mode.
Compatibility Mode Exceptions

#UD  The VMPTRST instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0)  If the current privilege level is not 0.
        If the destination operand is in the CS, DS, ES, FS, or GS segments and the memory address
        is in a non-canonical form.
#PF(fault-code)  If a page fault occurs in accessing the memory destination operand.
#SS(0)  If the destination operand is in the SS segment and the memory address is in a non-canonical
        form.
#UD  If operand is a register.
        If not in VMX operation.
VMREAD—Read Field from Virtual-Machine Control Structure

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 78 VMREAD r/m64, r64</td>
<td>MR</td>
<td>Reads a specified VMCS field (in 64-bit mode).</td>
</tr>
<tr>
<td>NP 0F 78 VMREAD r/m32, r32</td>
<td>MR</td>
<td>Reads a specified VMCS field (outside 64-bit mode).</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRMr/m (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Reads a specified field from a VMCS and stores it into a specified destination operand (register or memory). In VMX root operation, the instruction reads from the current VMCS. If executed in VMX non-root operation, the instruction reads from the VMCS referenced by the VMCS link pointer field in the current VMCS.

The VMCS field is specified by the VMCS-field encoding contained in the register source operand. Outside IA-32e mode, the source operand has 32 bits, regardless of the value of CS.D. In 64-bit mode, the source operand has 64 bits.

The effective size of the destination operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the source operand is shorter than this effective operand size, the high bits of the destination operand are cleared to 0. If the VMCS field is longer, then the high bits of the field are not read.

Note that any faults resulting from accessing a memory destination operand can occur only after determining, in the operation section below, that the relevant VMCS pointer is valid and that the specified VMCS field is supported.

### Operation

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
THEN #UD;
ELSIF in VMX non-root operation AND (“VMCS shadowing” is 0 OR source operand sets bits in range 63:15 OR VMREAD bit corresponding to bits 14:0 of source operand is 1)\(^1\)
THEN VMexit;
ELSIF CPL > 0
THEN #GP(0);
ELSIF (in VMX root operation AND current-VMCS pointer is not valid) OR (in VMX non-root operation AND VMCS link pointer is not valid)
THEN VMfailInvalid;
ELSIF source operand does not correspond to any VMCS field
THEN VMfailInvalid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSE
IF in VMX root operation
THEN destination operand := contents of field indexed by source operand in current VMCS;
ELSE destination operand := contents of field indexed by source operand in VMCS referenced by VMCS link pointer;
FI;
VMsucceed;
FI;

\(^1\) The VMREAD bit for a source operand is defined as follows. Let \(x\) be the value of bits 14:0 of the source operand and let \(addr\) be the VMREAD-bitmap address. The corresponding VMREAD bit is in bit position \(x \& 7\) of the byte at physical address \(addr \mid (x \gg 3)\).
Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions
#GP(0) If the current privilege level is not 0.
   If a memory destination operand effective address is outside the CS, DS, ES, FS, or GS
   segment limit.
   If the DS, ES, FS, or GS register contains an unusable segment.
   If the destination operand is located in a read-only data segment or any code segment.
#PF(fault-code) If a page fault occurs in accessing a memory destination operand.
#SS(0) If a memory destination operand effective address is outside the SS segment limit.
   If the SS register contains an unusable segment.
#UD If not in VMX operation.

Real-Address Mode Exceptions
#UD The VMREAD instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The VMREAD instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The VMREAD instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions
#GP(0) If the current privilege level is not 0.
   If the memory destination operand is in the CS, DS, ES, FS, or GS segments and the memory
   address is in a non-canonical form.
#PF(fault-code) If a page fault occurs in accessing a memory destination operand.
#SS(0) If the memory destination operand is in the SS segment and the memory address is in a non-
   canonical form.
#UD If not in VMX operation.
VMRESUME—Resume Virtual Machine

See VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine.
VMWRITE—Write Field to Virtual-Machine Control Structure

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP OF 79 VMWRITE r64, r/m64</td>
<td>RM</td>
<td>Writes a specified VMCS field (in 64-bit mode).</td>
</tr>
<tr>
<td>NP OF 79 VMWRITE r32, r/m32</td>
<td>RM</td>
<td>Writes a specified VMCS field (outside 64-bit mode).</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Writes the contents of a primary source operand (register or memory) to a specified field in a VMCS. In VMX root operation, the instruction writes to the current VMCS. If executed in VMX non-root operation, the instruction writes to the VMCS referenced by the VMCS link pointer field in the current VMCS.

The VMCS field is specified by the VMCS-field encoding contained in the register secondary source operand. Outside IA-32e mode, the secondary source operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode, the secondary source operand has 64 bits.

The effective size of the primary source operand, which may be a register or in memory, is always 32 bits outside IA-32e mode (the setting of CS.D is ignored with respect to operand size) and 64 bits in 64-bit mode. If the VMCS field specified by the secondary source operand is shorter than this effective operand size, the high bits of the primary source operand are ignored. If the VMCS field is longer, then the high bits of the field are cleared to 0.

Note that any faults resulting from accessing a memory source operand occur after determining, in the operation section below, that the relevant VMCS pointer is valid but before determining if the destination VMCS field is supported.

Operation

IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation AND (“VMCS shadowing” is 0 OR secondary source operand sets bits in range 63:15 OR VMWRITE bit corresponding to bits 14:0 of secondary source operand is 1)1
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF (in VMX root operation AND current-VMCS pointer is not valid) OR (in VMX non-root operation AND VMCS-link pointer is not valid)
    THEN VMfailInvalid;
ELSIF secondary source operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSIF VMCS field indexed by secondary source operand is a VM-exit information field AND processor does not support writing to such fields2
    THEN VMfailValid(VMWRITE to read-only VMCS component);
ELSE

1. The VMWRITE bit for a secondary source operand is defined as follows. Let \( x \) be the value of bits 14:0 of the secondary source operand and let \( addr \) be the VMWRITE-bitmap address. The corresponding VMWRITE bit is in bit position \( x \mod 8 \) of the byte at physical address \( addr \cdot (x \gg 3) \).
2. Software can discover whether these fields can be written by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).
IF in VMX root operation
THEN field indexed by secondary source operand in current VMCS := primary source operand;
ELSE field indexed by secondary source operand in VMCS referenced by VMCS link pointer := primary source operand;
FI;
VMsucceed;
FI;

Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions
#GP(0) If the current privilege level is not 0.
If a memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the source operand is located in an execute-only code segment.
#PF(fault-code) If a page fault occurs in accessing a memory source operand.
#SS(0) If a memory source operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.
#UD If not in VMX operation.

Real-Address Mode Exceptions
#UD The VMWRITE instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The VMWRITE instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD The VMWRITE instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions
#GP(0) If the current privilege level is not 0.
If the memory source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs in accessing a memory source operand.
#SS(0) If the memory source operand is in the SS segment and the memory address is in a non-canonical form.
#UD If not in VMX operation.
VMXOFF—Leave VMX Operation

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 C4 VMXOFF</td>
<td>ZO</td>
<td>Leaves VMX operation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>ZO</td>
</tr>
</tbody>
</table>

Description
Takes the logical processor out of VMX operation, unblocks INIT signals, conditionally re-enables A20M, and clears any address-range monitoring.1

Operation
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
THEN #UD;
ELSIF in VMX non-root operation
THEN VMexit;
ELSIF CPL > 0
THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
THEN VMfail(VMXOFF under dual-monitor treatment of SMIs and SMM);
ELSE
leave VMX operation;
unblock INIT;
IF IA32_SMM_MONITOR_CTL[2] = 02
THEN unblock SMIs;
IF outside SMX operation3
THEN unblock and enable A20M;
FI;
clear address-range monitoring;
VMsucceed;
FI;

Flags Affected
See the operation section and Section 31.2.

Protected Mode Exceptions
#GP(0) If executed in VMX root operation with CPL > 0.

---

1. See the information on MONITOR/MWAIT in Chapter 9, “Multiple-Processor Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
2. Setting IA32_SMM_MONITOR_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register’s value bit (bit 0). Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine whether this is allowed.
3. A logical processor is outside SMX operation if GETSEC[SENSOR] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENSOR]. See Chapter 6, “Safer Mode Extensions Reference.”
#UD If executed outside VMX operation.

**Real-Address Mode Exceptions**

#UD The VMXOFF instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The VMXOFF instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD The VMXOFF instruction is not recognized in compatibility mode.

**64-Bit Mode Exceptions**

#GP(0) If executed in VMX root operation with CPL > 0.

#UD If executed outside VMX operation.
VMXON—Enter VMX Operation

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 0F C7 /6 VMXON m64</td>
<td>M</td>
<td>Enter VMX root operation.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Puts the logical processor in VMX operation with no current VMCS, blocks INIT signals, disables A20M, and clears any address-range monitoring established by the MONITOR instruction.1

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory.

**Operation**

IF (register operand) or (CR0.PE = 0) or (CR4.VMXE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0) THEN #UD;
ELSIF not in VMX operation
THEN
IF (CPL > 0) or (in A20M mode) or (the values of CR0 and CR4 are not supported in VMX operation; see Section 24.8) or (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or (in SMX operation2 and bit 1 of IA32_FEATURE_CONTROL MSR is clear) or (outside SMX operation and bit 2 of IA32_FEATURE_CONTROL MSR is clear)
THEN #GP(0);
ELSE
addr := contents of 64-bit in-memory source operand;
IF addr is not 4KB-aligned or addr sets any bits beyond the physical-address width3
THEN VMfailInvalid;
ELSE
rev := 32 bits located at physical address addr;
IF rev[30:0] ≠ VMCS revision identifier supported by processor OR rev[31] = 1
THEN VMfailInvalid;
ELSE
current-VMCS pointer := FFFFFFFF_FFFFFFFF;
enter VMX operation;
block INIT signals;
block and disable A20M;

---

1. See the information on MONITOR/MWAIT in Chapter 9, “Multiple-Processor Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

2. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, “Safer Mode Extensions Reference.”

3. If IA32_VMX_BASIC[48] is read as 1, VMfailInvalid occurs if addr sets any bits in the range 63:32; see Appendix A.1.
clear address-range monitoring;
IF the processor supports Intel PT but does not allow it to be used in VMX operation\(^1\)
THEN IA32_RTIT_CTL.TraceEn := 0;
FI;
VMsucceed;
FI;

ELSIF in VMX non-root operation
THEN VMexit;
ELSIF CPL > 0
THEN #GP(0);
ELSE VMfail("VMXON executed in VMX root operation");
FI;

**Flags Affected**
See the operation section and Section 31.2.

**Protected Mode Exceptions**

#GP(0) If executed outside VMX operation with CPL>0 or with invalid CR0 or CR4 fixed bits.
If executed in A20M mode.
If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains an unusable segment.
If the source operand is located in an execute-only code segment.
If the value of the IA32_FEATURE_CONTROL MSR does not support entry to VMX operation in
the current processor mode.

#PF(fault-code) If a page fault occurs in accessing the memory source operand.

#SS(0) If the memory source operand effective address is outside the SS segment limit.
If the SS register contains an unusable segment.

#UD If operand is a register.
If executed with CR4.VMXE = 0.

**Real-Address Mode Exceptions**

#UD The VMXON instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The VMXON instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD The VMXON instruction is not recognized in compatibility mode.

**64-Bit Mode Exceptions**

#GP(0) If executed outside VMX operation with CPL > 0 or with invalid CR0 or CR4 fixed bits.
If executed in A20M mode.
If the source operand is in the CS, DS, ES, FS, or GS segments and the memory address is in
a non-canonical form.

\(^1\) Software should read the VMX capability MSR IA32_VMX_MISC to determine whether the processor allows Intel PT to be used in
VMX operation (see Appendix A.6).
If the value of the IA32_FEATURE_CONTROL MSR does not support entry to VMX operation in the current processor mode.

#PF(fault-code)  If a page fault occurs in accessing the memory source operand.

#SS(0)  If the source operand is in the SS segment and the memory address is in a non-canonical form.

#UD  If operand is a register.

If executed with CR4.VMXE = 0.
### 31.4 VM INSTRUCTION ERROR NUMBERS

For certain error conditions, the VM-instruction error field is loaded with an error number to indicate the source of the error. Table 31-1 lists VM-instruction error numbers.

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VMCALL executed in VMX root operation</td>
</tr>
<tr>
<td>2</td>
<td>VMCLEAR with invalid physical address</td>
</tr>
<tr>
<td>3</td>
<td>VMCLEAR with VMXON pointer</td>
</tr>
<tr>
<td>4</td>
<td>VMLAUNCH with non-clear VMCS</td>
</tr>
<tr>
<td>5</td>
<td>VMRESUME with non-launched VMCS</td>
</tr>
<tr>
<td>6</td>
<td>VMRESUME after VMXOFF (VMXOFF and VMXON between VMLAUNCH and VMRESUME)</td>
</tr>
<tr>
<td>7</td>
<td>VM entry with invalid control field(s)</td>
</tr>
<tr>
<td>8</td>
<td>VM entry with invalid host-state field(s)</td>
</tr>
<tr>
<td>9</td>
<td>VMPTRLD with invalid physical address</td>
</tr>
<tr>
<td>10</td>
<td>VMPTRLD with VMXON pointer</td>
</tr>
<tr>
<td>11</td>
<td>VMPTRLD with incorrect VMCS revision identifier</td>
</tr>
<tr>
<td>12</td>
<td>VMREAD/VMWRITE from/to unsupported VMCS component</td>
</tr>
<tr>
<td>13</td>
<td>VMWRITE to read-only VMCS component</td>
</tr>
<tr>
<td>15</td>
<td>VMXON executed in VMX root operation</td>
</tr>
<tr>
<td>16</td>
<td>VM entry with invalid executive-VMCS pointer</td>
</tr>
<tr>
<td>17</td>
<td>VM entry with non-launched executive VMCS</td>
</tr>
<tr>
<td>18</td>
<td>VM entry with executive-VMCS pointer not VMXON pointer (when attempting to deactivate the dual-monitor treatment of SMI and SMM)</td>
</tr>
<tr>
<td>19</td>
<td>VMCALL with non-clear VMCS (when attempting to activate the dual-monitor treatment of SMI and SMM)</td>
</tr>
<tr>
<td>20</td>
<td>VMCALL with invalid VM-exit control fields</td>
</tr>
<tr>
<td>22</td>
<td>VMCALL with incorrect MSEG revision identifier (when attempting to activate the dual-monitor treatment of SMI and SMM)</td>
</tr>
<tr>
<td>23</td>
<td>VMXOFF under dual-monitor treatment of SMI and SMM</td>
</tr>
<tr>
<td>24</td>
<td>VM entry with invalid SMM-monitor features (when attempting to activate the dual-monitor treatment of SMI and SMM)</td>
</tr>
<tr>
<td>25</td>
<td>VM entry with invalid VM-execution control fields in executive VMCS (when attempting to return from SMM)</td>
</tr>
<tr>
<td>26</td>
<td>VM entry with events blocked by MOV SS.</td>
</tr>
<tr>
<td>28</td>
<td>Invalid operand to INVEPT/INVVPID.</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Earlier versions of this manual described this error as “VMRESUME with a corrupted VMCS”.
2. VM-entry checks on control fields and host-state fields may be performed in any order. Thus, an indication by error number of one cause does not imply that there are no other errors. Different processors may give different error numbers for the same VMCS.
3. Error number 7 is not used for VM entries that return from SMM that fail due to invalid VM-execution control fields in the executive VMCS. Error number 25 is used for these cases.
This chapter describes aspects of IA-64 and IA-32 architecture used in system management mode (SMM).

SMM provides an alternate operating environment that can be used to monitor and manage various system resources for more efficient energy usage, to control system hardware, and/or to run proprietary code. It was introduced into the IA-32 architecture in the Intel386 SL processor (a mobile specialized version of the Intel386 processor). It is also available in the Pentium M, Pentium 4, Intel Xeon, P6 family, and Pentium and Intel486 processors (beginning with the enhanced versions of the Intel486 SL and Intel486 processors).

### 32.1 SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

When SMM is invoked through a system management interrupt (SMI), the processor saves the current state of the processor (the processor’s context), then switches to a separate operating environment defined by a new address space. The system management software executive (SMI handler) starts execution in that environment, and the critical code and data of the SMI handler reside in a physical memory region (SMRAM) within that address space. While in SMM, the processor executes SMI handler code to perform operations such as powering down unused disk drives or monitors, executing proprietary code, or placing the whole system in a suspended state. When the SMI handler has completed its operations, it executes a resume (RSM) instruction. This instruction causes the processor to reload the saved context of the processor, switch back to protected or real mode, and resume executing the interrupted application or operating-system program or task.

The following SMM mechanisms make it transparent to applications programs and operating systems:

- The only way to enter SMM is by means of an SMI.
- The processor executes SMM code in a separate address space that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.
- All interrupts normally handled by the operating system are disabled upon entry into SMM.
- The RSM instruction can be executed only in SMM.

Section 32.3 describes transitions into and out of SMM. The execution environment after entering SMM is in real-address mode with paging disabled (CR0.PE = CR0.PG = 0). In this initial execution environment, the SMI handler can address up to 4 GBytes of memory and can execute all I/O and system instructions. Section 32.5 describes in detail the initial SMM execution environment for an SMI handler and operation within that environment. The SMI handler may subsequently switch to other operating modes while remaining in SMM.

**NOTES**

Software developers should be aware that, even if a logical processor was using the physical-address extension (PAE) mechanism (introduced in the P6 family processors) or was in IA-32e mode before an SMI, this will not be the case after the SMI is delivered. This is because delivery of an SMI disables paging (see Table 32-4). (This does not apply if the dual-monitor treatment of SMIs and SMM is active; see Section 32.15.)
32.1.1 System Management Mode and VMX Operation

Traditionally, SMM services system management interrupts and then resumes program execution (back to the software stack consisting of executive and application software; see Section 32.2 through Section 32.13).

A virtual machine monitor (VMM) using VMX can act as a host to multiple virtual machines and each virtual machine can support its own software stack of executive and application software. On processors that support VMX, virtual-machine extensions may use system-management interrupts (SMIs) and system-management mode (SMM) in one of two ways:

- **Default treatment.** System firmware handles SMIs. The processor saves architectural states and critical states relevant to VMX operation upon entering SMM. When the firmware completes servicing SMIs, it uses RSM to resume VMX operation.
- **Dual-monitor treatment.** Two VM monitors collaborate to control the servicing of SMIs: one VMM operates outside of SMM to provide basic virtualization in support for guests; the other VMM operates inside SMM (while in VMX operation) to support system-management functions. The former is referred to as **executive monitor**, the latter **SMM-transfer monitor (STM).**

The default treatment is described in Section 32.14, “Default Treatment of SMIs and SMM with VMX Operation and SMX Operation.” Dual-monitor treatment of SMM is described in Section 32.15, “Dual-Monitor Treatment of SMIs and SMM.”

32.2 System Management Interrupt (SMI)

The only way to enter SMM is by signaling an SMI through the SMI# pin on the processor or through an SMI message received through the APIC bus. The SMI is a nonmaskable external interrupt that operates independently from the processor’s interrupt- and exception-handling mechanism and the local APIC. The SMI takes precedence over an NMI and a maskable interrupt. SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

**NOTES**

In the Pentium 4, Intel Xeon, and P6 family processors, when a processor that is designated as an application processor during an MP initialization sequence is waiting for a startup IPI (SIPI), it is in a mode where SMIs are masked. However if a SMI is received while an application processor is in the wait for SIPI mode, the SMI will be pended. The processor then responds on receipt of a SIPI by immediately servicing the pended SMI and going into SMM before handling the SIPI.

An SMI may be blocked for one instruction following execution of STI, MOV to SS, or POP into SS.

32.3 Switching Between SMM and the Other Processor Operating Modes

Figure 2-3 shows how the processor moves between SMM and the other processor operating modes (protected, real-address, and virtual-8086). Signaling an SMI while the processor is in real-address, protected, or virtual-8086 modes always causes the processor to switch to SMM. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

32.3.1 Entering SMM

The processor always handles an SMI on an architecturally defined “interruptible” point in program execution (which is commonly at an IA-32 architecture instruction boundary). When the processor receives an SMI, it waits for all instructions to retire and for all stores to complete. The processor then saves its current context in SMRAM (see Section 32.4), enters SMM, and begins to execute the SMI handler.

1. The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1) to determine whether it is supported.
Upon entering SMM, the processor signals external hardware that SMI handling has begun. The signaling mechanism used is implementation dependent. For the P6 family processors, an SMI acknowledgment transaction is generated on the system bus and the multiplexed status signal EXF4 is asserted each time a bus transaction is generated while the processor is in SMM. For the Pentium and Intel486 processors, the SMIACT# pin is asserted.

An SMI has a greater priority than debug exceptions and external interrupts. Thus, if an NMI, maskable hardware interrupt, or a debug exception occurs at an instruction boundary along with an SMI, only the SMI is handled. Subsequent SMI requests are not acknowledged while the processor is in SMM. The first SMI interrupt request that occurs while the processor is in SMM (that is, after SMM has been acknowledged to external hardware) is latched and serviced when the processor exits SMM with the RSM instruction. The processor will latch only one SMI while in SMM.

See Section 32.5 for a detailed description of the execution environment when in SMM.

### 32.3.2 Exiting From SMM

The only way to exit SMM is to execute the RSM instruction. The RSM instruction is only available to the SMI handler; if the processor is not in SMM, attempts to execute the RSM instruction result in an invalid-opcode exception (#UD) being generated.

The RSM instruction restores the processor’s context by loading the state save image from SMRAM back into the processor’s registers. The processor then returns an SMIACK transaction on the system bus and returns program control back to the interrupted program.

#### NOTE

On processors that support the shadow-stack feature, RSM loads the SSP register from the state save image in SMRAM (see Table 32-3). The value is made canonical by sign-extension before loading it into SSP.

Upon successful completion of the RSM instruction, the processor signals external hardware that SMM has been exited. For the P6 family processors, an SMI acknowledge transaction is generated on the system bus and the multiplexed status signal EXF4 is no longer generated on bus cycles. For the Pentium and Intel486 processors, the SMIACT# pin is deserted.

If the processor detects invalid state information saved in the SMRAM, it enters the shutdown state and generates a special bus cycle to indicate it has entered shutdown state. Shutdown happens only in the following situations:

- A reserved bit in control register CR4 is set to 1 on a write to CR4. This error should not happen unless SMI handler code modifies reserved areas of the SMRAM saved state map (see Section 32.4.1). CR4 is saved in the state map in a reserved location and cannot be read or modified in its saved state.
- An illegal combination of bits is written to control register CR0, in particular PG set to 1 and PE set to 0, or NW set to 1 and CD set to 0.
- CR4.PCIDE would be set to 1 and IA32_EFER.LMA to 0.
- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.
- CR4.CET would be set to 1 and CR0.WP to 0.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined. On Pentium 4 and later processors, shutdown will inhibit INTR and A20M but will not change any of the other inhibits. On these processors, NMIs will be inhibited if no action is taken in the SMI handler to uninhibit them (see Section 32.8).

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 32.10). Also, the SMBASE address can be changed on a return from SMM (see Section 32.11).
32.4 SMRAM

Upon entering SMM, the processor switches to a new address space. Because paging is disabled upon entering SMM, this initial address space maps all memory accesses to the low 4 GBytes of the processor's physical address space. The SMI handler's critical code and data reside in a memory region referred to as system-management RAM (SMRAM). The processor uses a pre-defined region within SMRAM to save the processor's pre-SMI context. SMRAM can also be used to store system management information (such as the system configuration and specific information about powered-down devices) and OEM-specific information.

The default SMRAM size is 64 KBytes beginning at a base physical address in physical memory called the SMBASE (see Figure 32-1). The SMBASE default value following a hardware reset is 30000H. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 8000H]. It stores the processor's state in the area from [SMBASE + FE00H] to [SMBASE + FFFFH]. See Section 32.4.1 for a description of the mapping of the state save area.

The system logic is minimally required to decode the physical address range for the SMRAM from [SMBASE + 8000H] to [SMBASE + FFFFH]. A larger area can be decoded if needed. The size of this SMRAM can be between 32 KBytes and 4 GBytes.

The location of the SMRAM can be changed by changing the SMBASE value (see Section 32.11). It should be noted that all processors in a multiple-processor system are initialized with the same SMBASE value (30000H). Initialization software must sequentially place each processor in SMM and change its SMBASE so that it does not overlap those of other processors.

The actual physical location of the SMRAM can be in system memory or in a separate RAM memory. The processor generates an SMI acknowledge transaction (P6 family processors) or asserts the SMIACT# pin (Pentium and Intel486 processors) when the processor receives an SMI (see Section 32.3.1).

System logic can use the SMI acknowledge transaction or the assertion of the SMIACT# pin to decode accesses to the SMRAM and redirect them (if desired) to specific SMRAM memory. If a separate RAM memory is used for SMRAM, system logic should provide a programmable method of mapping the SMRAM into system memory space when the processor is not in SMM. This mechanism will enable start-up procedures to initialize the SMRAM space (that is, load the SMI handler) before executing the SMI handler during SMM.

32.4.1 SMRAM State Save Map

When an IA-32 processor that does not support Intel 64 architecture initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area begins at [SMBASE + 8000H + 7FFFH] and extends down to [SMBASE + 8000H + 7E00H]. Table 32-1 shows the state save map. The offset in column 1 is relative to the SMBASE value plus 8000H. Reserved spaces should not be used by software.

Some of the registers in the SMRAM state save area (marked YES in column 3) may be read and changed by the SMI handler, with the changed values restored to the processor registers by the RSM instruction. Some register images are read-only, and must not be modified (modifying these registers will result in unpredictable behavior). An SMI handler should not rely on any values stored in an area that is marked as reserved.
The following registers are saved (but not readable) and restored upon exiting SMM:

- Control register CR4. (This register is cleared to all 0s when entering SMM).
- The hidden segment descriptor information stored in segment registers CS, DS, ES, FS, GS, and SS.

If an SMI request is issued for the purpose of powering down the processor, the values of all reserved locations in the SMM state save must be saved to nonvolatile memory.

The following state is not automatically saved and restored following an SMI and the RSM instruction, respectively:

### Table 32-1. SMRAM State Save Map

<table>
<thead>
<tr>
<th>Offset (Added to SMBASE + 8000H)</th>
<th>Register</th>
<th>Writable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFCH</td>
<td>CR0</td>
<td>No</td>
</tr>
<tr>
<td>7FF8H</td>
<td>CR3</td>
<td>No</td>
</tr>
<tr>
<td>7FF4H</td>
<td>EFLAGS</td>
<td>Yes</td>
</tr>
<tr>
<td>7FF0H</td>
<td>EIP</td>
<td>Yes</td>
</tr>
<tr>
<td>7FEC8H</td>
<td>EDI</td>
<td>Yes</td>
</tr>
<tr>
<td>7FEB8H</td>
<td>ESI</td>
<td>Yes</td>
</tr>
<tr>
<td>7FE4H</td>
<td>EBP</td>
<td>Yes</td>
</tr>
<tr>
<td>7FEO8H</td>
<td>ESP</td>
<td>Yes</td>
</tr>
<tr>
<td>7FDCH</td>
<td>EBX</td>
<td>Yes</td>
</tr>
<tr>
<td>7FD8H</td>
<td>EDX</td>
<td>Yes</td>
</tr>
<tr>
<td>7D4H</td>
<td>ECX</td>
<td>Yes</td>
</tr>
<tr>
<td>7F00H</td>
<td>EAX</td>
<td>Yes</td>
</tr>
<tr>
<td>7FC8H</td>
<td>DR6</td>
<td>No</td>
</tr>
<tr>
<td>7FC4H</td>
<td>DR7</td>
<td>No</td>
</tr>
<tr>
<td>7FC4H</td>
<td>TR1</td>
<td>No</td>
</tr>
<tr>
<td>7FC0H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7FBCH</td>
<td>GS1</td>
<td>No</td>
</tr>
<tr>
<td>7FB8H</td>
<td>FS1</td>
<td>No</td>
</tr>
<tr>
<td>7FB4H</td>
<td>DS1</td>
<td>No</td>
</tr>
<tr>
<td>7FBO8H</td>
<td>SS1</td>
<td>No</td>
</tr>
<tr>
<td>7FACH</td>
<td>CS1</td>
<td>No</td>
</tr>
<tr>
<td>7FA8H</td>
<td>ES1</td>
<td>No</td>
</tr>
<tr>
<td>7FA4H</td>
<td>I/O State Field, see Section 32.7</td>
<td>No</td>
</tr>
<tr>
<td>7FAO8H</td>
<td>I/O Memory Address Field, see Section 32.7</td>
<td>No</td>
</tr>
<tr>
<td>7F9FH-7F03H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7FO2H</td>
<td>Auto HALT Restart Field (Word)</td>
<td>Yes</td>
</tr>
<tr>
<td>7F00H</td>
<td>I/O Instruction Restart Field (Word)</td>
<td>Yes</td>
</tr>
<tr>
<td>7FCH</td>
<td>SMM Revision Identifier Field (Doubleword)</td>
<td>No</td>
</tr>
<tr>
<td>7EF8H</td>
<td>SMBASE Field (Doubleword)</td>
<td>Yes</td>
</tr>
<tr>
<td>7E7H-7EO0H</td>
<td>Reserved</td>
<td>No</td>
</tr>
</tbody>
</table>

**NOTE:**
1. The two most significant bytes are reserved.

Vol. 3C 32-5
SYSTEM MANAGEMENT MODE

- Debug registers DR0 through DR3.
- The x87 FPU registers.
- The MTRRs.
- Control register CR2.
- The model-specific registers (for the P6 family and Pentium processors) or test registers TR3 through TR7 (for the Pentium and Intel486 processors).
- The state of the trap controller.
- The machine-check architecture registers.
- The APIC internal interrupt state (ISR, IRR, etc.).
- The microcode update state.

If an SMI is used to power down the processor, a power-on reset will be required before returning to SMM, which will reset much of this state back to its default values. So an SMI handler that is going to trigger power down should first read these registers listed above directly, and save them (along with the rest of RAM) to nonvolatile storage. After the power-on reset, the continuation of the SMI handler should restore these values, along with the rest of the system's state. Anytime the SMI handler changes these registers in the processor, it must also save and restore them.

NOTES
A small subset of the MSRs (such as, the time-stamp counter and performance-monitoring counters) are not arbitrarily writable and therefore cannot be saved and restored. SMM-based power-down and restoration should only be performed with operating systems that do not use or rely on the values of these registers.
Operating system developers should be aware of this fact and ensure that their operating-system assisted power-down and restoration software is immune to unexpected changes in these register values.

32.4.1.1 SMRAM State Save Map and Intel 64 Architecture

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area on an Intel 64 processor at [SMBASE + 8000H + 7FFFH] and extends to [SMBASE + 8000H + 7C00H].

Support for Intel 64 architecture is reported by CPUID.80000001:EDX[29] = 1. The layout of the SMRAM state save map is shown in Table 32-3.

Additionally, the SMRAM state save map shown in Table 32-3 also applies to processors with the following CPUID signatures listed in Table 32-2, irrespective of the value in CPUID.80000001:EDX[29].

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Processor Families/Processor Number Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_17H</td>
<td>Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processor Q9xxx, Intel Core 2 Duo processors E8000, T9000,</td>
</tr>
<tr>
<td>06_0FH</td>
<td>Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad, Intel Core 2 Extreme, Intel Core 2 Duo processors, Intel Pentium dual-core processors</td>
</tr>
<tr>
<td>06_1CH</td>
<td>45 nm Intel Atom® processors</td>
</tr>
</tbody>
</table>

Table 32-2. Processor Signatures and 64-bit SMRAM State Save Map Format
<table>
<thead>
<tr>
<th>Offset (Added to SMBASE + 8000H)</th>
<th>Register</th>
<th>Writable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FF8H</td>
<td>CR0</td>
<td>No</td>
</tr>
<tr>
<td>7FF0H</td>
<td>CR3</td>
<td>No</td>
</tr>
<tr>
<td>7FE8H</td>
<td>RFLAGS</td>
<td>Yes</td>
</tr>
<tr>
<td>7FE0H</td>
<td>IA32_EFER</td>
<td>Yes</td>
</tr>
<tr>
<td>7FD8H</td>
<td>RIP</td>
<td>Yes</td>
</tr>
<tr>
<td>7FD0H</td>
<td>DR6</td>
<td>No</td>
</tr>
<tr>
<td>7FC8H</td>
<td>DR7</td>
<td>No</td>
</tr>
<tr>
<td>7FC4H</td>
<td>TR SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FC0H</td>
<td>LDTR SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FBCH</td>
<td>GS SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FB8H</td>
<td>FS SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FB4H</td>
<td>DS SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FB0H</td>
<td>SS SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FA8H</td>
<td>ES SEL</td>
<td>No</td>
</tr>
<tr>
<td>7FA4H</td>
<td>IO_MISC</td>
<td>No</td>
</tr>
<tr>
<td>7F9CH</td>
<td>IO_MEM_ADDR</td>
<td>No</td>
</tr>
<tr>
<td>7F94H</td>
<td>RDI</td>
<td>Yes</td>
</tr>
<tr>
<td>7F8CH</td>
<td>RSI</td>
<td>Yes</td>
</tr>
<tr>
<td>7F84H</td>
<td>RBP</td>
<td>Yes</td>
</tr>
<tr>
<td>7F7CH</td>
<td>RSP</td>
<td>Yes</td>
</tr>
<tr>
<td>7F74H</td>
<td>RBX</td>
<td>Yes</td>
</tr>
<tr>
<td>7F6CH</td>
<td>RDX</td>
<td>Yes</td>
</tr>
<tr>
<td>7F64H</td>
<td>RCX</td>
<td>Yes</td>
</tr>
<tr>
<td>7F5CH</td>
<td>RAX</td>
<td>Yes</td>
</tr>
<tr>
<td>7F54H</td>
<td>R8</td>
<td>Yes</td>
</tr>
<tr>
<td>7F4CH</td>
<td>R9</td>
<td>Yes</td>
</tr>
<tr>
<td>7F44H</td>
<td>R10</td>
<td>Yes</td>
</tr>
<tr>
<td>7F3CH</td>
<td>R11</td>
<td>Yes</td>
</tr>
<tr>
<td>7F34H</td>
<td>R12</td>
<td>Yes</td>
</tr>
<tr>
<td>7F2CH</td>
<td>R13</td>
<td>Yes</td>
</tr>
<tr>
<td>7F24H</td>
<td>R14</td>
<td>Yes</td>
</tr>
<tr>
<td>7F1CH</td>
<td>R15</td>
<td>Yes</td>
</tr>
<tr>
<td>7F1BH-7F04H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7F02H</td>
<td>Auto HALT Restart Field (Word)</td>
<td>Yes</td>
</tr>
<tr>
<td>7F00H</td>
<td>I/O Instruction Restart Field (Word)</td>
<td>Yes</td>
</tr>
<tr>
<td>7F0CH</td>
<td>SMM Revision Identifier Field (Doubleword)</td>
<td>No</td>
</tr>
<tr>
<td>7FE8H</td>
<td>SMBASE Field (Doubleword)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
32.4.2 SMRAM Caching

An IA-32 processor does not automatically write back and invalidate its caches before entering SMM or before exiting SMM. Because of this behavior, care must be taken in the placement of the SMRAM in system memory and in the caching of the SMRAM to prevent cache incoherence when switching back and forth between SMM and protected mode operation. Any of the following three methods of locating the SMRAM in system memory will guarantee cache coherency.

- Place the SMRAM in a dedicated section of system memory that the operating system and applications are prevented from accessing. Here, the SMRAM can be designated as cacheable (WB, WT, or WC) for optimum processor performance, without risking cache incoherence when entering or exiting SMM.
- Place the SMRAM in a section of memory that overlaps an area used by the operating system (such as the video memory), but designate the SMRAM as uncacheable (UC). This method prevents cache access when in SMM to maintain cache coherency, but the use of uncacheable memory reduces the performance of SMM code.
- Place the SMRAM in a section of system memory that overlaps an area used by the operating system and/or application code, but explicitly flush (write back and invalidate) the caches upon entering and exiting SMM mode. This method maintains cache coherency, but incurs the overhead of two complete cache flushes.

For Pentium 4, Intel Xeon, and P6 family processors, a combination of the first two methods of locating the SMRAM is recommended. Here the SMRAM is split between an overlapping and a dedicated region of memory. Upon entering SMM, the SMRAM space that is accessed overlaps video memory (typically located in low memory). This SMRAM section is designated as UC memory. The initial SMM code then jumps to a second SMRAM section that is

### Table 32-3. SMRAM State Save Map for Intel 64 Architecture (Contd.)

<table>
<thead>
<tr>
<th>Offset (Added to SMBASE + 8000H)</th>
<th>Register</th>
<th>Writable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7EF7H - 7EE4H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7EE0H</td>
<td>Setting of “enable EPT” VM-execution control</td>
<td>No</td>
</tr>
<tr>
<td>7ED8H</td>
<td>Value of EPTP VM-execution control field</td>
<td>No</td>
</tr>
<tr>
<td>7ED7H - 7ECC0H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7EC8H</td>
<td>SSP</td>
<td>Yes</td>
</tr>
<tr>
<td>7EC7H - 7EA0H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7E9CH</td>
<td>LDT Base (lower 32 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7E98H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7E94H</td>
<td>IDT Base (lower 32 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7E90H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7E8CH</td>
<td>GDT Base (lower 32 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7E88H - 7E48H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7E40H</td>
<td>CR4 (64 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7E3FH - 7DF0H</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7DE8H</td>
<td>IO_RIP</td>
<td>Yes</td>
</tr>
<tr>
<td>7DE7H - 7DDCH</td>
<td>Reserved</td>
<td>No</td>
</tr>
<tr>
<td>7DD8H</td>
<td>IDT Base (Upper 32 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7DD4H</td>
<td>LDT Base (Upper 32 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7DD0H</td>
<td>GDT Base (Upper 32 bits)</td>
<td>No</td>
</tr>
<tr>
<td>7DCFH - 7C00H</td>
<td>Reserved</td>
<td>No</td>
</tr>
</tbody>
</table>

**NOTE:**
1. The two most significant bytes are reserved.
located in a dedicated region of system memory (typically in high memory). This SMRAM section can be cached for optimum processor performance.

For systems that explicitly flush the caches upon entering SMM (the third method described above), the cache flush can be accomplished by asserting the FLUSH# pin at the same time as the request to enter SMM (generally initiated by asserting the SMI# pin). The priorities of the FLUSH# and SMI# pins are such that the FLUSH# is serviced first. To guarantee this behavior, the processor requires that the following constraints on the interaction of FLUSH# and SMI# be met. In a system where the FLUSH# and SMI# pins are synchronous and the set up and hold times are met, then the FLUSH# and SMI# pins may be asserted in the same clock. In asynchronous systems, the FLUSH# pin must be asserted at least one clock before the SMI# pin to guarantee that the FLUSH# pin is serviced first.

Upon leaving SMM (for systems that explicitly flush the caches), the WBINVD instruction should be executed prior to leaving SMM to flush the caches.

**NOTES**

In systems based on the Pentium processor that use the FLUSH# pin to write back and invalidate cache contents before entering SMM, the processor will prefetch at least one cache line in between when the Flush Acknowledge cycle is run and the subsequent recognition of SMI# and the assertion of SMIACT#.

It is the obligation of the system to ensure that these lines are not cached by returning KEN# inactive to the Pentium processor.

### 32.4.2.1 System Management Range Registers (SMRR)

SMI handler code and data stored by SMM code resides in SMRAM. The SMRR interface is an enhancement in Intel 64 architecture to limit cacheable reference of addresses in SMRAM to code running in SMM. The SMRR interface can be configured only by code running in SMM. Details of SMRR is described in Section 12.11.2.4.

### 32.5 SMI HANDLER EXECUTION ENVIRONMENT

Section 32.5.1 describes the initial execution environment for an SMI handler. An SMI handler may re-configure its execution environment to other supported operating modes. Section 32.5.2 discusses modifications an SMI handler can make to its execution environment. Section 32.5.3 discusses Control-flow Enforcement Technology (CET) interactions in the environment.

### 32.5.1 Initial SMM Execution Environment

After saving the current context of the processor, the processor initializes its core registers to the values shown in Table 32-4. Upon entering SMM, the PE and PG flags in control register CR0 are cleared, which places the processor in an environment similar to real-address mode. The differences between the SMM execution environment and the real-address mode execution environment are as follows:

- The addressable address space ranges from 0 to FFFFFFFFH (4 GBytes).
- The normal 64-KByte segment limit for real-address mode is increased to 4 GBytes.
- The default operand and address sizes are set to 16 bits, which restricts the addressable SMRAM address space to the 1-MByte real-address mode limit for native real-address-mode code. However, operand-size and address-size override prefixes can be used to access the address space beyond the 1-MByte.

<table>
<thead>
<tr>
<th>Register</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>General-purpose registers</td>
<td>Undefined</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>000000002H</td>
</tr>
<tr>
<td>EIP</td>
<td>00008000H</td>
</tr>
<tr>
<td>CS selector</td>
<td>SMM Base shifted right 4 bits (default 3000H)</td>
</tr>
</tbody>
</table>
SYSTEM MANAGEMENT MODE

32.5.2 SMI Handler Operating Mode Switching

Within SMM, an SMI handler may change the processor’s operating mode (e.g., to enable PAE paging, enter 64-bit mode, etc.) after it has made proper preparation and initialization to do so. For example, if switching to 32-bit protected mode, the SMI handler should follow the guidelines provided in Chapter 10, “Processor Management and Initialization.” If the SMI handler does wish to change operating mode, it is responsible for executing the appropriate mode-transition code after each SMI.

It is recommended that the SMI handler make use of all means available to protect the integrity of its critical code and data. In particular, it should use the system-management range register (SMRR) interface if it is available (see Section 11.11.2.4). The SMRR interface can protect only the first 4 GBytes of the physical address space. The SMI handler should take that fact into account if it uses operating modes that allow access to physical addresses beyond that 4-GByte limit (e.g., PAE paging or 64-bit mode).

Execution of the RSM instruction restores the pre-SMI processor state from the SMRAM state-state map (see Section 32.4.1) into which it was stored when the processor entered SMM. (The SMBASE field in the SMRAM state-save map does not determine the state following RSM but rather the initial environment following the next entry to SMM.) Any required change to operating mode is performed by the RSM instruction; there is no need for the SMI handler to change modes explicitly prior to executing RSM.

Table 32-4. Processor Register Initialization in SMM

<table>
<thead>
<tr>
<th>Segment</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS base</td>
<td>SMM Base (default 30000H)</td>
</tr>
<tr>
<td>DS, ES, FS, GS, SS Selectors</td>
<td>0000H</td>
</tr>
<tr>
<td>DS, ES, FS, GS, SS Bases</td>
<td>000000000H</td>
</tr>
<tr>
<td>DS, ES, FS, GS, SS Limits</td>
<td>0FFFFFFFFH</td>
</tr>
<tr>
<td>CR0</td>
<td>PE, EM, TS, and PG flags set to 0; others unmodified</td>
</tr>
<tr>
<td>CR4</td>
<td>Cleared to zero</td>
</tr>
<tr>
<td>DR6</td>
<td>Undefined</td>
</tr>
<tr>
<td>DR7</td>
<td>00000400H</td>
</tr>
</tbody>
</table>

- Near jumps and calls can be made to anywhere in the 4-GByte address space if a 32-bit operand-size override prefix is used. Due to the real-address-mode style of base-address formation, a far call or jump cannot transfer control to a segment with a base address of more than 20 bits (1 MByte). However, since the segment limit in SMM is 4 GBytes, offsets into a segment that go beyond the 1-MByte limit are allowed when using 32-bit operand-size override prefixes. Any program control transfer that does not have a 32-bit operand-size override prefix truncates the EIP value to the 16 low-order bits.

- Data and the stack can be located anywhere in the 4-GByte address space, but can be accessed only with a 32-bit address-size override if they are located above 1 MByte. As with the code segment, the base address for a data or stack segment cannot be more than 20 bits.

The value in segment register CS is automatically set to the default of 30000H for the SMBASE shifted 4 bits to the right; that is, 3000H. The EIP register is set to 8000H. When the EIP value is added to shifted CS value (the SMBASE), the resulting linear address points to the first instruction of the SMI handler.

The other segment registers (DS, SS, ES, FS, and GS) are cleared to 0 and their segment limits are set to 4 GBytes. In this state, the SMRAM address space may be treated as a single flat 4-GByte linear address space. If a segment register is loaded with a 16-bit value, that value is then shifted left by 4 bits and loaded into the segment base (hidden part of the segment register). The limits and attributes are not modified.

Maskable hardware interrupts, exceptions, NMI interrupts, SMI interrupts, A20M interrupts, single-step traps, breakpoint traps, and INIT operations are inhibited when the processor enters SMM. Maskable hardware interrupts, exceptions, single-step traps, and breakpoint traps can be enabled in SMM if the SMM execution environment provides and initializes an interrupt table and the necessary interrupt and exception handlers (see Section 32.6).
32.5.3 Control-flow Enforcement Technology Interactions

On processors that support CET shadow stacks, when the processor enters SMM, the processor saves the SSP register to the SMRAM state save area (see Table 32-3) and clears CR4.CET to 0. Thus, the initial execution environment of the SMI handler has CET disabled and all of the CET state of the interrupted program is still in the machine. An SMM that uses CET is required to save the interrupted program’s CET state and restore the CET state prior to exiting SMM.

32.6 EXCEPTIONS AND INTERRUPTS WITHIN SMM

When the processor enters SMM, all hardware interrupts are disabled in the following manner:

- The IF flag in the EFLAGS register is cleared, which inhibits maskable hardware interrupts from being generated.
- The TF flag in the EFLAGS register is cleared, which disables single-step traps.
- Debug register DR7 is cleared, which disables breakpoint traps. (This action prevents a debugger from accidentally breaking into an SMI handler if a debug breakpoint is set in normal address space that overlays code or data in SMRAM.)
- NMI, SMI, and A20M interrupts are blocked by internal SMM logic. (See Section 32.8 for more information about how NMIs are handled in SMM.)

Software-invoked interrupts and exceptions can still occur, and maskable hardware interrupts can be enabled by setting the IF flag. Intel recommends that SMM code be written in so that it does not invoke software interrupts (with the INT n, INTO, INT1, INT3, or BOUND instructions) or generate exceptions.

If the SMI handler requires interrupt and exception handling, an SMM interrupt table and the necessary exception and interrupt handlers must be created and initialized from within SMM. Until the interrupt table is correctly initialized (using the LIDT instruction), exceptions and software interrupts will result in unpredictable processor behavior.

The following restrictions apply when designing SMM interrupt and exception-handling facilities:

- The interrupt table should be located at linear address 0 and must contain real-address mode style interrupt vectors (4 bytes containing CS and IP).
- Due to the real-address mode style of base address formation, an interrupt or exception cannot transfer control to a segment with a base address of more than 20 bits.
- An interrupt or exception cannot transfer control to a segment offset of more than 16 bits (64 KBytes).
- When an exception or interrupt occurs, only the 16 least-significant bits of the return address (EIP) are pushed onto the stack. If the offset of the interrupted procedure is greater than 64 KBytes, it is not possible for the interrupt/exception handler to return control to that procedure. (One solution to this problem is for a handler to adjust the return address on the stack.)
- The SMBASE relocation feature affects the way the processor will return from an interrupt or exception generated while the SMI handler is executing. For example, if the SMBASE is relocated to above 1 MByte, but the exception handlers are below 1 MByte, a normal return to the SMI handler is not possible. One solution is to provide the exception handler with a mechanism for calculating a return address above 1 MByte from the 16-bit return address on the stack, then use a 32-bit far call to return to the interrupted procedure.
- If an SMI handler needs access to the debug trap facilities, it must ensure that an SMM accessible debug handler is available and save the current contents of debug registers DR0 through DR3 (for later restoration). Debug registers DR0 through DR3 and DR7 must then be initialized with the appropriate values.
- If an SMI handler needs access to the single-step mechanism, it must ensure that an SMM accessible single-step handler is available, and then set the TF flag in the EFLAGS register.
- If the SMI design requires the processor to respond to maskable hardware interrupts or software-generated interrupts while in SMM, it must ensure that SMM accessible interrupt handlers are available and then set the IF flag in the EFLAGS register (using the STI instruction). Software interrupts are not blocked upon entry to SMM, so they do not need to be enabled.
32.7 MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS

When coding for a multiprocessor system or a system with Intel HT Technology, it was not always possible for an SMI handler to distinguish between a synchronous SMI (triggered during an I/O instruction) and an asynchronous SMI. To facilitate the discrimination of these two events, incremental state information has been added to the SMM state save map.

Processors that have an SMM revision ID of 30004H or higher have the incremental state information described below.

### 32.7.1 I/O State Implementation

Within the extended SMM state save map, a bit (IO_SMI) is provided that is set only when an SMI is either taken immediately after a successful I/O instruction or is taken after a successful iteration of a REP I/O instruction (the successful notion pertains to the processor point of view; not necessarily to the corresponding platform function). When set, the IO_SMI bit provides a strong indication that the corresponding SMI was synchronous. In this case, the SMM State Save Map also supplies the port address of the I/O operation. The IO_SMI bit and the I/O Port Address may be used in conjunction with the information logged by the platform to confirm that the SMI was indeed synchronous.

The IO_SMI bit by itself is a strong indication, not a guarantee, that the SMI is synchronous. This is because an asynchronous SMI might coincidentally be taken after an I/O instruction. In such a case, the IO_SMI bit would still be set in the SMM state save map.

Information characterizing the I/O instruction is saved in two locations in the SMM State Save Map (Table 32-5). The IO_SMI bit also serves as a valid bit for the rest of the I/O information fields. The contents of these I/O information fields are not defined when the IO_SMI bit is not set.

#### Table 32-5. I/O Instruction Information in the SMM State Save Map

<table>
<thead>
<tr>
<th>State (SMM Rev. ID: 30004H or higher)</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31 16</td>
</tr>
<tr>
<td>I/O State Field</td>
<td></td>
</tr>
<tr>
<td>SMRAM offset 7FA4</td>
<td></td>
</tr>
</tbody>
</table>

| I/O Memory Address Field             |        |       | I/O Memory Address |
| SMRAM offset 7FA0                    |        |       |

When IO_SMI is set, the other fields may be interpreted as follows:

- **I/O length:**
  - 001 – Byte
  - 010 – Word
  - 100 – Dword
- **I/O instruction type (Table 32-6)**

#### Table 32-6. I/O Instruction Type Encodings

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN Immediate</td>
<td>1001</td>
</tr>
<tr>
<td>IN DX</td>
<td>0001</td>
</tr>
<tr>
<td>OUT Immediate</td>
<td>1000</td>
</tr>
</tbody>
</table>
32.8 NMI HANDLING WHILE IN SMM

NMI interrupts are blocked upon entry to the SMI handler. If an NMI request occurs during the SMI handler, it is latched and serviced after the processor exits SMM. Only one NMI request will be latched during the SMI handler. If an NMI request is pending when the processor executes the RSM instruction, the NMI is serviced before the next instruction of the interrupted code sequence. This assumes that NMIs were not blocked before the SMI occurred. If NMIs were blocked before the SMI occurred, they are blocked after execution of RSM.

Although NMI requests are blocked when the processor enters SMM, they may be enabled through software by executing an IRET instruction. If the SMI handler requires the use of NMI interrupts, it should invoke a dummy interrupt service routine for the purpose of executing an IRET instruction. Once an IRET instruction is executed, NMI interrupt requests are serviced in the same “real mode” manner in which they are handled outside of SMM.

Also, for the Pentium processor, exceptions that invoke a trap or fault handler will enable NMI interrupts from inside of SMM. This behavior is implementation specific for the Pentium processor and is not part of the IA-32 architecture.

32.9 SMM REVISION IDENTIFIER

The SMM revision identifier field is used to indicate the version of SMM and the SMM extensions that are supported by the processor (see Figure 32-2). The SMM revision identifier is written during SMM entry and can be examined in SMRAM space at offset 7EFCH. The lower word of the SMM revision identifier refers to the version of the base SMM architecture.

The upper word of the SMM revision identifier refers to the extensions available. If the I/O instruction restart flag (bit 16) is set, the processor supports the I/O instruction restart (see Section 32.12); if the SMBASE relocation flag (bit 17) is set, SMRAM base address relocation is supported (see Section 32.11).
32.10 AUTO HALT RESTART

If the processor is in a HALT state (due to the prior execution of a HLT instruction) when it receives an SMI, the processor records the fact in the auto HALT restart flag in the saved processor state (see Figure 32-3). (This flag is located at offset 7F02H and bit 0 in the state save area of the SMRAM.)

If the processor sets the auto HALT restart flag upon entering SMM (indicating that the SMI occurred when the processor was in the HALT state), the SMI handler has two options:

- It can leave the auto HALT restart flag set, which instructs the RSM instruction to return program control to the HLT instruction. This option in effect causes the processor to re-enter the HALT state after handling the SMI. (This is the default operation.)
- It can clear the auto HALT restart flag, which instructs the RSM instruction to return program control to the instruction following the HLT instruction.

These options are summarized in Table 32-7. If the processor was not in a HALT state when the SMI was received (the auto HALT restart flag is cleared), setting the flag to 1 will cause unpredictable behavior when the RSM instruction is executed.

<table>
<thead>
<tr>
<th>Value of Flag After Entry to SMM</th>
<th>Value of Flag When Exiting SMM</th>
<th>Action of Processor When Exiting SMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Returns to next instruction in interrupted program or task.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Unpredictable.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Returns to next instruction after HLT instruction.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Returns to HALT state.</td>
</tr>
</tbody>
</table>

If the HLT instruction is restarted, the processor will generate a memory access to fetch the HLT instruction (if it is not in the internal cache), and execute a HLT bus transaction. This behavior results in multiple HLT bus transactions for the same HLT instruction.

32.10.1 Executing the HLT Instruction in SMM

The HLT instruction should not be executed during SMM, unless interrupts have been enabled by setting the IF flag in the EFLAGS register. If the processor is halted in SMM, the only event that can remove the processor from this state is a maskable hardware interrupt or a hardware reset.

32.11 SMBASE RELOCATION

The default base address for the SMRAM is 30000H. This value is contained in an internal processor register called the SMBASE register. Software can relocate the SMRAM by setting the SMBASE field in the saved state map (at offset 7EF8H) to a new value (see Figure 32-4). The RSM instruction reloads the internal SMBASE register with the value in the SMBASE field each time it exits SMM. All subsequent SMI requests will use the new SMBASE value to find the starting address for the SMI handler (at SMBASE + 8000H) and the SMRAM state save area (from SMBASE...
+ FE00H to SMBASE + FFFFH). (The processor resets the value in its internal SMBASE register to 30000H on a
RESET, but does not change it on an INIT.)

In multiple-processor systems, initialization software must adjust the SMBASE value for each processor so that the
SMRAM state save areas for each processor do not overlap. (For Pentium and Intel486 processors, the SMBASE
values must be aligned on a 32-KByte boundary or the processor will enter shutdown state during the execution of
a RSM instruction.)

If the SMBASE relocation flag in the SMM revision identifier field is set, it indicates the ability to relocate the
SMBASE (see Section 32.9).

### 32.12 I/O INSTRUCTION RESTART

If the I/O instruction restart flag in the SMM revision identifier field is set (see Section 32.9), the I/O instruction
restart mechanism is present on the processor. This mechanism allows an interrupted I/O instruction to be re-
executed upon returning from SMM mode. For example, if an I/O instruction is used to access a powered-down I/O
device, a chipset supporting this device can intercept the access and respond by asserting SMI#. This action
invokes the SMI handler to power-up the device. Upon returning from the SMI handler, the I/O instruction restart
mechanism can be used to re-execute the I/O instruction that caused the SMI.

The I/O instruction restart field (at offset 7F00H in the SMM state-save area, see Figure 32-5) controls I/O instruc-
tion restart. When an RSM instruction is executed, if this field contains the value FFH, then the EIP register is modi-
fied to point to the I/O instruction that received the SMI request. The processor will then automatically re-execute
the I/O instruction that the SMI trapped. (The processor saves the necessary machine state to ensure that re-
exection of the instruction is handled coherently.)

If the I/O instruction restart field contains the value 00H when the RSM instruction is executed, then the processor
begins program execution with the instruction following the I/O instruction. (When a repeat prefix is being used,
the next instruction may be the next I/O instruction in the repeat loop.) Not re-executing the interrupted I/O
instruction is the default behavior; the processor automatically initializes the I/O instruction restart field to 00H
upon entering SMM. Table 32-8 summarizes the states of the I/O instruction restart field.
The I/O instruction restart mechanism does not indicate the cause of the SMI. It is the responsibility of the SMI handler to examine the state of the processor to determine the cause of the SMI and to determine if an I/O instruction was interrupted and should be restarted upon exiting SMM. If an SMI interrupt is signaled on a non-I/O instruction boundary, setting the I/O instruction restart field to FFH prior to executing the RSM instruction will likely result in a program error.

### 32.12.1 Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used

If an SMI interrupt is signaled while the processor is servicing an SMI interrupt that occurred on an I/O instruction boundary, the processor will service the new SMI request before restarting the originally interrupted I/O instruction. If the I/O instruction restart field is set to FFH prior to returning from the second SMI handler, the EIP will point to an address different from the originally interrupted I/O instruction, which will likely lead to a program error. To avoid this situation, the SMI handler must be able to recognize the occurrence of back-to-back SMI interrupts when I/O instruction restart is being used and ensure that the handler sets the I/O instruction restart field to 00H prior to returning from the second invocation of the SMI handler.

### 32.13 SMM Multiple-Processor Considerations

The following should be noted when designing multiple-processor systems:

- Any processor in a multiprocessor system can respond to an SMI.
- Each processor needs its own SMRAM space. This space can be in system memory or in a separate RAM.
- The SMRAMs for different processors can be overlapped in the same memory space. The only stipulation is that each processor needs its own state save area and its own dynamic data storage area. (Also, for the Pentium and Intel486 processors, the SMBASE address must be located on a 32-KByte boundary.) Code and static data can be shared among processors. Overlapping SMRAM spaces can be done more efficiently with the P6 family processors because they do not require that the SMBASE address be on a 32-KByte boundary.
- The SMI handler will need to initialize the SMBASE for each processor.
- Processors can respond to local SMIs through their SMI# pins or to SMIs received through the APIC interface. The APIC interface can distribute SMIs to different processors.
- Two or more processors can be executing in SMM at the same time.
- When operating Pentium processors in dual processing (DP) mode, the SMIACT# pin is driven only by the MRM processor and should be sampled with ADS#. For additional details, see Chapter 14 of the Pentium Processor Family User’s Manual, Volume 1.

SMM is not re-entrant, because the SMRAM State Save Map is fixed relative to the SMBASE. If there is a need to support two or more processors in SMM mode at the same time then each processor should have dedicated SMRAM spaces. This can be done by using the SMBASE Relocation feature (see Section 32.11).

### 32.14 Default Treatment of SMIs and SMM with VMX Operation and SMX Operation

Under the default treatment, the interactions of SMIs and SMM with VMX operation are few. This section details those interactions. It also explains how this treatment affects SMX operation.
32.14.1 Default Treatment of SMI Delivery

Ordinary SMI delivery saves processor state into SMRAM and then loads state based on architectural definitions. Under the default treatment, processors that support VMX operation perform SMI delivery as follows:

enter SMM;
save the following internal to the processor:
  CR4.VMXE
  an indication of whether the logical processor was in VMX operation (root or non-root)
IF the logical processor is in VMX operation
  THEN
    save current VMCS pointer internal to the processor;
    leave VMX operation;
    save VMX-critical state defined below;
  FI;
IF the logical processor supports SMX operation
  THEN
    save internal to the logical processor an indication of whether the Intel® TXT private space is locked;
    IF the TXT private space is unlocked
      THEN lock the TXT private space;
    FI;
  FI;
CR4.VMXE := 0;
perform ordinary SMI delivery:
  save processor state in SMRAM;
  set processor state to standard SMM values;\(^1\)
  invalidate linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP; see Section 29.4);

The pseudocode above makes reference to the saving of **VMX-critical state**. This state consists of the following: (1) SS.DPL (the current privilege level); (2) RFLAGS.VM\(^2\); (3) the state of blocking by STI and by MOV SS (see Table 25-3 in Section 25.4.2); (4) the state of virtual-NMI blocking (only if the processor is in VMX non-root operation and the "virtual NMIs" VM-execution control is 1); and (5) an indication of whether an MTF VM exit is pending (see Section 26.5.2). These data may be saved internal to the processor or in the VMCS region of the current VMCS. Processors that do not support SMI recognition while there is blocking by STI or by MOV SS need not save the state of such blocking.

If the logical processor supports the 1-setting of the "enable EPT" VM-execution control and the logical processor was in VMX non-root operation at the time of an SMI, it saves the value of that control into bit 0 of the 32-bit field at offset SMBASE + 8000H + 7EE0H (SMBASE + FEE0H; see Table 32-3).\(^3\) If the logical processor was not in VMX non-root operation at the time of the SMI, it saves 0 into that bit. If the logical processor saves 1 into that bit (it was in VMX non-root operation and the "enable EPT" VM-execution control was 1), it saves the value of the EPT pointer (EPTP) into the 64-bit field at offset SMBASE + 8000H + 7ED8H (SMBASE + FED8H).

Because SMI delivery causes a logical processor to leave VMX operation, all the controls associated with VMX non-root operation are disabled in SMM and thus cannot cause VM exits while the logical processor in SMM.

---

1. This causes the logical processor to block INIT signals, NMIs, and SMIs.
2. Section 32.14 and Section 32.15 use the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of these registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to the lower 32 bits of the register.
3. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, SMI functions as the "enable EPT" VM-execution control were 0. See Section 25.6.2.
32.14.2 Default Treatment of RSM

Ordinary execution of RSM restores processor state from SMRAM. Under the default treatment, processors that support VMX operation perform RSM as follows:

IF VMXE = 1 in CR4 image in SMRAM
    THEN fail and enter shutdown state;
ELSE
    restore state normally from SMRAM;
    invalidate linear mappings and combined mappings associated with all VPIDs and all PCIDs; combined mappings are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP; see Section 29.4);
    IF the logical processor supports SMX operation and the Intel® TXT private space was unlocked at the time of the last SMI (as saved)
    THEN unlock the TXT private space;
    CR4.VMXE := value stored internally;
    IF internal storage indicates that the logical processor had been in VMX operation (root or non-root)
    THEN
        enter VMX operation (root or non-root);
        restore VMX-critical state as defined in Section 32.14.1;
        set to their fixed values any bits in CR0 and CR4 whose values must be fixed in VMX operation (see Section 24.8);\textsuperscript{1}
        IF RFLAGS.VM = 0 AND (in VMX root operation OR the "unrestricted guest" VM-execution control is 0);\textsuperscript{2}
            THEN
                CS.RPL := SS.DPL;
                SS.RPL := SS.DPL;
            FI;
        restore current VMCS pointer;
    FI;
    leave SMM;
    IF logical processor will be in VMX operation or in SMX operation after RSM
    THEN block A20M and leave A20M mode;
    FI;
FI;

RSM unblocks SMI. It restores the state of blocking by NMI (see Table 25-3 in Section 25.4.2) as follows:

- If the RSM is not to VMX non-root operation or if the "virtual NMIs" VM-execution control will be 0, the state of NMI blocking is restored normally.
- If the RSM is to VMX non-root operation and the "virtual NMIs" VM-execution control will be 1, NMIs are not blocked after RSM. The state of virtual-NMI blocking is restored as part of VMX-critical state.

INIT signals are blocked after RSM if and only if the logical processor will be in VMX root operation.

If RSM returns a logical processor to VMX non-root operation, it re-establishes the controls associated with the current VMCS. If the “interrupt-window exiting” VM-execution control is 1, a VM exit occurs immediately after RSM if the enabling conditions apply. The same is true for the "NMI-window exiting" VM-execution control. Such VM exits occur with their normal priority. See Section 26.2.

If an MTF VM exit was pending at the time of the previous SMI, an MTF VM exit is pending on the instruction boundary following execution of RSM. The following items detail the treatment of MTF VM exits that may be pending following RSM:

\begin{itemize}
\item If the RSM is to VMX non-root operation and both the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls will be 1, CR0.PE and CR0.PG retain the values that were loaded from SMRAM regardless of what is reported in the capability MSR IA32_VMX_CR0_FIXED0.
\item "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "unrestricted guest" VM-execution control were 0. See Section 25.6.2.
\end{itemize}
- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over these MTF VM exits. These MTF VM exits take priority over debug-trap exceptions and lower priority events.
- These MTF VM exits wake the logical processor if RSM caused the logical processor to enter the HLT state (see Section 32.10). They do not occur if the logical processor just entered the shutdown state.

32.14.3 Protection of CR4.VMXE in SMM

Under the default treatment, CR4.VMXE is treated as a reserved bit while a logical processor is in SMM. Any attempt by software running in SMM to set this bit causes a general-protection exception. In addition, software cannot use VMX instructions or enter VMX operation while in SMM.

32.14.4 VMXOFF and SMI Unblocking

The VMXOFF instruction can be executed only with the default treatment (see Section 32.15.1) and only outside SMM. If SMIs are blocked when VMXOFF is executed, VMXOFF unblocks them unless IA32_SMM_MONITOR_CTL[bit 2] is 1 (see Section 32.15.5 for details regarding this MSR).1 Section 32.15.7 identifies a case in which SMIs may be blocked when VMXOFF is executed.

Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32_VMX_MISC (see Appendix A.6) to determine whether this is allowed.

32.15 DUAL-MONITOR TREATMENT OF SMIs AND SMM

Dual-monitor treatment is activated through the cooperation of the executive monitor (the VMM that operates outside of SMM to provide basic virtualization) and the SMM-transfer monitor (STM; the VMM that operates inside SMM—while in VMX operation—to support system-management functions). Control is transferred to the STM through VM exits; VM entries are used to return from SMM.

The dual-monitor treatment may not be supported by all processors. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1) to determine whether it is supported.

32.15.1 Dual-Monitor Treatment Overview

The dual-monitor treatment uses an executive monitor and an SMM-transfer monitor (STM). Transitions from the executive monitor or its guests to the STM are called SMM VM exits and are discussed in Section 32.15.2. SMM VM exits are caused by SMIs as well as executions of VMCALL in VMX root operation. The latter allow the executive monitor to call the STM for service.

The STM runs in VMX root operation and uses VMX instructions to establish a VMCS and perform VM entries to its own guests. This is done all inside SMM (see Section 32.15.3). The STM returns from SMM, not by using the RSM instruction, but by using a VM entry that returns from SMM. Such VM entries are described in Section 32.15.4.

Initially, there is no STM and the default treatment (Section 32.14) is used. The dual-monitor treatment is not used until it is enabled and activated. The steps to do this are described in Section 32.15.5 and Section 32.15.6.

It is not possible to leave VMX operation under the dual-monitor treatment; VMXOFF will fail if executed. The dual-monitor treatment must be deactivated first. The STM deactivates dual-monitor treatment using a VM entry that returns from SMM with the "deactivate dual-monitor treatment" VM-entry control set to 1 (see Section 32.15.7).

The executive monitor configures any VMCS that it uses for VM exits to the executive monitor. SMM VM exits, which transfer control to the STM, use a different VMCS. Under the dual-monitor treatment, each logical processor uses a separate VMCS called the SMM-transfer VMCS. When the dual-monitor treatment is active, the logical processor maintains another VMCS pointer called the SMM-transfer VMCS pointer. The SMM-transfer VMCS pointer is established when the dual-monitor treatment is activated.

---

1. Setting IA32_SMM_MONITOR_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register’s valid bit (bit 0).
32.15.2 SMM VM Exits

An SMM VM exit is a VM exit that begins outside SMM and that ends in SMM.

Unlike other VM exits, SMM VM exits can begin in VMX root operation. SMM VM exits result from the arrival of an SMI outside SMM or from execution of VMCALL in VMX root operation outside SMM. Execution of VMCALL in VMX root operation causes an SMM VM exit only if the valid bit is set in the IA32_SMM_MONITOR_CTL MSR (see Section 32.15.5).

Execution of VMCALL in VMX root operation causes an SMM VM exit even under the default treatment. This SMM VM exit activates the dual-monitor treatment (see Section 32.15.6).

Differences between SMM VM exits and other VM exits are detailed in Sections 32.15.2.1 through 32.15.2.5. Differences between SMM VM exits that activate the dual-monitor treatment and other SMM VM exits are described in Section 32.15.6.

32.15.2.1 Architectural State Before a VM Exit

System-management interrupts (SMIs) that cause SMM VM exits always do so directly. They do not save state to SMRAM as they do under the default treatment.

32.15.2.2 Updating the Current-VMCS and Executive-VMCS Pointers

SMM VM exits begin by performing the following steps:

1. The executive-VMCS pointer field in the SMM-transfer VMCS is loaded as follows:
   — If the SMM VM exit commenced in VMX non-root operation, it receives the current-VMCS pointer.
   — If the SMM VM exit commenced in VMX root operation, it receives the VMXON pointer.
2. The current-VMCS pointer is loaded with the value of the SMM-transfer VMCS pointer.

The last step ensures that the current VMCS is the SMM-transfer VMCS. VM-exit information is recorded in that VMCS, and VM-entry control fields in that VMCS are updated. State is saved into the guest-state area of that VMCS. The VM-exit controls and host-state area of that VMCS determine how the VM exit operates.

32.15.2.3 Recording VM-Exit Information

SMM VM exits differ from other VM exit with regard to the way they record VM-exit information. The differences follow.

- **Exit reason.**
  — Bits 15:0 of this field contain the basic exit reason. The field is loaded with the reason for the SMM VM exit: I/O SMI (an SMI arrived immediately after retirement of an I/O instruction), other SMI, or VMCALL. See Appendix C, “VMX Basic Exit Reasons.”
  — SMM VM exits are the only VM exits that may occur in VMX root operation. Because the SMM-transfer monitor may need to know whether it was invoked from VMX root or VMX non-root operation, this information is stored in bit 29 of the exit-reason field (see Table 25-18 in Section 25.9.1). The bit is set by SMM VM exits from VMX root operation.
  — If the SMM VM exit occurred in VMX non-root operation and an MTF VM exit was pending, bit 28 of the exit-reason field is set; otherwise, it is cleared.
  — Bits 27:16 and bits 31:30 are cleared.

- **Exit qualification.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, the exit qualification contains information about the I/O instruction that retired immediately before the SMI. It has the format given in Table 32-9.

- **Guest linear address.** This field is used for VM exits due to SMIs that arrive immediately after the retirement of an INS or OUTS instruction for which the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) is usable. The field receives the value of the linear address generated by ES:(E)DI (for INS) or segment:(E)SI (for OUTS; the default segment is DS but can be overridden by a segment override
**Table 32-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction**

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Size of access:</td>
</tr>
<tr>
<td></td>
<td>0 = 1-byte</td>
</tr>
<tr>
<td></td>
<td>1 = 2-byte</td>
</tr>
<tr>
<td></td>
<td>3 = 4-byte</td>
</tr>
<tr>
<td></td>
<td>Other values not used.</td>
</tr>
<tr>
<td>3</td>
<td>Direction of the attempted access (0 = OUT, 1 = IN)</td>
</tr>
<tr>
<td>4</td>
<td>String instruction (0 = not string; 1 = string)</td>
</tr>
<tr>
<td>5</td>
<td>REP prefixed (0 = not REP; 1 = REP)</td>
</tr>
<tr>
<td>6</td>
<td>Operand encoding (0 = DX, 1 = immediate)</td>
</tr>
<tr>
<td>15:7</td>
<td>Reserved (cleared to 0)</td>
</tr>
<tr>
<td>31:16</td>
<td>Port number (as specified in the I/O instruction)</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved (cleared to 0). These bits exist only on processors that</td>
</tr>
<tr>
<td></td>
<td>support Intel 64 architecture.</td>
</tr>
</tbody>
</table>

prefix) at the time the instruction started. If the relevant segment is not usable, the value is undefined. On processors that support Intel 64 architecture, bits 63:32 are clear if the logical processor was not in 64-bit mode before the VM exit.

**I/O RCX, I/O RSI, I/O RDI, and I/O RIP.** For an SMM VM exit due an SMI that arrives immediately after the retirement of an I/O instruction, these fields receive the values that were in RCX, RSI, RDI, and RIP, respectively, before the I/O instruction executed. Thus, the value saved for I/O RIP addresses the I/O instruction.

### 32.15.2.4 Saving Guest State

SMM VM exits save the contents of the SMBASE register into the corresponding field in the guest-state area.

The value of the VMX-preemption timer is saved into the corresponding field in the guest-state area if the “save VMX-preemption timer value” VM-exit control is 1. That field becomes undefined if, in addition, either the SMM VM exit is from VMX root operation or the SMM VM exit is from VMX non-root operation and the “activate VMX-preemption timer” VM-execution control is 0.

### 32.15.2.5 Updating State

If an SMM VM exit is from VMX non-root operation and the “Intel PT uses guest physical addresses” VM-execution control is 1, the IA32_RTIT_CTL MSR is cleared to 00000000_00000000H. This is done even if the “clear IA32_RTIT_CTL” VM-exit control is 0.

SMM VM exits affect the non-register state of a logical processor as follows:

- SMM VM exits cause non-maskable interrupts (NMIs) to be blocked; they may be unblocked through execution of IRET or through a VM entry (depending on the value loaded for the interruptibility state and the setting of the "virtual NMIs" VM-execution control).
- SMM VM exits cause SMIs to be blocked; they may be unblocked by a VM entry that returns from SMM (see Section 32.15.4).

1. In this situation, the value of this MSR was saved earlier into the guest-state area. All VM exits save this MSR if the 1-setting of the “load IA32_RTIT_CTL” VM-entry control is supported (see Section 28.3.1), which must be the case if the “Intel PT uses guest physical addresses” VM-execution control is 1 (see Section 27.2.1.1).
SYSTEM MANAGEMENT MODE

SMM VM exits invalidate linear mappings and combined mappings associated with VPID 0000H for all PCIDs. Combined mappings for VPID 0000H are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP; see Section 29.4). (Ordinary VM exits are not required to perform such invalidation if the “enable VPID” VM-execution control is 1; see Section 28.5.5.)

32.15.3 Operation of the SMM-Transfer Monitor

Once invoked, the SMM-transfer monitor (STM) is in VMX root operation and can use VMX instructions to configure VMCSs and to cause VM entries to virtual machines supported by those structures. As noted in Section 32.15.1, the VMXOFF instruction cannot be used under the dual-monitor treatment and thus cannot be used by the STM. The RSM instruction also cannot be used under the dual-monitor treatment. As noted in Section 26.1.3, it causes a VM exit if executed in SMM in VMX non-root operation. If executed in VMX root operation, it causes an invalid-opcode exception. The STM uses VM entries to return from SMM (see Section 32.15.4).

32.15.4 VM Entries that Return from SMM

The SMM-transfer monitor (STM) returns from SMM using a VM entry with the “entry to SMM” VM-entry control clear. VM entries that return from SMM reverse the effects of an SMM VM exit (see Section 32.15.2). VM entries that return from SMM may differ from other VM entries in that they do not necessarily enter VMX non-root operation. If the executive-VMCS pointer field in the current VMCS contains the VMXON pointer, the logical processor remains in VMX root operation after VM entry. For differences between VM entries that return from SMM and other VM entries see Sections 32.15.4.1 through 32.15.4.10.

32.15.4.1 Checks on the Executive-VMCS Pointer Field

VM entries that return from SMM perform the following checks on the executive-VMCS pointer field in the current VMCS:

- Bits 11:0 must be 0.
- The pointer must not set any bits beyond the processor’s physical-address width.1,2
- The 32 bits located in memory referenced by the physical address in the pointer must contain the processor’s VMCS revision identifier (see Section 25.2).

The checks above are performed before the checks described in Section 32.15.4.2 and before any of the following checks:

- If the “deactivate dual-monitor treatment” VM-entry control is 0 and the executive-VMCS pointer field does not contain the VMXON pointer, the launch state of the executive VMCS (the VMCS referenced by the executive-VMCS pointer field) must be launched (see Section 25.11.3).
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the executive-VMCS pointer field must contain the VMXON pointer (see Section 32.15.7).3

32.15.4.2 Checks on VM-Execution Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-execution control fields specified in Section 27.2.1.1. They do not apply the checks to the current VMCS. Instead, VM-entry behavior depends on whether the executive-VMCS pointer field contains the VMXON pointer:

1. Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. If IA32_VMX_BASIC[48] is read as 1, this pointer must not set any bits in the range 63:32; see Appendix A.1.
3. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.
• If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are not performed at all.

• If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the checks are performed on the VM-execution control fields in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS). These checks are performed after checking the executive-VMCS pointer field itself (for proper alignment).

Other VM entries ensure that, if "activate VMX-preemption timer" VM-execution control is 0, the "save VMX-preemption timer value" VM-exit control is also 0. This check is not performed by VM entries that return from SMM.

32.15.4.3 Checks on VM-Entry Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-entry control fields specified in Section 27.2.1.3.

Specifically, if the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the VM-entry interruption-information field must not indicate injection of a pending MTF VM exit (see Section 27.6.2). Specifically, the following cannot all be true for that field:

• the valid bit (bit 31) is 1
• the interruption type (bits 10:8) is 7 (other event); and
• the vector (bits 7:0) is 0 (pending MTF VM exit).

32.15.4.4 Checks on the Guest State Area

Section 27.3.1 specifies checks performed on fields in the guest-state area of the VMCS. Some of these checks are conditioned on the settings of certain VM-execution controls (e.g., "virtual NMIs" or "unrestricted guest").

VM entries that return from SMM modify these checks based on whether the executive-VMCS pointer field contains the VMXON pointer:

• If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are performed as all relevant VM-execution controls were 0. (As a result, some checks may not be performed at all.)

• If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), this check is performed based on the settings of the VM-execution controls in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the current VMCS).

For VM entries that return from SMM, the activity-state field must not indicate the wait-for-SIPI state if the executive-VMCS pointer field contains the VMXON pointer (the VM entry is to VMX root operation).

32.15.4.5 Loading Guest State

VM entries that return from SMM load the SMBASE register from the SMBASE field.

VM entries that return from SMM invalidate linear mappings and combined mappings associated with all VPID.s. Combined mappings are invalidated for all EPTRTA values (EPTRTA is the value of bits 51:12 of EPTP; see Section 29.4). (Ordinary VM entries are required to perform such invalidation only for VPID 0000H and are not required to do even that if the "enable VPID" VM-execution control is 1; see Section 27.3.2.5.)

32.15.4.6 VMX-Preemption Timer

A VM entry that returns from SMM activates the VMX-preemption timer only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation) and the "activate VMX-preemption timer" VM-execution control is 1 in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field). In this case, VM entry starts the VMX-preemption timer with the value in the VMX-preemption timer-value field in the current VMCS.

1. The STM can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.
32.15.4.7 Updating the Current-VMCS and SMM-Transfer VMCS Pointers

Successful VM entries (returning from SMM) load the SMM-transfer VMCS pointer with the current-VMCS pointer. Following this, they load the current-VMCS pointer from a field in the current VMCS:

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the current-VMCS pointer is loaded from the VMCS-link pointer field.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the current-VMCS pointer is loaded with the value of the executive-VMCS pointer field.

If the VM entry successfully enters VMX non-root operation, the VM-execution controls in effect after the VM entry are those from the new current VMCS. This includes any structures external to the VMCS referenced by VM-execution control fields.

The updating of these VMCS pointers occurs before event injection. Event injection is determined, however, by the VM-entry control fields in the VMCS that was current when the VM entry commenced.

32.15.4.8 VM Exits Induced by VM Entry

Section 27.6.1.2 describes how the event-delivery process invoked by event injection may lead to a VM exit. Section 27.7.3 to Section 27.7.7 describe other situations that may cause a VM exit to occur immediately after a VM entry.

Whether these VM exits occur is determined by the VM-execution control fields in the current VMCS. For VM entries that return from SMM, they can occur only if the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation).

In this case, determination is based on the VM-execution control fields in the VMCS that is current after the VM entry. This is the VMCS referenced by the value of the executive-VMCS pointer field at the time of the VM entry (see Section 32.15.4.7). This VMCS also controls the delivery of such VM exits. Thus, VM exits induced by a VM entry returning from SMM are to the executive monitor and not to the STM.

32.15.4.9 SMI Blocking

VM entries that return from SMM determine the blocking of system-management interrupts (SMIs) as follows:

- If the “deactivate dual-monitor treatment” VM-entry control is 0, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.
- If the “deactivate dual-monitor treatment” VM-entry control is 1, the blocking of SMIs depends on whether the logical processor is in SMX operation:
  - If the logical processor is in SMX operation, SMIs are blocked after VM entry.
  - If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

VM entries that return from SMM and that do not deactivate the dual-monitor treatment may leave SMIs blocked. This feature exists to allow the STM to invoke functionality outside of SMM without unblocking SMIs.

32.15.4.10 Failures of VM Entries That Return from SMM

Section 27.8 describes the treatment of VM entries that fail during or after loading guest state. Such failures record information in the VM-exit information fields and load processor state as would be done on a VM exit. The VMCS used is the one that was current before the VM entry commenced. Control is thus transferred to the STM and the logical processor remains in SMM.

---

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 7, “Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.
32.15.5 Enabling the Dual-Monitor Treatment

Code and data for the SMM-transfer monitor (STM) reside in a region of SMRAM called the **monitor segment** (MSEG). Code running in SMM determines the location of MSEG and establishes its content. This code is also responsible for enabling the dual-monitor treatment.

SMM code enables the dual-monitor treatment and specifies the location of MSEG by writing to the IA32_SMM_MONITOR_CTL MSR (index 9BH). The MSR has the following format:

- Bit 0 is the register’s valid bit. The STM may be invoked using VMCALL only if this bit is 1. Because VMCALL is used to activate the dual-monitor treatment (see Section 32.15.6), the dual-monitor treatment cannot be activated if the bit is 0. This bit is cleared when the logical processor is reset.
- Bit 1 is reserved.
- Bit 2 determines whether executions of VMXOFF unblock SMIs under the default treatment of SMIs and SMM. Executions of VMXOFF unblock SMIs unless bit 2 is 1 (the value of bit 0 is irrelevant). See Section 32.14.4. Certain leaf functions of the GETSEC instruction clear this bit (see Chapter 7, "Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D).
- Bits 11:3 are reserved.
- Bits 31:12 contain a value that, when shifted left 12 bits, is the physical address of MSEG (the MSEG base address).
- Bits 63:32 are reserved.

The following items detail use of this MSR:

- The IA32_SMM_MONITOR_CTL MSR is supported only on processors that support the dual-monitor treatment.\(^1\) On other processors, accesses to the MSR using RDMSR or WRMSR generate a general-protection fault (#GP(0)).
- A write to the IA32_SMM_MONITOR_CTL MSR using WRMSR generates a general-protection fault (#GP(0)) if executed outside of SMM or if an attempt is made to set any reserved bit. An attempt to write to the IA32_SMM_MONITOR_CTL MSR fails if made as part of a VM exit that does not end in SMM or part of a VM entry that does not begin in SMM.
- Reads from the IA32_SMM_MONITOR_CTL MSR using RDMSR are allowed any time RDMSR is allowed. The MSR may be read as part of any VM exit.
- The dual-monitor treatment can be activated only if the valid bit in the MSR is set to 1.

The 32 bytes located at the MSEG base address are called the **MSEG header**. The format of the MSEG header is given in Table 32-10 (each field is 32 bits).

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSEG-header revision identifier</td>
</tr>
<tr>
<td>4</td>
<td>SMM-transfer monitor features</td>
</tr>
<tr>
<td>8</td>
<td>GDTR limit</td>
</tr>
<tr>
<td>12</td>
<td>GDTR base offset</td>
</tr>
<tr>
<td>16</td>
<td>CS selector</td>
</tr>
<tr>
<td>20</td>
<td>EIP offset</td>
</tr>
<tr>
<td>24</td>
<td>ESP offset</td>
</tr>
<tr>
<td>28</td>
<td>CR3 offset</td>
</tr>
</tbody>
</table>

---

1. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.
To ensure proper behavior in VMX operation, software should maintain the MSEG header in writeback cacheable memory. Future implementations may allow or require a different memory type. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

SMM code should enable the dual-monitor treatment (by setting the valid bit in IA32_SMM_MONITOR_CTL MSR) only after establishing the content of the MSEG header as follows:

- Bytes 3:0 contain the **MSEG revision identifier**. Different processors may use different MSEG revision identifiers. These identifiers enable software to avoid using an MSEG header formatted for one processor on a processor that uses a different format. Software can discover the MSEG revision identifier that a processor uses by reading the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).
- Bytes 7:4 contain the **SMM-transfer monitor features** field. Bits 31:1 of this field are reserved and must be zero. Bit 0 of the field is the **IA-32e mode SMM feature bit**. It indicates whether the logical processor will be in IA-32e mode after the STM is activated (see Section 32.15.6).
- Bytes 31:8 contain fields that determine how processor state is loaded when the STM is activated (see Section 32.15.6.5). SMM code should establish these fields so that activating of the STM invokes the STM's initialization code.

### 32.15.6 Activating the Dual-Monitor Treatment

The dual-monitor treatment may be enabled by SMM code as described in Section 32.15.5. The dual-monitor treatment is activated only if it is enabled and only by the executive monitor. The executive monitor activates the dual-monitor treatment by executing VMCALL in VMX root operation.

When VMCALL activates the dual-monitor treatment, it causes an SMM VM exit. Differences between this SMM VM exit and other SMM VM exits are discussed in Sections 32.15.6.1 through 32.15.6.6. See also “VMCALL—Call to VM Monitor” in Chapter 31.

#### 32.15.6.1 Initial Checks

An execution of VMCALL attempts to activate the dual-monitor treatment if (1) the processor supports the dual-monitor treatment; (2) the logical processor is in VMX root operation; (3) the logical processor is outside SMM and the valid bit is set in the IA32_SMM_MONITOR_CTL MSR; (4) the logical processor is not in virtual-8086 mode and not in compatibility mode; (5) CPL = 0; and (6) the dual-monitor treatment is not active.

Such an execution of VMCALL begins with some initial checks. These checks are performed before updating the current-VMCS pointer and the executive-VMCS pointer field (see Section 32.15.2.2).

The VMCS that manages SMM VM exit caused by this VMCALL is the current VMCS established by the executive monitor. The VMCALL performs the following checks on the current VMCS in the order indicated:

1. There must be a current VMCS pointer.
2. The launch state of the current VMCS must be clear.
3. The VM-exit controls in the current VMCS must be set properly:
   - Reserved bits in the primary VM-exit controls must be set properly. Software may consult the VMX capability MSRs to determine the proper setting (see Appendix A.4.1).
   - If the “activate secondary controls” primary VM-exit control is 1, reserved bits in the secondary VM-exit controls must be cleared. Software may consult the VMX capability MSRs to determine which bits are reserved (see Appendix A.4.2).

---

1. Alternatively, software may map the MSEG header with the UC memory type; this may be necessary, depending on how memory is organized. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly. In addition, the processor will continue to use the memory type reported in the VMX capability MSR IA32_VMX_BASIC with exceptions noted in Appendix A.1.
2. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1) to determine whether the dual-monitor treatment is supported.
• If the "activate secondary controls" primary VM-exit control is 0 (or if the processor does not support
the 1-setting of that control), no checks are performed on the secondary VM-exit controls. The logical
processor operates as if all the secondary VM-exit controls were 0.

If any of these checks fail, subsequent checks are skipped and VMCALL fails. If all these checks succeed, the logical
processor uses the IA32_SMM_MONITOR_CTL MSR to determine the base address of MSEG. The following checks
are performed in the order indicated:
1. The logical processor reads the 32 bits at the base of MSEG and compares them to the processor’s MSEG
revision identifier.
2. The logical processor reads the SMM-transfer monitor features field:
   — Bit 0 of the field is the IA-32e mode SMM feature bit, and it indicates whether the logical processor will be
     in IA-32e mode after the SMM-transfer monitor (STM) is activated.
   • If the VMCALL is executed on a processor that does not support Intel 64 architecture, the IA-32e mode
     SMM feature bit must be 0.
   • If the VMCALL is executed in 64-bit mode, the IA-32e mode SMM feature bit must be 1.
     — Bits 31:1 of this field are currently reserved and must be zero.
If any of these checks fail, subsequent checks are skipped and the VMCALL fails.

32.15.6.2 Updating the Current-VMCS and Executive-VMCS Pointers
Before performing the steps in Section 32.15.2.2, SMM VM exits that activate the dual-monitor treatment begin by
loading the SMM-transfer VMCS pointer with the value of the current-VMCS pointer.

32.15.6.3 Saving Guest State
As noted in Section 32.15.2.4, SMM VM exits save the contents of the SMBASE register into the corresponding field
in the guest-state area. While this is true also for SMM VM exits that activate the dual-monitor treatment, the
VMCS used for those VM exits exists outside SMRAM.
The SMM-transfer monitor (STM) can also discover the current value of the SMBASE register by using the RDMSR
instruction to read the IA32_SMBASE MSR (MSR address 9EH). The following items detail use of this MSR:
• The MSR is supported only if IA32_VMX_MISC[15] = 1 (see Appendix A.6).
• A write to the IA32_SMBASE MSR using WRMSR generates a general-protection fault (#GP(0)). An attempt to
  write to the IA32_SMBASE MSR fails if made as part of a VM exit or part of a VM entry.
• A read from the IA32_SMBASE MSR using RDMSR generates a general-protection fault (#GP(0)) if executed
  outside of SMM. An attempt to read from the IA32_SMBASE MSR fails if made as part of a VM exit that does not
  end in SMM.

32.15.6.4 Saving MSRs
The VM-exit MSR-store area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are
saved into that area.

32.15.6.5 Loading Host State
The VMCS that is current during an SMM VM exit that activates the dual-monitor treatment was established by the
executive monitor. It does not contain the VM-exit controls and host state required to initialize the STM. For this
reason, such SMM VM exits do not load processor state as described in Section 28.5. Instead, state is set to fixed
values or loaded based on the content of the MSEG header (see Table 32-10):
• CR0 is set to as follows:
  — PG, NE, ET, MP, and PE are all set to 1.
  — CD and NW are left unchanged.
SYSTEM MANAGEMENT MODE

- All other bits are cleared to 0.

• CR3 is set as follows:
  - Bits 63:32 are cleared on processors that support IA-32e mode.
  - Bits 31:12 are set to bits 31:12 of the sum of the MSEG base address and the CR3-offset field in the MSEG header.
  - Bits 11:5 and bits 2:0 are cleared (the corresponding bits in the CR3-offset field in the MSEG header are ignored).
  - Bits 4:3 are set to bits 4:3 of the CR3-offset field in the MSEG header.

• CR4 is set as follows:
  - MCE, PGE, CET, PCIDE, and LA57 are cleared.
  - PAE is set to the value of the IA-32e mode SMM feature bit.
  - If the IA-32e mode SMM feature bit is clear, PSE is set to 1 if supported by the processor; if the bit is set, PSE is cleared.
  - All other bits are unchanged.

• DR7 is set to 400H.

• The IA32_DEBUGCTL MSR is cleared to 00000000_00000000H.

• The registers CS, SS, DS, ES, FS, and GS are loaded as follows:
  - All registers are usable.
  - CS.selector is loaded from the corresponding field in the MSEG header (the high 16 bits are ignored), with bits 2:0 cleared to 0. If the result is 0000H, CS.selector is set to 0008H.
  - The selectors for SS, DS, ES, FS, and GS are set to CS.selector+0008H. If the result is 0000H (if the CS selector was FFF8H), these selectors are instead set to 0008H.
  - The base addresses of all registers are cleared to zero.
  - The segment limits for all registers are set to FFFFFFFFH.
  - The AR bytes for the registers are set as follows:
    - CS.Type is set to 11 (execute/read, accessed, non-conforming code segment).
    - For SS, DS, ES, FS, and GS, the Type is set to 3 (read/write, accessed, expand-up data segment).
    - The S bits for all registers are set to 1.
    - The DPL for each register is set to 0.
    - The P bits for all registers are set to 1.
    - On processors that support Intel 64 architecture, CS.L is loaded with the value of the IA-32e mode SMM feature bit.
    - CS.D is loaded with the inverse of the value of the IA-32e mode SMM feature bit.
    - For each of SS, DS, ES, FS, and GS, the D/B bit is set to 1.
    - The G bits for all registers are set to 1.

• LDTR is unusable. The LDTR selector is cleared to 0000H, and the register is otherwise undefined (although the base address is always canonical)

• GDTR.base is set to the sum of the MSEG base address and the GDTR base-offset field in the MSEG header (bits 63:32 are always cleared on processors that support IA-32e mode). GDTR.limit is set to the corresponding field in the MSEG header (the high 16 bits are ignored).

• IDTR.base is unchanged. IDTR.limit is cleared to 0000H.

• RIP is set to the sum of the MSEG base address and the value of the RIP-offset field in the MSEG header (bits 63:32 are always cleared on logical processors that support IA-32e mode).

• RSP is set to the sum of the MSEG base address and the value of the RSP-offset field in the MSEG header (bits 63:32 are always cleared on logical processor that supports IA-32e mode).
• RFLAGS is cleared, except bit 1, which is always set.
• The logical processor is left in the active state.
• Event blocking after the SMM VM exit is as follows:
  — There is no blocking by STI or by MOV SS.
  — There is blocking by non-maskable interrupts (NMIs) and by SMIs.
• There are no pending debug exceptions after the SMM VM exit.
• For processors that support IA-32e mode, the IA32_EFER MSR is modified so that LME and LMA both contain the value of the IA-32e mode SMM feature bit.

If any of CR3[63:5], CR4.PAE, CR4.PSE, or IA32_EFER.LMA is changing, the TLBs are updated so that, after VM exit, the logical processor does not use translations that were cached before the transition. This is not necessary for changes that would not affect paging due to the settings of other bits (for example, changes to CR4.PSE if IA32_EFER.LMA was 1 before and after the transition).

32.15.6.6 Loading MSRs
The VM-exit MSR-load area is not used by SMM VM exits that activate the dual-monitor treatment. No MSRs are loaded from that area.

32.15.7 Deactivating the Dual-Monitor Treatment
The SMM-transfer monitor may deactivate the dual-monitor treatment and return the processor to default treatment of SMIs and SMM (see Section 32.14). It does this by executing a VM entry with the “deactivate dual-monitor treatment” VM-entry control set to 1.

As noted in Section 27.2.1.3 and Section 32.15.4.1, an attempt to deactivate the dual-monitor treatment fails in the following situations: (1) the processor is not in SMM; (2) the “entry to SMM” VM-entry control is 1; or (3) the executive-VMCS pointer does not contain the VMXON pointer (the VM entry is to VMX non-root operation).

As noted in Section 32.15.4.9, VM entries that deactivate the dual-monitor treatment ignore the SMI bit in the interruptibility-state field of the guest-state area. Instead, the blocking of SMIs following such a VM entry depends on whether the logical processor is in SMX operation:

• If the logical processor is in SMX operation, SMIs are blocked after VM entry. SMIs may later be unblocked by the VMXOFF instruction (see Section 32.14.4) or by certain leaf functions of the GETSEC instruction (see Chapter 7, “Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D).

• If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

32.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT
On processors that support processor extended states using XSAVE/XRSTOR (see Chapter 13, “Managing State Using the XSAVE Feature Set,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1), the processor does not save any XSAVE/XRSTOR related state on an SMI. It is the responsibility of the SMI handler code to properly preserve the state information (including CR4.OSXSAVE, XCR0, and possibly processor extended states using XSAVE/XRSTOR). Therefore, the SMI handler must follow the rules described in Chapter 13, “Managing State Using the XSAVE Feature Set,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 7, “Safer Mode Extensions Reference,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.
32.17 MODEL-SPECIFIC SYSTEM MANAGEMENT ENHANCEMENT

This section describes enhancement of system management features that apply only to the 4th generation Intel Core processors. These features are model-specific. BIOS and SMM handler must use CPUID to enumerate Display-Family_DisplayModel signature when programming with these interfaces.

32.17.1 SMM Handler Code Access Control

The BIOS may choose to restrict the address ranges of code that SMM handler executes. When SMM handler code execution check is enabled, an attempt by the SMM handler to execute outside the ranges specified by SMRR (see Section 32.4.2.1) will cause the assertion of an unrecoverable machine check exception (MCE).

The interface to enable SMM handler code access check resides in a per-package scope model-specific register MSR_SMM_FEATURE_CONTROL at address 4E0H. An attempt to access MSR_SMM_FEATURE_CONTROL outside of SMM will cause a #GP. Writes to MSR_SMM_FEATURE_CONTROL is further protected by configuration interface of MSR_SMM_MCA_CAP at address 17DH.

Details of the interface of MSR_SMM_FEATURE_CONTROL and MSR_SMM_MCA_CAP are described in Table 2-29 in Chapter 2, "Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

32.17.2 SMI Delivery Delay Reporting

Entry into the system management mode occurs at instruction boundary. In situations where a logical processor is executing an instruction involving a long flow of internal operations, servicing an SMI by that logical processor will be delayed. Delayed servicing of SMI of each logical processor due to executing long flows of internal operation in a physical processor can be queried via a package-scope register MSR_SMM_DELAYED at address 4E2H.

The interface to enable reporting of SMI delivery delay due to long internal flows resides in a per-package scope model-specific register MSR_SMM_DELAYED. An attempt to access MSR_SMM_DELAYED outside of SMM will cause a #GP. Availability to MSR_SMM_DELAYED is protected by configuration interface of MSR_SMM_MCA_CAP at address 17DH.

Details of the interface of MSR_SMM_DELAYED and MSR_SMM_MCA_CAP are described in Table 2-29 in Chapter 2, "Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.

32.17.3 Blocked SMI Reporting

A logical processor may have entered into a state and blocked from servicing other interrupts (including SMI). Logical processors in a physical processor that are blocked in serving SMI can be queried in a package-scope register MSR_SMM_BLOCKED at address 4E3H. An attempt to access MSR_SMM_BLOCKED outside of SMM will cause a #GP.

Details of the interface of MSR_SMM_BLOCKED is described in Table 2-29 in Chapter 2, "Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4.
33.1 OVERVIEW

Intel® Processor Trace (Intel PT) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in data packets. The initial implementations of Intel PT offer control flow tracing, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information (e.g., branch targets, branch taken/not taken indications) and program-induced mode related information (e.g., Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem or other output mechanism available in the platform. Debug software can process the trace data and reconstruct the program flow.

Intel Processor Trace was first introduced in Intel® processors based on Broadwell microarchitecture and Intel Atom® processors based on Goldmont microarchitecture. Later generations include additional trace sources, including software trace instrumentation using PTWRITE, and Power Event tracing.

33.1.1 Features and Capabilities

Intel PT’s control flow trace generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks).

Intel PT can also be configured to log software-generated packets using PTWRITE, and packets describing processor power management events. Further, Precise Event-Based Sampling (PEBS) can be configured to log PEBS records in the Intel PT trace; see Section 20.5.5.2.

In addition, the packets record other contextual, timing, and bookkeeping information that enables both functional and performance debugging of applications. Intel PT has several control and filtering capabilities available to customize the tracing information collected and to append other processor state and timing information to enable debugging. For example, there are modes that allow packets to be filtered based on the current privilege level (CPL) or the value of CR3.

Configuration of the packet generation and filtering capabilities are programmed via a set of MSRs. The MSRs generally follow the naming convention of IA32_RTIT_. The capability provided by these configuration MSRs are enumerated by CPUID, see Section 33.3. Details of the MSRs for configuring Intel PT are described in Section 33.2.8.

33.1.1.1 Packet Summary

After a tracing tool has enabled and configured the appropriate MSRs, the processor will collect and generate trace information in the following categories of packets (for more details on the packets, see Section 33.4):

- Packets about basic information on program execution; these include:
  - Packet Stream Boundary (PSB) packets: PSB packets act as ‘heartbeats’ that are generated at regular intervals (e.g., every 4K trace packet bytes). These packets allow the packet decoder to find the packet boundaries within the output data stream; a PSB packet should be the first packet that a decoder looks for when beginning to decode a trace.
  - Paging Information Packet (PIP): PIPs record modifications made to the CR3 register. This information, along with information from the operating system on the CR3 value of each process, allows the debugger to attribute linear addresses to their correct application source.
  - Time-Stamp Counter (TSC) packets: TSC packets aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.
  - Core Bus Ratio (CBR) packets: CBR packets contain the core:bus clock ratio.
— Mini Time Counter (MTC) packets: MTC packets provide periodic indication of the passing of wall-clock time.
— Cycle Count (CYC) packets: CYC packets provide indication of the number of processor core clock cycles that pass between packets.
— Overflow (OVF) packets: OVF packets are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.

• Packets about control flow information:
  — Taken Not-Taken (TNT) packets: TNT packets track the "direction" of direct conditional branches (taken or not taken).
  — Target IP (TIP) packets: TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP. There are various types of TIP packets; they are covered in more detail in Section 33.4.2.2.
  — Flow Update Packets (FUP): FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.
  — MODE packets: These packets provide the decoder with important processor execution information so that it can properly interpret the dis-assembled binary and trace log. MODE packets have a variety of formats that indicate details such as the execution mode (16-bit, 32-bit, or 64-bit).

• Packets inserted by software:
  — PTWRITE (PTW) packets: includes the value of the operand passed to the PTWRITE instruction (see "PTWRITE—Write Data to a Processor Trace Packet" in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B).

• Packets about processor power management events:
  — MWAIT packets: Indicate successful completion of an MWAIT operation to a C-state deeper than C0.0.
  — Power State Entry (PWRE) packets: Indicate entry to a C-state deeper than C0.0.
  — Power State Exit (PWRX) packets: Indicate exit from a C-state deeper than C0.0, returning to C0.
  — Execution Stopped (EXSTOP) packets: Indicate that software execution has stopped, due to events such as P-state change, C-state change, or thermal throttling.

• Packets containing groups of processor state values:
  — Block Begin Packets (BBP): Indicate the type of state held in the following group.
  — Block Item Packets (BIP): Indicate the state values held in the group.
  — Block End Packets (BEP): Indicate the end of the current group.

33.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL

This section describes the overall Intel Processor Trace mechanism and the essential concepts relevant to how it operates.

33.2.1 Change of Flow Instruction (COFI) Tracing

A basic program block is a section of code where no jumps or branches occur. The instruction pointers (IPs) in this block of code need not be traced, as the processor will execute them from start to end without redirecting code flow. Instructions such as branches, and events such as exceptions or interrupts, can change the program flow. These instructions and events that change program flow are called Change of Flow Instructions (COFI). There are three categories of COFI:
• Direct transfer COFI.
• Indirect transfer COFI.
• Far transfer COFI.
The following subsections describe the COFI events that result in trace packet generation. Table 33-1 lists branch instruction by COFI types. For detailed description of specific instructions, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual.

### 33.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two sub-categories:

- **Conditional Branch (Jcc, J*CXZ) and LOOP**
  
  To track this type of instruction, the processor encodes a single bit (taken or not taken — TNT) to indicate the program flow after the instruction.

  Jcc, J*CXZ, and LOOP can be traced with TNT bits. To improve the trace packet output efficiency, the processor will compact several TNT bits into a single packet.

- **Unconditional Direct Jumps**
  
  There is no trace output required for direct unconditional jumps (like JMP near relative or CALL near relative) since they can be directly inferred from the application assembly. Direct unconditional jumps do not generate a TNT bit or a Target IP packet, though TIP.PGD and TIP.PGE packets can be generated by unconditional direct jumps that toggle Intel PT enables (see Section 33.2.6).

### 33.2.1.2 Indirect Transfer COFI

Indirect transfer instructions involve updating the IP from a register or memory location. Since the register or memory contents can vary at any time during execution, there is no way to know the target of the indirect transfer until the register or memory contents are read. As a result, the disassembled code is not sufficient to determine the target of this type of COFI. Therefore, tracing hardware must send out the destination IP in the trace packet for debug software to determine the target address of the COFI. Note that this IP may be a linear or effective address (see Section 33.3.1.1).

An indirect transfer instruction generates a Target IP Packet (TIP) that contains the target address of the branch. There are two sub-categories:

- **Near JMP Indirect and Near Call Indirect**
  
  As previously mentioned, the target of an indirect COFI resides in the contents of either a register or memory location. Therefore, the processor must generate a packet that includes this target address to allow the decoder to determine the program flow.

- **Near RET**
  
  When a CALL instruction executes, it pushes onto the stack the address of the next instruction following the CALL. Upon completion of the call procedure, the RET instruction is often used to pop the return address off of the call stack and redirect code flow back to the instruction following the CALL.

  A RET instruction simply transfers program flow to the address it popped off the stack. Because a called procedure may change the return address on the stack before executing the RET instruction, debug software
can be misled if it assumes that code flow will return to the instruction following the last CALL. Therefore, even for near RET, a Target IP Packet may be sent.

— **RET Compression**

A special case is applied if the target of the RET is consistent with what would be expected from tracking the CALL stack. If it is assured that the decoder has seen the corresponding CALL (with “corresponding” defined as the CALL with matching stack depth), and the RET target is the instruction after that CALL, the RET target may be “compressed”. In this case, only a single TNT bit of “taken” is generated instead of a Target IP Packet. To ensure that the decoder will not be confused in cases of RET compression, only RETs that correspond to CALLs which have been seen since the last PSB packet may be compressed in a given logical processor. For details, see “Indirect Transfer Compression for Returns (RET)” in Section 33.4.2.2.

### 33.2.1.3 Far Transfer COFI

All operations that change the instruction pointer and are not near jumps are “far transfers”. This includes exceptions, interrupts, traps, TSX aborts, and instructions that do far transfers.

All far transfers will produce a Target IP (TIP) packet, which provides the destination IP address. For those far transfers that cannot be inferred from the binary source (e.g., asynchronous events such as exceptions and interrupts), the TIP will be preceded by a Flow Update packet (FUP), which provides the source IP address at which the event was taken. Table 33-23 indicates exactly which IP will be included in the FUP generated by a far transfer.

### 33.2.2 Software Trace Instrumentation with PTWRITE

PTWRITE provides a mechanism by which software can instrument the Intel PT trace. PTWRITE is a ring3-accessible instruction that can be passed to a register or memory variable, see “PTWRITE—Write Data to a Processor Trace Packet” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B, for details. The contents of that variable will be used as the payload for the PTW packet (see Table 33-40 “PTW Packet Definition”), inserted at the time of PTWRITE retirement, assuming PTWRITE is enabled and all other filtering conditions are met. Decode and analysis software will then be able to determine the meaning of the PTWRITE packet based on the IP of the associated PTWRITE instruction.

PTWRITE is enabled via IA32_RTIT_CTL.PTWEn[12] (see Table 33-6). Optionally, the user can use IA32_RTIT_CTL.FUPonPTW[5] to enable PTW packets to be followed by FUP packets containing the IP of the associated PTWRITE instruction. Support for PTWRITE is introduced in Intel Atom processors based on the Goldmont Plus microarchitecture.

### 33.2.3 Power Event Tracing

Power Event Trace is a capability that exposes core- and thread-level sleep state and power down transition information. When this capability is enabled, the trace will expose information about:

— Scenarios where software execution stops.
  - Due to sleep state entry, frequency change, or other powerdown.
  - Includes the IP, when in the tracing context.

— The requested and resolved hardware thread C-state.
  - Including indication of hardware autonomous C-state entry.

— The last and deepest core C-state achieved during a sleep session.

— The reason for C-state wake.

This information is in addition to the bus ratio (CBR) information provided by default after any powerdown, and the timing information (TSC, TMA, MTC, CYC) provided during or after a powerdown state.

Power Event Trace is enabled via IA32_RTIT_CTL.PwrEvtEn[4]. Support for Power Event Tracing is introduced in Intel Atom processors based on the Goldmont Plus microarchitecture.
33.2.4 Event Tracing

Event Trace is a capability that exposes details about the asynchronous events, when they are generated, and when their corresponding software event handler completes execution. These include:

- Interrupts, including NMI and SMI, including the interrupt vector when defined.
- Faults, exceptions including the fault vector.
  - Page faults additionally include the page fault address, when in context.
- Event handler returns, including IRET and RSM.
- VM exits and VM entries.\(^1\)
  - VM exits include the values written to the "exit reason" and "exit qualification" VMCS fields.
- INIT and SIPI events.
- TSX aborts, including the abort status returned for the RTM instructions.
- Shutdown.

Additionally, it provides indication of the status of the Interrupt Flag (IF), to indicate when interrupts are masked.

Event Trace is enabled via IA32_RTIT_CTL.EventEn[31]. Event Trace information is conveyed in Control Flow Event (CFE) and Event Data (EVD) packets, as well as the legacy MODE.Exec packet. See Section 33.4.2 for packet details. Support for Event Trace is introduced in Intel® processors based on Gracemont microarchitecture.

33.2.5 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

33.2.5.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when CPL > 0, and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 33.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 33.2.6.1 for details.

33.2.5.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which the generation of packets containing architectural states can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSRs. For the reconstruction of traces from software with multiple threads, debug software may wish to context-switch for the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 33.3.5, "Context Switch Consideration").

To trace for only a single CR3 value, software can write that value to the IA32_RTIT_CR3_MATCH MSR, and set IA32_RTIT_CTL.CR3Filter. When CR3 value does not match IA32_RTIT_CR3_MATCH and IA32_RTIT_CTL.CR3Filter is 1, ContextEn is forced to 0, and packets containing architectural states will not be generated. Some other packets can be generated when ContextEn is 0; see Section 33.2.6.3 for details. When CR3 does match IA32_RTIT_CR3_MATCH (or when IA32_RTIT_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

1. Logging of VMX transitions depends on VMCS configuration, see Section 33.5.1.
CR3 matches IA32_RTIT_CR3_MATCH if the two registers are identical for bits 63:12, or 63:5 when in PAE paging mode; the lower 5 bits of CR3 and IA32_RTIT_CR3_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when CPL = 0 (IA32_RTIT_CTL.OS = 1). If not, no PIP packets will be seen.

### 33.2.5.3 Filtering by IP

Trace packet generation with configurable filtering by IP is supported if CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 1. Intel PT can be configured to enable the generation of packets containing architectural states only when the processor is executing code within certain IP ranges. If the IP is outside of these ranges, generation of some packets is blocked.

IP filtering is enabled using the ADDRn_CFG fields in the IA32_RTIT_CTL MSR (Section 33.2.8.2), where the digit ‘n’ is a zero-based number that selects which address range is being configured. Each ADDRn_CFG field configures the use of the register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B (Section 33.2.8.5). IA32_RTIT_ADDRn_A defines the base and IA32_RTIT_ADDRn_B specifies the limit of the range in which tracing is enabled. Thus each range, referred to as the ADDRn range, is defined by [IA32_RTIT_ADDRn_A, IA32_RTIT_ADDRn_B]. There can be multiple such ranges, software can query CPUID (Section 33.3.1) for the number of ranges supported on a processor.

Default behavior (ADDRn_CFG=0) defines no IP filter range, meaning FilterEn is always set. In this case code at any IP can be traced, though other filters, such as CR3 or CPL, could limit tracing. When ADDRn_CFG is set to enable IP filtering (see Section 33.3.1), tracing will commence when a taken branch or event is seen whose target address is in the ADDRn range.

While inside a tracing region and with FilterEn is set, leaving the tracing region may only be detected once a taken branch or event with a target outside the range is retired. If an ADDRn range is entered or exited by executing the next sequential instruction, rather than by a control flow transfer, FilterEn may not toggle immediately. See Section 33.2.6.5 for more details on FilterEn.

Note that these address range base and limit values are inclusive, such that the range includes the first and last instruction whose first instruction byte is in the ADDRn range.

Depending upon processor implementation, IP filtering may be based on linear or effective address. This can cause different behavior between implementations if CSbase is not equal to zero or in real mode. See Section 33.3.1.1 for details. Software can query CPUID to determine filters are based on linear or effective address (Section 33.3.1).

Note that some packets, such as MTC (Section 33.3.7) and other timing packets, do not depend on FilterEn. For details on which packets depend on FilterEn, and hence are impacted by IP filtering, see Section 33.4.1.

### TraceStop

The ADDRn ranges can also be configured to cause tracing to be disabled upon entry to the specified region. This is intended for cases where unexpected code is executed, and the user wishes to immediately stop generating packets in order to avoid overwriting previously written packets.

The TraceStop mechanism works much the same way that IP filtering does, and uses the same address comparison logic. The TraceStop region base and limit values are programmed into one or more ADDRn ranges, but IA32_RTIT_CTL.ADDRn_CFG is configured with the TraceStop encoding. Like FilterEn, TraceStop is detected when a taken branch or event lands in a TraceStop region.

Further, TraceStop requires that TriggerEn=1 at the beginning of the branch/event, and ContextEn=1 upon completion of the branch/event. When this happens, the CPU will set IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn and hence disabling packet generation. This may generate a TIP.PGD packet with the target IP of the branch or event that entered the TraceStop region. Finally, a TraceStop packet will be inserted, to indicate that the condition was hit.

If a TraceStop condition is encountered during buffer overflow (Section 33.3.8), it will not be dropped, but will instead be signaled once the overflow has resolved.

Note that a TraceStop event does not guarantee that all internally buffered packets are flushed out of internal buffers. To ensure that this has occurred, the user should clear TraceEn.
To resume tracing after a TraceStop event, the user must first disable Intel PT by clearing IA32_RTIT_CTL.TraceEn before the IA32_RTIT_STATUS.Stopped bit can be cleared. At this point Intel PT can be reconfigured, and tracing resumed.

Note that the IA32_RTIT_STATUS.Stopped bit can also be set using the ToPA STOP bit. See Section 33.2.7.2.

**IP Filtering Example**

The following table gives an example of IP filtering behavior. Assume that IA32_RTIT_ADDRn_A = the IP of RangeBase, and that IA32_RTIT_ADDRn_B = the IP of RangeLimit, while IA32_RTIT_CTL.ADDRn_CFG = 0x1 (enable ADDRn range as a FilterEn range).

<table>
<thead>
<tr>
<th>Code Flow</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar:</td>
<td></td>
</tr>
<tr>
<td>jmp RangeBase // jump into filter range</td>
<td>TIP.PGE(RangeBase)</td>
</tr>
<tr>
<td>RangeBase:</td>
<td>TNT(0)</td>
</tr>
<tr>
<td>jcc Foo // not taken</td>
<td>TIP.PGD(RangeLimit+1)</td>
</tr>
<tr>
<td>add eax, 1</td>
<td></td>
</tr>
<tr>
<td>Foo:</td>
<td></td>
</tr>
<tr>
<td>jmp RangeLimit+1 // jump out of filter range</td>
<td></td>
</tr>
<tr>
<td>RangeLimit:</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>jcc Bar</td>
<td></td>
</tr>
</tbody>
</table>

**IP Filtering and TraceStop**

It is possible for the user to configure IP filter range(s) and TraceStop range(s) that overlap. In this case, code executing in the non-overlapping portion of either range will behave as would be expected from that range. Code executing in the overlapping range will get TraceStop behavior.

**33.2.6 Packet Generation Enable Controls**

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (PacketEn) is set. PacketEn is an internal state maintained in hardware in response to software configurable enable controls, PacketEn is not visible to software directly. The relationship of PacketEn to the software-visible controls in the configuration MSRs is described in this section.

**33.2.6.1 Packet Enable (PacketEn)**

When PacketEn is set, the processor is in the mode that Intel PT is monitoring. PacketEn is composed of other states according to this relationship:

\[
\text{PacketEn} := \text{TriggerEn} \text{ AND ContextEn} \text{ AND FilterEn AND BranchEn}
\]

These constituent controls are detailed in the following subsections. PacketEn ultimately determines when the processor is tracing. When PacketEn is set, all control flow packets are enabled. When PacketEn is clear, no control flow packets are generated, though other packets (timing and bookkeeping packets) may still be sent. See Section 33.2.7 for details of PacketEn and packet generation.

Note that, on processors that do not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 0), FilterEn is treated as always set.
33.2.6.2 Trigger Enable (TriggerEn)

Trigger Enable (TriggerEn) is the primary indicator that trace packet generation is active. TriggerEn is set when IA32_RTIT_CTL.TraceEn is set, and cleared by any of the following conditions:

- TraceEn is cleared by software.
- A TraceStop condition is encountered and IA32_RTIT_STATUS.Stopped is set.
- IA32_RTIT_STATUS.Error is set due to an operational error (see Section 33.3.10).

Software can discover the current TriggerEn value by reading the IA32_RTIT_STATUS.TriggerEn bit. When TriggerEn is clear, tracing is inactive and no packets are generated.

33.2.6.3 Context Enable (ContextEn)

Context Enable (ContextEn) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with CPL = 0 code is not being traced (IA32_RTIT_CTL.OS = 0), then ContextEn will be 0 when the processor is in CPL0.

Software can discover the current ContextEn value by reading the IA32_RTIT_STATUS.ContextEn bit. ContextEn is defined as follows:

\[
\text{ContextEn} = \neg ((\text{IA32_RTIT_CTL.OS} = 0 \text{ AND CPL} = 0) \text{ OR } (\text{IA32_RTIT_CTL.USER} = 0 \text{ AND CPL} > 0) \text{ OR } (\text{IS_IN_A_PRODUCTION_ENCLAVE}) \text{ OR } (\text{IA32_RTIT_CTL.CR3Filter} = 1 \text{ AND IA32_RTIT_CR3_MATCH does not match } CR3))
\]

If the clearing of ContextEn causes PacketEn to be cleared, a Packet Generation Disable (TIP.PGD) packet is generated, but its IP payload is suppressed. If the setting of ContextEn causes PacketEn to be set, a Packet Generation Enable (TIP.PGE) packet is generated.

When ContextEn is 0, control flow packets (TNT, FUP, TIP.*, MODE.*) are not generated, and no Linear Instruction Pointers (LIPs) are exposed. However, some packets, such as MTC and PSB (see Section 33.4.2.16 and Section 33.4.2.17), may still be generated while ContextEn is 0. For details of which packets are generated only when ContextEn is set, see Section 33.4.1.

The processor does not update ContextEn when TriggerEn = 0. The value of ContextEn will toggle only when TriggerEn = 1.

33.2.6.4 Branch Enable (BranchEn)

This value is based purely on the IA32_RTIT_CTL.BranchEn value. If BranchEn is not set, then relevant COFI packets (TNT, TIP*, FUP, MODE.*) are suppressed. Other packets related to timing (TSC, TMA, MTC, CYC), as well as PSB, will be generated normally regardless. Further, PIP and VMCS continue to be generated, as indicators of what software is running.

33.2.6.5 Filter Enable (FilterEn)

Filter Enable indicates that the Instruction Pointer (IP) is within the range of IPs that Intel PT is configured to watch. Software can get the state of Filter Enable by a RDMSR of IA32_RTIT_STATUS.FilterEn. For details on configuration and use of IP filtering, see Section 33.2.5.3.

On clearing of FilterEn that also clears PacketEn, a Packet Generation Disable (TIP.PGD) will be generated, but unlike the ContextEn case, the IP payload may not be suppressed. For direct, unconditional branches, as well as for indirect branches (including RETs), the PGD generated by leaving the tracing region and clearing FilterEn will contain the target IP. This means that IPs from outside the configured range can be exposed in the trace, as long as they are within context.

When FilterEn is 0, control flow packets are not generated (e.g., TNT, TIP). However, some packets, such as PIP, MTC, and PSB, may still be generated while FilterEn is clear. For details on packet enable dependencies, see Section 33.4.1.

---

1. Trace packets generation is disabled in a production enclave, see Section 33.2.9.5. See the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D, about differences between a production enclave and a debug enclave.
After TraceEn is set, FilterEn is set to 1 at all times if there is no IP filter range configured by software (IA32_RTIT_CTL.ADDRn_CFG != 1, for all n), or if the processor does not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 0). FilterEn will toggle only when TraceEn=1 and ContextEn=1, and when at least one range is configured for IP filtering.

33.2.7 Trace Output

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms.

Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bit fields of IA32_RTIT_CTL (see Section 33.2.8.2):

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (ToPA). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.
- A platform-specific trace transport subsystem.

Regardless of the output scheme chosen, Intel PT stores bypass the processor caches by default. This ensures that they don't consume precious cache space, but they do not have the serializing aspects associated with un-cacheable (UC) stores. Software should avoid using MTRRs to mark any portion of the Intel PT output region as UC, as this may override the behavior described above and force Intel PT stores to UC, thereby incurring severe performance impact.

There is no guarantee that a packet will be written to memory or other trace endpoint after some fixed number of cycles after a packet-producing instruction executes. The only way to assure that all packets generated have reached their endpoint is to clear TraceEn and follow that with a store, fence, or serializing instruction; doing so ensures that all buffered packets are flushed out of the processor.

33.2.7.1 Single Range Output

When IA32_RTIT_CTL.ToPA and IA32_RTIT_CTL.FabricEn bits are clear, trace packet output is sent to a single, contiguous memory (or MMIO if DRAM is not available) range defined by a base address in IA32_RTIT_OUTPUT_BASE (Section 33.2.8.7) and mask value in IA32_RTIT_OUTPUT_MASK_PTRS (Section 33.2.8.8). The current write pointer in this range is also stored in IA32_RTIT_OUTPUT_MASK_PTRS. This output range is circular, meaning that when the writes wrap around the end of the buffer they begin again at the base address.

This output method is best suited for cases where Intel PT output is either:
- Configured to be directed to a sufficiently large contiguous region of DRAM.
- Configured to go to an MMIO debug port, in order to route Intel PT output to a platform-specific trace endpoint (e.g., JTAG). In this scenario, a specific range of addresses is written in a circular manner, and SoC will intercept these writes and direct them to the proper device. Repeated writes to the same address do not overwrite each other, but are accumulated by the debugger, and hence no data is lost by the circular nature of the buffer.

The processor will determine the address to which to write the next trace packet output byte as follows:

```
OutputBase[63:0] := IA32_RTIT_OUTPUT_BASE[63:0]
OutputMask[63:0] := ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[31:0])
OutputOffset[63:0] := ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[63:32])

trace_store_phys_addr := (OutputBase & ~OutputMask) + (OutputOffset & OutputMask)
```
### Single-Range Output Errors

If the output base and mask are not properly configured by software, an operational error (see Section 33.3.10) will be signaled, and tracing disabled. Error scenarios with single-range output are:

- **Mask value is non-contiguous.**
  
  $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTablePointer}$ value has a 0 in a less significant bit position than the most significant bit containing a 1.

- **Base address and Mask are mis-aligned, and have overlapping bits set.**
  
  $\text{IA32\_RTIT\_OUTPUT\_BASE} \&\& \text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS[31:0]} > 0$.

- **Illegal Output Offset**
  
  $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset}$ is greater than the mask value $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS[31:0]}$.

Also note that errors can be signaled due to trace packet output overlapping with restricted memory, see Section 33.2.7.4.

### 33.2.7.2 Table of Physical Addresses (ToPA)

When $\text{IA32\_RTIT\_CTL.ToPA}$ is set and $\text{IA32\_RTIT\_CTL.FabricEn}$ is clear, the ToPA output mechanism is utilized. The ToPA mechanism uses a linked list of tables; see Figure 33-1 for an illustrative example. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table (for circular array) or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry.

The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

The ToPA mechanism is controlled by three values maintained by the processor:

- **proc_trace_table_base.**
  
  This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the $\text{IA32\_RTIT\_OUTPUT\_BASE}$ MSR. While tracing is enabled, the processor updates the $\text{IA32\_RTIT\_OUTPUT\_BASE}$ MSR with changes to $\text{proc\_trace\_table\_base}$, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of $\text{proc\_trace\_table\_base}$.

- **proc_trace_table_offset.**
  
  This indicates the entry of the current table that is currently in use. (This entry contains the address of the current output region.) When tracing is enabled, the processor loads the value from bits 31:7 ($\text{MaskOrTableOffset}$) of the $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS}$ into bits 27:3 of $\text{proc\_trace\_table\_offset}$. While tracing is enabled, the processor updates $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS.MaskOrTableOffset}$ with changes to $\text{proc\_trace\_table\_offset}$, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of $\text{proc\_trace\_table\_offset}$.

- **proc_trace_output_offset.**
  
  This a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 ($\text{OutputOffset}$) of the $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS}$. While tracing is enabled, the processor updates $\text{IA32\_RTIT\_OUTPUT\_MASK\_PTRS.OutputOffset}$ with changes to $\text{proc\_trace\_output\_offset}$, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of $\text{proc\_trace\_output\_offset}$. 


Figure 33-1 provides an illustration (not to scale) of the table and associated pointers.

With the ToPA mechanism, the processor writes packets to the current output region (identified by proc_trace_table_base and the proc_trace_table_offset). The offset within that region to which the next byte will be written is identified by proc_trace_output_offset. When that region is filled with packet output (thus proc_trace_output_offset = RegionSize–1), proc_trace_table_offset is moved to the next ToPA entry, proc_trace_output_offset is set to 0, and packet writes begin filling the new output region specified by proc_trace_table_offset.

As packets are written out, each store derives its physical address as follows:

\[
\text{trace_store_phys_addr} := \text{Base address from current ToPA table entry} + \text{proc_trace_output_offset}
\]

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Table 33-3 and the section that follows for details on STOP.

When an END entry is reached, the processor loads proc_trace_table_base with the base address held in this END entry, thereby moving the current table pointer to this new table. The proc_trace_table_offset is reset to 0, as is the proc_trace_output_offset, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (proc_trace_table_offset = 0FFFFFF8H). In this case, the proc_trace_table_offset and proc_trace_output_offset are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.
It is important to note that processor updates to the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT MASK_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs while Intel PT is enabled may return stale values. Like all IA32_RTIT_* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32_RTIT_CTL.TraceEn. This ensures that the output MSR values account for all packets generated to that point, after which the processor will cease updating the output MSR values until tracing resumes. The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

**Single Output Region ToPA Implementation**

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0 and CPUID.(EAX=14H,ECX=0):ECX.TOPAOUT[bit 0] = 1.

If CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0, ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Hence only one contiguous block can be used as output.

The lone output entry can have INT or STOP set, but nonetheless must be followed by an END entry as described above. Note that, if INT=1, the PMI will actually be delivered before the region is filled.

**ToPA Table Entry Format**

The format of ToPA table entries is shown in Figure 33-2. The size of the address field is determined by the processor’s physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

<table>
<thead>
<tr>
<th>Table 33-3. ToPA Table Entry Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ToPA Entry Field</strong></td>
</tr>
<tr>
<td>Output Region Base Physical Address</td>
</tr>
</tbody>
</table>

---

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.
ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32_RTIT_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 33.2.8.4 for details on IA32_RTIT_STATUS.Stopped. This sequence is known as “ToPA Stop”. No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, output ceases after the last byte of the region is filled, which may mean that a packet is cut off in the middle. Any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32_RTIT_OUTPUT_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset will hold the index value for that entry, and the IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset should be set to the size of the region minus one.

Note that this means the offset pointer is pointing to the next byte after the end of the region, a configuration that would produce an operational error if the configuration remained when tracing is re-enabled with IA32_RTIT_STATUS.Stopped cleared.

ToPA PMI

Each ToPA entry has an INT bit. If this bit is set, the processor will signal a performance-monitoring interrupt (PMI) when the corresponding trace output region is filled. This interrupt is not precise, and it is thus likely that writes to the next region will occur by the time the interrupt is taken.

The following steps should be taken to configure this interrupt:

1. Enable PMI via the LVT Performance Monitor register (at MMIO offset 340H in xAPIC mode; via MSR 834H in x2APIC mode). See the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, for more details on this register. For ToPA PMI, set all fields to 0, save for the interrupt vector, which can be selected by software.

2. Set up an interrupt handler to service the interrupt vector that a ToPA PMI can raise.

3. Set the interrupt flag by executing STI.

4. Set the INT bit in the ToPA entry of interest and enable packet generation, using the ToPA output option. Thus, TraceEn=ToPA=1 in the IA32_RTIT_CTL MSR.

Once the INT region has been filled with packet output data, the interrupt will be signaled. This PMI can be distinguished from others by checking bit 55 (Trace_ToPA_PMI) of the IA32_PERF_GLOBAL_STATUS MSR (MSR 38EH). Once the ToPA PMI handler has serviced the relevant buffer, writing 1 to bit 55 of the MSR at 390H (IA32_GLOBAL_STATUS_RESET) clears IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI.

Table 33-3. ToPA Table Entry Fields (Contd.)

<table>
<thead>
<tr>
<th>ToPA Entry Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).</td>
</tr>
<tr>
<td>INT</td>
<td>When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).</td>
</tr>
<tr>
<td>END</td>
<td>If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be set in the second table entry.</td>
</tr>
</tbody>
</table>
Intel PT is not frozen on PMI, and thus the interrupt handler will be traced (though filtering can prevent this). The Freeze_Perfmon_on_PMI and Freeze_LBRs_on_PMI settings in IA32_DEBUGCTL will be applied on ToPA PMI just as on other PMIs, and hence Perfmon counters are frozen.

Assuming the PMI handler wishes to read any buffered packets for persistent output, or wishes to modify any Intel PT MSRs, software should first disable packet generation by clearing TraceEn. This ensures that all buffered packets are written to memory and avoids tracing of the PMI handler. The configuration MSRs can then be used to determine where tracing has stopped. If packet generation is disabled by the handler, it should then be manually re-enabled before the IRET if continued tracing is desired.

In rare cases, it may be possible to trigger a second ToPA PMI before the first is handled. This can happen if another ToPA region with INT=1 is filled before, or shortly after, the first PMI is taken, perhaps due to EFLAGS.IF being cleared for an extended period of time. This can manifest in two ways: either the second PMI is triggered before the first is taken, and hence only one PMI is taken, or the second is triggered after the first is taken, and thus will be taken when the handler for the first completes. Software can minimize the likelihood of the second case by clearing TraceEn at the beginning of the PMI handler. Further, it can detect such cases by then checking the Interrupt Request Register (IRR) for PMI pending, and checking the ToPA table base and off-set pointers (in IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) to see if multiple entries with INT=1 have been filled.

PMI Preservation

In some cases a ToPA PMI may be taken after completion of an XSAVES instruction that saves Intel PT state, and in such cases any modification of Intel PT MSRs within the PMI handler will not persist when the saved Intel PT context is later restored with XRSTORS. To account for such a scenario, the PMI Preservation feature has been added. Support for this feature is indicated by CPUID.(EAX=14H, ECX=0):EBX[bit 6].

When IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, PMI preservation is enabled. When a ToPA region with INT=1 is filled, a PMI is pended and the new IA32_RTIT_STATUS.PendToPAPMI[7] is set to 1. If this bit is set when Intel PT is enabled, such that IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a ToPA PMI is pended. This behavior ensures that any ToPA PMI that is pended during XSAVES, and hence can’t be properly handled, will be re-pended when the saved PT state is restored.

When this feature is enabled, the PMI handler should take the following actions:

1. Ignore ToPA PMIs that are taken when TraceEn = 0. This indicates that the PMI was pended during Intel PT disable, and the PendToPAPMI flag will ensure that the PMI is re-pended once Intel PT is re-enabled in the same context. For this reason, the PendToPAPMI bit should be left set to 1.

2. If TraceEn=1 and the PMI can be properly handled, clear the new PendToPAPMI bit. This will ensure that additional, spurious ToPA PMIs are not taken. It is required that PendToPAPMI is cleared before the PMI LVT mask is cleared in the APIC, and before any clearing of either LBRS_FROZEN or COUNTERS_FROZEN in IA32_PERF_GLOBAL_STATUS.

ToPA PMI and Single Output Region ToPA Implementation

A processor that supports only a single ToPA output region implementation (such that only one output region is supported; see above) will attempt to signal a ToPA PMI interrupt before the output wraps and overwrites the top of the buffer. To support this functionality, the PMI handler should disable packet generation as soon as possible.

Due to PMI skid, it is possible that, in rare cases, the wrap will have occurred before the PMI is delivered. Software can avoid this by setting the STOP bit in the ToPA entry (see Table 33-3); this will disable tracing once the region is filled, and no wrap will occur. This approach has the downside of disabling packet generation so that some of the instructions that led up to the PMI will not be traced. If the PMI skid is significant enough to cause the region to fill and tracing to be disabled, the PMI handler will need to clear the IA32_RTIT_STATUS.Stopped indication before tracing can resume.
ToPA PMI and XSAVES/XRSTORS State Handling

In some cases the ToPA PMI may be taken after completion of an XSAVES instruction that switches Intel PT state, and in such cases any modification of Intel PT MSRs within the PMI handler will not persist when the saved Intel PT context is later restored with XRSTORS. To account for such a scenario, it is recommended that the Intel PT output configuration be modified by altering the ToPA tables themselves, rather than the Intel PT output MSRs. On processors that support PMI preservation (CPUID.(EAX=14H, ECX=0):EBX[bit 6] = 1), setting IA32_RTIT_CTL.InjectPsbsPmiOnEnable[56] = 1 will ensure that a PMI that is pending at the time PT is disabled will be recorded by setting IA32_RTIT_STATUS.PendTopaPmi[7] = 1. A PMI will then be pended when the saved PT context is later restored.

Table 33-4 depicts a recommended PMI handler algorithm for managing multi-region ToPA output and handling ToPA PMIs that may arrive between XSAVES and XRSTORS, if PMI preservation is not in use. This algorithm is flexible to allow software to choose between adding entries to the current ToPA table, adding a new ToPA table, or using the current ToPA table as a circular buffer. It assumes that the ToPA entry that triggers the PMI is not the last entry in the table, which is the recommended treatment.

| Table 33-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS |

| Pseudo Code Flow |

IF (IA32_PERF_GLOBAL_STATUS.ToPA)
    Save IA32_RTIT_CTL value;
    IF (IA32_RTIT_CTL.TraceEN)
        Disable Intel PT by clearing TraceEn;
    FI;
    IF (there is space available to grow the current ToPA table)
        Add one or more ToPA entries after the last entry in the ToPA table;
        Point new ToPA entry address field(s) to new output region base(s);
    ELSE
        Modify an upcoming ToPA entry in the current table to have END=1;
        IF (output should transition to a new ToPA table)
            Point the address of the “END=1” entry of the current table to the new table base;
        ELSE
            /* Continue to use the current ToPA table, make a circular. */
            Point the address of the “END=1” entry to the base of the current table;
            Modify the ToPA entry address fields for filled output regions to point to new, unused output regions;
            /* Filled regions are those with index in the range of 0 to (IA32_RTIT_MASK_PTRS.MaskOrTableOffset -1). */
            FI;
        FI;
    FI;
Restore saved IA32_RTIT_CTL.value;
FI;

ToPA Errors

When a malformed ToPA entry is found, an operational error results (see Section 33.3.10). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation.**
   This describes cases where a bit marked as reserved in Section 33.2.7.2 above is set to 1.

2. **ToPA alignment violation.**
   This includes cases where illegal ToPA entry base address bits are set to 1:
   a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32_RTIT_OUTPUT_BASE, or from a ToPA entry with END=1.
   b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0), for ToPA entries with END=0.
   c. ToPA entry base address sets upper physical address bits not supported by the processor.
3. **Illegal ToPA Output Offset.**
   IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than or equal to the size of the current ToPA output region size.

4. **ToPA rules violations.**
   These are similar to ToPA entry reserved bit violations; they are cases when a ToPA entry is encountered with illegal field combinations. They include the following:
   a. Setting the STOP or INT bit on an entry with END=1.
   b. Setting the END bit in entry 0 of a ToPA table.
   c. On processors that support only a single ToPA entry (see above), two additional illegal settings apply:
      i) ToPA table entry 1 with END=0.
      ii) ToPA table entry 1 with base address not matching the table base.
   In all cases, the error will be logged by setting IA32_RTIT_STATUS.Error, thereby disabling tracing when the problematic ToPA entry is reached (when proc_trace_table_offset points to the entry containing the error). Any packet bytes that are internally buffered when the error is detected may be lost.

   Note that operational errors may also be signaled due to attempts to access restricted memory. See Section 33.2.7.4 for details.

   A tracing software have a range of flexibility using ToPA to manage the interaction of Intel PT with application buffers, see Section 33.4.2.26.

33.2.7.3  Trace Transport Subsystem

When IA32_RTIT_CTL.FabricEn is set, the IA32_RTIT_CTL.ToPA bit is ignored, and trace output is written to the trace transport subsystem. The endpoints of this transport are platform-specific, and details of configuration options should refer to the specific platform documentation. The FabricEn bit is available to be set if CPUID(EAX=14H,ECX=0):EBX[bit 3] = 1.

33.2.7.4  Restricted Memory Access

Packet output cannot be directed to any regions of memory that are restricted by the platform. In particular, all memory accesses on behalf of packet output are checked against the SMRR regions. If there is any overlap with these regions, trace data collection will not function properly. Exact processor behavior is implementation-dependent; Table 33-5 summarizes several scenarios.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToPA output region overlaps with SMRR</td>
<td>Stores to the restricted memory region will be dropped, and that packet data will be lost. Any attempt to read from that restricted region will return all 1s. The processor also may signal an error (Section 33.3.10) and disable tracing when the output pointer reaches the restricted region. If packet generation remains enabled, then packet output may continue once stores are no longer directed to restricted memory (on wrap, or if the output region is larger than the restricted memory region).</td>
</tr>
<tr>
<td>ToPA table overlaps with SMRR</td>
<td>The processor will signal an error (Section 33.3.10) and disable tracing when the ToPA write pointer (IA32_RTIT_OUTPUT_BASE + proc_trace_table_offset) enters the restricted region.</td>
</tr>
</tbody>
</table>

It should also be noted that packet output should not be routed to the 4KB APIC MMIO region, as defined by the IA32_APIC_BASE MSR. For details about the APIC, refer to the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A. No error is signaled for this case.

**Modifications to Restricted Memory Regions**

It is recommended that software disable packet generation before modifying the SMRRs to change the scope of the SMRR regions. This is because the processor reserves the right to cache any number of ToPA table entries internally, after checking them against restricted memory ranges. Once cached, the entries will not be checked again, meaning one could potentially route packet output to a newly restricted region. Software can ensure that any cached entries are written to memory by clearing IA32_RTIT_CTL.TraceEn.
33.2.8 Enabling and Configuration MSRs

33.2.8.1 General Considerations
Trace packet generation is enabled and configured by a collection of model-specific registers (MSRs), which are detailed below. Some notes on the configuration MSR behavior:

- If Intel Processor Trace is not supported by the processor (see Section 33.3.1), RDMSR or WRMSR of the IA32_RTIT_* MSRs will cause #GP.

- A WRMSR to any of the IA32_RTIT_* configuration MSRs while packet generation is enabled (IA32_RTIT_CTL.TraceEn=1) will generate a #GP exception. Packet generation must be disabled before the configuration MSRs can be changed.

  Note: Software may write the same value back to IA32_RTIT_CTL without #GP, even if TraceEn=1.

- All configuration MSRs for Intel PT are duplicated per logical processor

- For each configuration MSR, any MSR write that attempts to change bits marked reserved, or utilize encodings marked reserved, will cause a #GP fault.

- All configuration MSRs for Intel PT are cleared on a warm or cold RESET.
  
  — If CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 1, only the TraceEn bit is cleared on warm RESET; though this may have the impact of clearing other bits in IA32_RTIT_STATUS. Other MSR values of the trace configuration MSRs are preserved on warm RESET.

- The semantics of MSR writes to trace configuration MSRs in this chapter generally apply to explicit WRMSR to these registers, using VMexit or VM entry MSR load list to these MSRs, XRSTORS with requested feature bit map including XSAVE map component of state_8 (corresponding to IA32_XSS[bit 8]), and the write to IA32_RTIT_CTL.TraceEn by XSAVES (Section 33.2.8.3).

33.2.8.2 IA32_RTIT_CTL MSR
IA32_RTIT_CTL, at address 570H, is the primary enable and control MSR for trace packet generation. Bit positions are listed in Table 33-6.

Table 33-6. IA32_RTIT_CTL MSR

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TraceEn</td>
<td>0</td>
<td>If 1, enables tracing; else tracing is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>When this bit transitions from 1 to 0, all buffered packets are flushed out of internal buffers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A further store, fence, or architecturally serializing instruction may be required to ensure that packet data can be observed at the trace endpoint. See Section 33.2.8.3 for details of enabling and disabling packet generation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note that the processor will clear this bit on #SMI (Section 33.2.9.3) and warm reset. Other MSR bits of IA32_RTIT_CTL (and other trace configuration MSRs) are not impacted by these events.</td>
</tr>
<tr>
<td>1</td>
<td>CYCEn</td>
<td>0</td>
<td>0: Enables CYC Packet (see Section 33.4.2.14).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: Enables CYC Packet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 1] = 0.</td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td>0</td>
<td>0: Packet generation is disabled when CPL = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: Packet generation may be enabled when CPL = 0.</td>
</tr>
<tr>
<td>3</td>
<td>User</td>
<td>0</td>
<td>0: Packet generation is disabled when CPL &gt; 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: Packet generation may be enabled when CPL &gt; 0.</td>
</tr>
<tr>
<td>4</td>
<td>PwrEvtEn</td>
<td>0</td>
<td>0: Power Event Trace packets are disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: Power Event Trace packets are enabled (see Section 33.2.3, &quot;Power Event Tracing&quot;).</td>
</tr>
</tbody>
</table>
### Table 33-6. IA32_RTIT_CTL MSR (Contd.)

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
</table>
| 5        | FUPonPTW         | 0        | 0: PTW packets are not followed by FUPs.  
1: PTW packets are followed by FUPs.  
This bit is reserved when CPUID.(EAX=14H, ECX=0):EBX[bit 4] ("PTWRITE Supported") is 0.                                                                 |
| 6        | FabricEn         | 0        | 0: Trace output is directed to the memory subsystem, mechanism depends on IA32_RTIT_CTL.ToPA.  
1: Trace output is directed to the trace transport subsystem, IA32_RTIT_CTL.ToPA is ignored.  
This bit is reserved if CPUID.(EAX=14H, ECX=0):ECX[bit 3] = 0.                                                                 |
| 7        | CR3Filter        | 0        | 0: Disables CR3 filtering.  
1: Enables CR3 filtering.  
This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 0] ("CR3 Filtering Support") is 0.                                                                 |
| 8        | ToPA             | 0        | 0: Single-range output scheme enabled if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 1 and IA32_RTIT_CTL.FabricEn=0.  
1: ToPA output scheme enabled (see Section 33.2.7.2) if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 1, and IA32_RTIT_CTL.FabricEn=0.  
Note: WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit and FabricEn would cause #GP, if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 0.  
WRMSR to IA32_RTIT_CTL that sets this bit causes #GP, if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 0.                                                                 |
| 9        | MTCEn            | 0        | 0: Disables MTC Packet (see Section 33.4.2.16).  
1: Enables MTC Packet.  
This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 3] = 0.                                                                 |
| 10       | TSCEn            | 0        | 0: Disable TSC packets.  
1: Enable TSC packets (see Section 33.4.2.11).                                                                                                                                  |
| 11       | DisRETC          | 0        | 0: Enable RET compression.  
1: Disable RET compression (see Section 33.2.1.2).                                                                                                                                 |
| 12       | PTWEn            | 0        | 0: PTWRITE packet generation disabled.  
1: PTWRITE packet generation enabled (see Table 33-40 "PTW Packet Definition").  
This bit is reserved when CPUID.(EAX=14H, ECX=0):EBX[bit 4] ("PTWRITE Supported") is 0.                                                                 |
| 13       | BranchEn         | 0        | 0: Disable COFI-based packets.  
1: Enable COFI-based packets: FUP, TIP, TIP.PGE, TIP.PGD, TNT, MODE.Exec, MODE.TSX.  
See Section 33.2.6.4 for details on BranchEn.                                                                                                                                  |
| 17:14    | MTCFreq          | 0        | Defines MTC packet Frequency, which is based on the core crystal clock, or Always Running Timer (ART). MTC will be sent each time the selected ART bit toggles. The following Encodings are defined:  
Software must use CPUID to query the supported encodings in the processor, see Section 33.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 3] = 0.                                                                 |
| 18       | Reserved         | 0        | Must be 0.                                                                                                                  |
### Table 33-6. IA32_RTIT_CTL MSR (Contd.)

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22:19</td>
<td>CycThresh</td>
<td>0</td>
<td>CYC packet threshold, see Section 33.3.6 for details. CYC packets will be sent with the first eligible packet after N cycles have passed since the last CYC packet. If CycThresh is 0 then N=0, otherwise N is defined as (2^{\text{CycThresh}-1}). The following Encodings are defined: 0: 0, 1: 1, 2: 2, 3: 4, 4: 8, 5: 16, 6: 32, 7: 64, 8: 128, 9: 256, 10: 512, 11: 1024, 12: 2048, 13: 4096, 14: 8192, 15: 16384. Software must use CPUID to query the supported encodings in the processor, see Section 33.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 1] = 0.</td>
</tr>
<tr>
<td>23</td>
<td>Reserved</td>
<td>0</td>
<td>Must be 0.</td>
</tr>
<tr>
<td>27:24</td>
<td>PSBFreq</td>
<td>0</td>
<td>Indicates the frequency of PSB packets. PSB packet frequency is based on the number of Intel PT packet bytes output, so this field allows the user to determine the increment of IA32_RTIT_STATUS.PacketByteCnt that should cause a PSB to be generated. Note that PSB insertion is not precise, but the average output bytes per PSB should approximate the SW selected period. The following Encodings are defined: 0: 2K, 1: 4K, 2: 8K, 3: 16K, 4: 32K, 5: 64K, 6: 128K, 7: 256K, 8: 512K, 9: 1M, 10: 2M, 11: 4M, 12: 8M, 13: 16M, 14: 32M, 15: 64M. Software must use CPUID to query the supported encodings in the processor, see Section 33.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 1] = 0.</td>
</tr>
<tr>
<td>30:28</td>
<td>Reserved</td>
<td>0</td>
<td>Must be 0.</td>
</tr>
<tr>
<td>31</td>
<td>EventEn</td>
<td>0</td>
<td>0: Event Trace packets are disabled. 1: Event Trace packets are enabled. This bit is reserved when CPUID.(EAX=14H, ECX=0):EBX[bit 7] (&quot;Event Trace Supported&quot;) is 0.</td>
</tr>
<tr>
<td>35:32</td>
<td>ADDRO_CFG</td>
<td>0</td>
<td>Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDRO range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 33.2.5.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3:15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] &lt; 1.</td>
</tr>
<tr>
<td>39:36</td>
<td>ADDR1_CFG</td>
<td>0</td>
<td>Configures the base/limit register pair IA32_RTIT_ADDR1_A/B based on the following encodings: 0: ADDR1 range unused. 1: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 33.2.5.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 33.4.2.10 for details on TraceStop. 3:15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] &lt; 2.</td>
</tr>
</tbody>
</table>
33.2.8.3 Enabling and Disabling Packet Generation with TraceEn

When TraceEn transitions from 0 to 1, Intel Processor Trace is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. A full PSB+ (see Section 33.4.2.17) will be generated if IA32_RTIT_STATUS.PacketByteCnt=0, and may be generated in other cases as well. Otherwise, timing packets will be generated, including TSC, TMA, and CBR (see Section 33.4.1.1).

In addition to the packets discussed above, if and when PacketEn (Section 33.2.6.1) transitions from 0 to 1 (which may happen immediately, depending on filtering settings), a TIP.PGE packet (Section 33.4.2.3) will be generated. When TraceEn is set, the processor may read ToPA entries from memory and cache them internally. For this reason, software should disable packet generation before making modifications to the ToPA tables (or changing the config-

---

Table 33-6. IA32_RTIT_CTL MSR (Contd.)

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>43:40</td>
<td>ADDR2_CFG</td>
<td>0</td>
<td>Configures the base/limit register pair IA32_RTIT_ADDR2_A/B based on the following encodings: 0: ADDR2 range unused. 1: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 33.2.5.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 33.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] &lt; 3.</td>
</tr>
<tr>
<td>47:44</td>
<td>ADDR3_CFG</td>
<td>0</td>
<td>Configures the base/limit register pair IA32_RTIT_ADDR3_A/B based on the following encodings: 0: ADDR3 range unused. 1: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 33.2.5.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 33.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] &lt; 4.</td>
</tr>
<tr>
<td>54:48</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.</td>
</tr>
<tr>
<td>55</td>
<td>DisTNT</td>
<td>0</td>
<td>0: Include TNT packets in control flow trace. 1: Omit TNT packets from control flow trace. This bit is reserved when CPUID.(EAX=14H, ECX=0):EBX[bit 8] (&quot;TNT Disable Supported&quot;) is 0. See Section 33.3.9 for details.</td>
</tr>
<tr>
<td>56</td>
<td>InjectPsbPmiOnEnable</td>
<td>0</td>
<td>1: Enables use of IA32_RTIT_STATUS bits PendPSB[6] and PendTopaPMI[7], see Section 33.2.8.4, &quot;IA32_RTIT_STATUS MSR,&quot; for behavior of these bits. 0: IA32_RTIT_STATUS bits 6 and 7 are ignored. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 6] = 0.</td>
</tr>
<tr>
<td>59:57</td>
<td>Reserved</td>
<td>0</td>
<td>Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.</td>
</tr>
<tr>
<td>63:60</td>
<td>Reserved</td>
<td>0</td>
<td>Must be 0.</td>
</tr>
</tbody>
</table>
uration of restricted memory regions). See Section 33.7 for more details of packets that may be generated with modifications to TraceEn.

### Disabling Packet Generation

Clearing TraceEn causes any packet data buffered within the logical processor to be flushed out, after which the output MSRs (IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) will have stable values. When output is directed to memory, a store, fence, or architecturally serializing instruction may be required to ensure that the packet data is globally observed. No special packets are generated by disabling packet generation, though a TIP.PGD may result if PacketEn=1 at the time of disable.

### Other Writes to IA32_RTIT_CTL

Any attempt to modify IA32_RTIT_CTL while TraceEn is set will result in a general-protection fault (#GP) unless the same write also clears TraceEn. However, writes to IA32_RTIT_CTL that do not modify any bits will not cause a #GP, even if TraceEn remains set.

### 33.2.8.4 IA32_RTIT_STATUS MSR

The IA32_RTIT_STATUS MSR is readable and writable by software, though some fields cannot be modified by software. See Table 33-7 for details. The WRMSR instruction ignores these bits in the source operand (attempts to modify these bits are ignored and do not cause WRMSR to fault).

This MSR can only be written when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). The processor does not modify the value of this MSR while TraceEn is 0 (software can modify it with WRMSR).

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FilterEn</td>
<td>0</td>
<td>This bit is written by the processor, and indicates that tracing is allowed for the current IP, see Section 33.2.6.5. Writes are ignored.</td>
</tr>
<tr>
<td>1</td>
<td>ContextEn</td>
<td>0</td>
<td>The processor sets this bit to indicate that tracing is allowed for the current context. See Section 33.2.6.3. Writes are ignored.</td>
</tr>
<tr>
<td>2</td>
<td>TriggerEn</td>
<td>0</td>
<td>The processor sets this bit to indicate that tracing is enabled. See Section 33.2.6.2. Writes are ignored.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td>0</td>
<td>Must be 0.</td>
</tr>
<tr>
<td>4</td>
<td>Error</td>
<td>0</td>
<td>The processor sets this bit to indicate that an operational error has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see “ToPA Errors” in Section 33.2.7.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.</td>
</tr>
<tr>
<td>5</td>
<td>Stopped</td>
<td>0</td>
<td>The processor sets this bit to indicate that a ToPA Stop condition has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see “ToPA STOP” in Section 33.2.7.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.</td>
</tr>
<tr>
<td>6</td>
<td>PendPSB</td>
<td>0</td>
<td>If IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1, the processor sets this bit when the threshold for a PSB+ to be inserted has been reached. The processor will clear this bit when the PSB+ has been inserted into the trace. If PendPSB = 1 and InjectPsbPmiOnEnable = 1 when IA32_RTIT_CTL.TraceEn[0] transitions from 0 to 1, a PSB+ will be inserted into the trace. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX[bit 6] = 0.</td>
</tr>
</tbody>
</table>
33.2.8.5  IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B MSRs

The role of the IA32_RTIT_ADDRn_A/B register pairs, for each n, is determined by the corresponding ADDRn_CFG fields in IA32_RTIT_CTL (see Section 33.2.8.2). The number of these register pairs is enumerated by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0].

- Processors that enumerate support for 1 range support:
  - IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
- Processors that enumerate support for 2 ranges support:
  - IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
  - IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
- Processors that enumerate support for 3 ranges support:
  - IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
  - IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
  - IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B
- Processors that enumerate support for 4 ranges support:
  - IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B
  - IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
  - IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B
  - IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B

Each register has a single 64-bit field that holds a linear address value. Writes must ensure that the address is in canonical form, otherwise a general-protection fault (#GP) fault will result.

Each MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

33.2.8.6  IA32_RTIT_CR3_MATCH MSR

The IA32_RTIT_CR3_MATCH register is compared against CR3 when IA32_RTIT_CTL.CR3Filter is 1. Bits 63:5 hold the CR3 address value to match, bits 4:0 are reserved to 0. For more details on CR3 filtering and the treatment of this register, see Section 33.2.5.2.
This MSR is accessible if CPUID.(EAX=14H, ECX=0):EBX[bit 0], "CR3 Filtering Support", is 1. This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). IA32_RTIT_CR3_MATCH[4:0] are reserved and must be 0; an attempt to set those bits using WRMSR causes a #GP.

### 33.2.8.7 IA32_RTIT_OUTPUT_BASE MSR

This MSR is used to configure the trace output destination, when output is directed to memory (IA32_RTIT_CTL.FabricEn = 0). The size of the address field is determined by the maximum physical address width (MAXPHYADDR), as reported by CPUID.80000008H:EAX[7:0].

When the ToPA output scheme is used, the processor may update this MSR when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6:0</td>
<td>Reserved</td>
<td>0</td>
<td>Must be 0.</td>
</tr>
<tr>
<td>MAXPHYADDR-1:7</td>
<td>BasePhysAddr</td>
<td>0</td>
<td>The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA: 0: This is the base physical address of a single, contiguous physical output region. This could be mapped to DRAM or to MMIO, depending on the value. The base address should be aligned with the size of the region, such that none of the 1s in the mask value(Section 33.2.8.8) overlap with 1s in the base address. If the base is not aligned, an operational error will result (see Section 33.3.10). 1: The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See “ToPA Errors” in Section 33.2.7.2, as well as Section 33.3.10.</td>
</tr>
<tr>
<td>63:MAXPHYADDR</td>
<td>Reserved</td>
<td>0</td>
<td>Must be 0.</td>
</tr>
</tbody>
</table>

### 33.2.8.8 IA32_RTIT_OUTPUT_MASK_PTRS MSR

This MSR holds any mask or pointer values needed to indicate where the next byte of trace output should be written. The meaning of the values held in this MSR depend on whether the ToPA output mechanism is in use. See Section 33.2.7.2 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).
33.2.9 Interaction of Intel® Processor Trace and Other Processor Features

33.2.9.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on a TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

If BranchEn=1, this MODE packet will be sent each time the transaction status changes. See Table 33-10 for details.

### Table 33-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR

<table>
<thead>
<tr>
<th>Position</th>
<th>Bit Name</th>
<th>At Reset</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6:0</td>
<td>LowerMask</td>
<td>7FH</td>
<td>Forced to 1, writes are ignored.</td>
</tr>
<tr>
<td>31:7</td>
<td>MaskOrTableOffset</td>
<td>0</td>
<td>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: This field holds bits 31:7 of the mask value for the single, contiguous physical output region. This value (combined with the lower 7 bits, which are reserved to 1) will be ANDed with the OutputOffset field to determine the next write address. All 1s in this field should be consecutive and starting at bit 7, otherwise the region will not be contiguous, and an operational error (Section 33.3.10) will be signaled when TraceEn is set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.</td>
</tr>
<tr>
<td>63:32</td>
<td>OutputOffset</td>
<td>0</td>
<td>The use of this field depends on the value of IA32_RTIT_CTL.ToPA:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: This is bits 31:0 of the offset pointer into the single, contiguous physical output region. This value will be added to the IA32_RTIT_OUTPUT_BASE value to form the physical address at which the next byte of packet output data will be written. This value must be less than or equal to the MaskOrTableOffset field, otherwise an operational error (Section 33.3.10) will be signaled when TraceEn is set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 33.3.10) will be signaled when TraceEn is set.</td>
</tr>
</tbody>
</table>

### Table 33-10. TSX Packet Scenarios with BranchEn=1

<table>
<thead>
<tr>
<th>TSX Event</th>
<th>Instruction</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Begin</td>
<td>Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)</td>
<td>MODE(TXAbort=0, InTX=1), FUP(CurrentIP)</td>
</tr>
<tr>
<td>Transaction Commit</td>
<td>Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit</td>
<td>MODE(TXAbort=0, InTX=0), FUP(CurrentIP)</td>
</tr>
</tbody>
</table>
The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.

### 33.2.9.2 TSX and IP Filtering

A complication with tracking transactions is handling transactions that start or end outside of the tracing region. Transactions can't span across a change in ContextEn, because CPL changes and CR3 changes each cause aborts. But a transaction can start within the IP filter region and end outside it.

To assist the decoder handling this situation, MODE.TSX packets can be sent even if FilterEn=0, though there will be no FUP attached. Instead, they will merely serve to indicate to the decoder when transactions are active and when they are not. When tracing resumes (due to PacketEn=1), the last MODE.TSX preceding the TIP.PGE will indicate the current transaction status.

### 33.2.9.3 System Management Mode (SMM)

SMM code has special privileges that non-SMM code does not have. Intel Processor Trace can be used to trace SMM code, but special care is taken to ensure that SMM handler context is not exposed in any non-SMM trace collection. Additionally, packet output from tracing non-SMM code cannot be written into memory space that is either protected by SMRR or used by the SMM handler.

SMM is entered via a system management interrupt (SMI). SMI delivery saves the value of IA32_RTIT_CTL.TraceEn into SMRAM and then clears it, thereby disabling packet generation.

The saving and clearing of IA32_RTIT_CTL.TraceEn ensures two things:

1. All internally buffered packet data is flushed before entering SMM (see Section 33.2.8.2).
2. Packet generation ceases before entering SMM, so any tracing that was configured outside SMM does not continue into SMM. No SMM instruction pointers or other state will be exposed in the non-SMM trace.

When the RSM instruction is executed to return from SMM, the TraceEn value that was saved by SMI delivery is restored, allowing tracing to be resumed. As is done any time packet generation is enabled, ContextEn is re-evaluated, based on the values of CPL, CR3, etc., established by RSM.

Like other interrupts, delivery of an SMI produces a FUP containing the IP of the next instruction to execute. By toggling TraceEn, SMI and RSM can produce TIP.PGD and TIP.PGE packets, respectively, indicating that tracing was disabled or re-enabled. See Table 33.7 for more information about packets entering and leaving SMM.

Although #SMI and RSM change CR3, PIP packets are not generated in these cases. With #SMI tracing is disabled before the CR3 change; with RSM TraceEn is restored after CR3 is written.

TraceEn must be cleared before executing RSM, otherwise it will cause a shutdown. Further, on processors that restrict use of Intel PT with LBRs (see Section 33.3.1.2), any RSM that results in enabling of both will cause a shutdown.

Intel PT can support tracing of System Transfer Monitor operating in SMM, see Section 33.6.

### Table 33-10. TSX Packet Scenarios with BranchEn=1

<table>
<thead>
<tr>
<th>TSX Event</th>
<th>Instruction</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Abort</td>
<td>XABORT or other transactional abort</td>
<td>MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)</td>
</tr>
<tr>
<td>Other</td>
<td>One of the following:</td>
<td>None. No change to TSX mode bits for these cases.</td>
</tr>
<tr>
<td></td>
<td>• Nested XBEGIN or XACQUIRE lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• An outer XACQUIRE lock that doesn't begin a transaction (InTX not set)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Non-outermost XEND or XRELEASE lock</td>
<td></td>
</tr>
</tbody>
</table>

The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.
33.2.9.4 Virtual-Machine Extensions (VMX)

Initial implementations of Intel Processor Trace do not support tracing in VMX operation. Such processors indicate this by returning 0 for IA32_VMX_MISC[bit 14]. On these processors, execution of the VMXON instruction clears IA32_RTIT_CTL.TraceEn and any attempt to write IA32_RTIT_CTL in VMX operation causes a general-protection exception (#GP).

Processors that support Intel Processor Trace in VMX operation return 1 for IA32_VMX_MISC[bit 14]. Details of tracing in VMX operation are described in Section 33.4.2.26.

33.2.9.5 Intel® Software Guard Extensions (Intel® SGX)

Intel SGX provides an application with the ability to instantiate a protective container (an enclave) with confidentiality and integrity (see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D). On a processor with both Intel PT and Intel SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. An enclave entry will clear ContextEn, thereby blocking control flow packet generation. A TIP.PGD packet will be generated if PacketEn=1 at the time of the entry.

Upon enclave exit, ContextEn will no longer be forced to 0. If other enables are set at the time, a TIP.PGE may be generated to indicate that tracing is resumed.

During the enclave execution, Intel PT remains enabled, and periodic or timing packets such as PSB, TSC, MTC, or CBR can still be generated. No IPs or other architectural state will be exposed.

For packet generation examples on enclave entry or exit, see Section 33.7.

Debug Enclaves

Intel SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by an enclave entry or exit. Hence, the generation of ContextEn-dependent packets within a debug enclave is allowed.

33.2.9.6 SENTER/ENTERACCS and ACM

GETSEC[SENTER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to flush internally buffered packets. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

33.2.9.7 Intel® Memory Protection Extensions (Intel® MPX)

Bounds exceptions (#BR) caused by Intel MPX are treated like other exceptions, producing FUP and TIP packets that indicate the source and destination IPs.

33.3 CONFIGURATION AND PROGRAMMING GUIDELINE

33.3.1 Detection of Intel Processor Trace and Capability Enumeration

Processor support for Intel Processor Trace is indicated by CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. CPUID function 14H is dedicated to enumerate the resource and capability of processors that report CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. Different processor generations may have architecturally-defined variation in capabilities. Table 33-11 describes details of the enumerable capabilities that software must use across generations of processors that support Intel Processor Trace.
<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Name</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>31:0</td>
<td>Maximum valid sub-leaf Index</td>
<td>Specifies the index of the maximum valid sub-leaf for this CPUID leaf.</td>
</tr>
</tbody>
</table>
| EBX      | 0    | CR3 Filtering Support | 1: Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CTL.CR3_MATCH MSR can be accessed. See Section 33.2.8.  
          |       |                  | 0: Indicates that writes that set IA32_RTIT_CTL.CR3Filter to 1, or any access to IA32_RTIT_CTL.CR3_MATCH, will generate a #GP exception. |
|          | 1    | Configurable PSB and Cycle-Accurate Mode Supported | 1: (a) IA32_RTIT_CTL.PSBFreq can be set to a non-zero value, in order to select the preferred PSB frequency (see below for allowed values). (b) IA32_RTIT_STATUS.PacketByteCnt can be set to a non-zero value, and will be incremented by the processor when tracing to indicate progress towards the next PSB. If trace packet generation is enabled by setting TraceEn, a PSB will only be generated if PacketByteCnt=0. (c) IA32_RTIT_CTL.CYCEn can be set to 1 to enable Cycle-Accurate Mode. See Section 33.2.8.  
          |       |                  | 0: (a) Any attempt to write a non-zero value to IA32_RTIT_CTL.PSBFreq or IA32_RTIT_STATUS.PacketByteCnt will generate a #GP exception. (b) If trace packet generation is enabled by setting TraceEn, a PSB is always generated. (c) Any attempt to write a non-zero value to IA32_RTIT_CTL.CYCEn or IA32_RTIT_CTL.CycThresh will generate a #GP exception. |
|          | 2    | IP Filtering and TraceStop supported, and Preserve Intel PT MSRs across warm reset | 1: (a) IA32_RTIT_CTL provides at one or more ADDRn_CFG field to configure the corresponding address range MSRs for IP Filtering or IP TraceStop. Each ADDRn_CFG field accepts a value in the range of 0:2 inclusive. The number of ADDRn_CFG fields is reported by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0]. (b) At least one register pair IA32_RTIT_ADDRx_A and IA32_RTIT_ADDRx_B are provided to configure address ranges for IP filtering or IP TraceStop. (c) On warm reset, all Intel PT MSRs will retain their pre-reset values, though IA32_RTIT_CTL.TraceEn will be cleared. The Intel PT MSRs are listed in Section 33.2.8.  
          |       |                  | 0: (a) An Attempt to write IA32_RTIT_CTL.ADDRx_CFG with non-zero encoding values will cause #GP. (b) Any access to IA32_RTIT_ADDRx_A and IA32_RTIT_ADDRx_B will generate a #GP exception. (c) On warm reset, all Intel PT MSRs will be cleared. |
|          | 3    | MTC Supported | 1: IA32_RTIT_CTL.MTCEn can be set to 1, and MTC packets will be generated. See Section 33.2.8.  
          |       |                  | 0: An attempt to set IA32_RTIT_CTL.MTCEn or IA32_RTIT_CTL.MTCFREQ to a non-zero value will generate a #GP exception. |
|          | 4    | PTWRITE Supported | 1: Writes can set IA32_RTIT_CTL[12](PTWEn) and IA32_RTIT_CTL[5](FUPonPTW), and PTWRITE can generate packets.  
          |       |                  | 0: Writes that set IA32_RTIT_CTL[12] or IA32_RTIT_CTL[5] will generate a #GP exception, and PTWRITE will #UD fault. |
|          | 5    | Power Event Trace Supported | 1: Writes can set IA32_RTIT_CTL[4](PwrEvtEn), enabling Power Event Trace packet generation.  
<pre><code>      |       |                  | 0: Writes that set IA32_RTIT_CTL[4] will generate a #GP exception. |
</code></pre>
<table>
<thead>
<tr>
<th>CPUID.(EAX=14H, ECX=0)</th>
<th>Name</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>PSB and PMI Preservation Supported</td>
<td>1: Writes can set IA32_RTIT_CTL[56] (InjectPsbPmiOnEnable), enabling the processor to set IA32_RTIT_STATUS[7] (PendTopaPmi) and/or IA32_RTIT_STATUS[6] (PendPSB) in order to preserve ToPA PMIs and/or PSBs otherwise lost due to Intel PT disable. Writes can also set PendToPAPMI and PendPSB. 0: Writes that set IA32_RTIT_CTL[56], IA32_RTIT_STATUS[7], or IA32_RTIT_STATUS[6] will generate a #GP exception.</td>
</tr>
<tr>
<td>7</td>
<td>Event Trace Supported</td>
<td>1: Writes can set IA32_RTIT_CTL[31] (EventEn), enabling Event Trace packet generation. 0: Writes that set IA32_RTIT_CTL[31] will generate a #GP exception.</td>
</tr>
<tr>
<td>8</td>
<td>TNT Disable Supported</td>
<td>1: Writes can set IA32_RTIT_CTL[55] (DisTNT), disabling TNT packet generation. 0: Writes that set IA32_RTIT_CTL[55] will generate a #GP exception.</td>
</tr>
<tr>
<td>ECX</td>
<td>ToPA Output Supported</td>
<td>1: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme (Section 33.2.7.2) IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.SNGLRNGOUT[bit 2] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will generate a #GP exception.</td>
</tr>
<tr>
<td>1</td>
<td>ToPA Tables Allow Multiple Output Entries</td>
<td>1: ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 33.2.7.2. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see “ToPA Errors” in Section 33.2.7.2).</td>
</tr>
<tr>
<td>2</td>
<td>Single-Range Output Supported</td>
<td>1: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=0 is supported. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.TOPAOUT[bit 0] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will generate a #GP exception.</td>
</tr>
<tr>
<td>3</td>
<td>Output to Trace Transport Subsystem Supported</td>
<td>1: Setting IA32_RTIT_CTL.FabricEn to 1 is supported. 0: IA32_RTIT_CTL.FabricEn is reserved. Write 1 to IA32_RTIT_CTL.FabricEn will generate a #GP exception.</td>
</tr>
<tr>
<td>31</td>
<td>IP Payloads are LIP</td>
<td>1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.</td>
</tr>
</tbody>
</table>

If CPUID.(EAX=14H, ECX=0):EAX reports a non-zero value, additional capabilities of Intel Processor Trace are described in the sub-leaves of CPUID leaf 14H.
<table>
<thead>
<tr>
<th>CPUID.(EAX=14H, ECX=1)</th>
<th>Name</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EAX 2:0</strong></td>
<td>Number of Address Ranges</td>
<td>A non-zero value specifies the number ADDRn_CFG field supported in IA32_RTIT_CTL and the number of register pair IA32_RTIT_ADDRn_A/IA32_RTIT_ADDRn_B supported for IP filtering and IP TraceStop. <strong>NOTE:</strong> Currently, no processors support more than 4 address ranges.</td>
</tr>
<tr>
<td><strong>15:3</strong></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td><strong>31:16</strong></td>
<td>Bitmap of supported MTC Period Encodings</td>
<td>The non-zero bits indicate the map of supported encoding values for the IA32_RTIT_CTL.MTCFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX[bit 3] = 1 (MTC Packet generation is supported), otherwise the MTCFreq field is reserved to 0. Each bit position in this field represents 1 encoding value in the 4-bit MTCFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: MTCFreq can be assigned the associated encoding value. 0: MTCFreq cannot be assigned to the associated encoding value. A write to IA32_RTIT_CTL.MTCFreq with unsupported encoding will cause #GP fault.</td>
</tr>
<tr>
<td><strong>EBX 15:0</strong></td>
<td>Bitmap of supported Cycle Threshold values</td>
<td>The non-zero bits indicate the map of supported encoding values for the IA32_RTIT_CTL.CycThresh field. This applies only if CPUID.(EAX=14H, ECX=0):EBX[bit 1] = 1 (Cycle-Accurate Mode is supported), otherwise the CycThresh field is reserved to 0. See Section 33.2.8. Each bit position in this field represents 1 encoding value in the 4-bit CycThresh field (ie, bit 0 is associated with encoding value 0). For each bit: 1: CycThresh can be assigned the associated encoding value. 0: CycThresh cannot be assigned to the associated encoding value. A write to CycThresh with unsupported encoding will cause #GP fault.</td>
</tr>
<tr>
<td><strong>31:16</strong></td>
<td>Bitmap of supported Configurable PSB Frequency encoding</td>
<td>The non-zero bits indicate the map of supported encoding values for the IA32_RTIT_CTL.PSBFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX[bit 1] = 1 (Configurable PSB is supported), otherwise the PSBFreq field is reserved to 0. See Section 33.2.8. Each bit position in this field represents 1 encoding value in the 4-bit PSBFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: PSBFreq can be assigned the associated encoding value. 0: PSBFreq cannot be assigned to the associated encoding value. A write to PSBFreq with unsupported encoding will cause #GP fault.</td>
</tr>
<tr>
<td><strong>ECX 31:0</strong></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td><strong>EDX 31:0</strong></td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
33.3.1.1 Packet Decoding of RIP versus LIP

FUP, TIP, TIP.PGE, and TIP.PGE packets can contain an instruction pointer (IP) payload. On some processor generations, this payload will be an effective address (RIP), while on others this will be a linear address (LIP). In the former case, the payload is the offset from the current CS base address, while in the latter it is the sum of the offset and the CS base address (Note that in real mode, the CS base address is the value of CS<<4, while in protected mode the CS base address is the base linear address of the segment indicated by the CS register.). Which IP type is in use is indicated by enumeration (see CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31] in Table 33-11).

For software that executes while the CS base address is 0 (including all software executing in 64-bit mode), the difference is indistinguishable. A trace decoder must account for cases where the CS base address is not 0 and the resolved LIP will not be evident in a trace generated on a CPU that enumerates use of RIP. This is likely to cause problems when attempting to link the trace with the associated binaries.

Note that IP comparison logic, for IP filtering and TraceStop range calculation, is based on the same IP type as these IP packets. For processors that output RIP, the IP comparison mechanism is also based on RIP, and hence on those processors RIP values should be written to IA32_RTIT_ADDRn_[AB] MSRs. This can produce differing behavior if the same trace configuration setting is run on processors reporting different IP types, i.e., CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31]. Care should be taken to check CPUID when configuring IP filters.

33.3.1.2 Model Specific Capability Restrictions

Some processor generations impose restrictions that prevent use of LBRs/BTS/BTM/LERs when software has enabled tracing with Intel Processor Trace. On these processors, when TraceEn is set, updates of LBR, BTS, BTM, LERs are suspended but the states of the corresponding IA32_DEBUGCTL control fields remained unchanged as if it were still enabled. When TraceEn is cleared, the LBR array is reset, and LBR/BTS/BTM/LERs updates will resume. Further, reads of these registers will return 0, and writes will be dropped.

The list of MSRs whose updates/accesses are restricted follows.
- MSR_LASTBRANCH_x_TO_IP, MSR_LASTBRANCH_x_FROM_IP, MSR_LBR_INFO_x, MSR_LASTBRANCH_TOS
- MSR_LER_FROM_LIP, MSR_LER_TO_LIP
- MSR_LBR_SELECT

For processors with CPUID DisplayFamily_DisplayModel signatures of 06_3DH, 06_47H, 06_4EH, 06_4FH, 06_56H, and 06_5EH, the use of Intel PT and LBRs are mutually exclusive.

33.3.2 Enabling and Configuration of Trace Packet Generation

To configure trace packets, enable packet generation, and capture packets, software starts with using CPUID instruction to detect its feature flag, CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1; followed by enumerating the capabilities described in Section 33.3.1.

Based on the capability queried from Section 33.3.1, software must configure a number of model-specific registers. This section describes programming considerations related to those MSRs.

33.3.2.1 Enabling Packet Generation

When configuring and enabling packet generation, the IA32_RTIT_CTL MSR should be written after any other Intel PT MSRs have been written, since writes to the other configuration MSRs cause a general-protection fault (#GP) if TraceEn = 1. If a prior trace collection context is not being restored, then software should first clear IA32_RTIT_STATUS. This is important since the Stopped, and Error fields are writable; clearing the MSR clears any values that may have persisted from prior trace packet collection contexts. See Section 33.2.8.2 for details of packets generated by setting TraceEn to 1.

If setting TraceEn to 1 causes an operational error (see Section 33.3.10), there may be a delay after the WRMSR completes before the error is signaled in the IA32_RTIT_STATUS MSR.

While packet generation is enabled, the values of some configuration MSRs (e.g., IA32_RTIT_STATUS and IA32_RTIT_OUTPUT_*) are transient, and reads may return values that are out of date. Only after packet generation is disabled (by clearing TraceEn) do reads of these MSRs return reliable values.
33.3.2.2 Disabling Packet Generation

After disabling packet generation by clearing IA32_RTIT_CTL, it is advisable to read the IA32_RTIT_STATUS MSR (Section 33.2.8.4):

- If the Error bit is set, an operational error was encountered, and the trace is most likely compromised. Software should check the source of the error (by examining the output MSR values), correct the source of the problem, and then attempt to gather the trace again. For details on operational errors, see Section 33.3.10. Software should clear IA32_RTIT_STATUS.Error before re-enabling packet generation.
- If the Stopped bit is set, software execution encountered an IP TraceStop (see Section 33.2.5.3) or the ToPA Stop condition (see “ToPA STOP” in Section 33.2.7.2) before packet generation was disabled.

33.3.3 Flushing Trace Output

Packets are first buffered internally and then written out asynchronously. To collect packet output for post-processing, a collector needs first to ensure that all packet data has been flushed from internal buffers. Software can ensure this by stopping packet generation by clearing IA32_RTIT_CTL.TraceEn (see “Disabling Packet Generation” in Section 33.2.8.2). When software clears IA32_RTIT_CTL.TraceEn to flush out internally buffered packets, the logical processor issues an SFENCE operation which ensures that WC trace output stores will be ordered with respect to the next store, or serializing operation. A subsequent read from the same logical processor will see the flushed trace data, while a read from another logical processor should be preceded by a store, fence, or architecturally serializing operation on the tracing logical processor.

When the flush operations complete, the IA32_RTIT_OUTPUT_* MSR values indicate where the trace ended. While TraceEn is set, these MSRs may hold stale values. Further, if a ToPA region with INT=1 is filled, meaning a ToPA PMI has been triggered, IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI[55] will be set by the time the flush completes.

33.3.4 Warm Reset

The MSRs software uses to program Intel Processor Trace are cleared after a power-on RESET (or cold RESET). On a warm RESET, the contents of those MSRs can retain their values from before the warm RESET with the exception that IA32_RTIT_CTL.TraceEn will be cleared (which may have the side effect of clearing some bits in IA32_RTIT_STATUS).

33.3.5 Context Switch Consideration

To facilitate construction of instruction execution traces at the granularity of a software process or thread context, software can save and restore the states of the trace configuration MSRs across the process or thread context switch boundary. The principle is the same as saving and restoring the typical architectural processor states across context switches.

33.3.5.1 Manual Trace Configuration Context Switch

The configuration can be saved and restored through a sequence of instructions of RDMSR, management of MSR content and WRMSR. To stop tracing and to ensure that all configuration MSRs contain stable values, software must clear IA32_RTIT_CTL.TraceEn before reading any other trace configuration MSRs. The recommended method for saving trace configuration context manually follows:

1. RDMSR IA32_RTIT_CTL, save value to memory
2. WRMSR IA32_RTIT_CTL with saved value from RDMSR above and TraceEn cleared
3. RDMSR all other configuration MSRs whose values had changed from previous saved value, save changed values to memory
When restoring the trace configuration context, IA32_RTIT_CTL should be restored last:

1. Read saved configuration MSR values, aside from IA32_RTIT_CTL, from memory, and restore them with WRMSR
2. Read saved IA32_RTIT_CTL value from memory, and restore with WRMSR.

33.3.5.2 Trace Configuration Context Switch Using XSAVES/XRSTORS

On processors whose XSAVE feature set supports XSAVES and XRSTORS, the Trace configuration state can be saved using XSAVES and restored by XRSTORS, in conjunction with the bit field associated with supervisory state component in IA32_XSS. See Chapter 13, “Managing State Using the XSAVE Feature Set,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

33.3.6 Cycle-Accurate Mode

Intel PT can be run in a cycle-accurate mode which enables CYC packets (see Section 33.4.2.14) that provide low-level information in the processor core clock domain. This cycle counter data in CYC packets can be used to compute IPC (Instructions Per Cycle), or to track wall-clock time on a fine-grain level.

To enable cycle-accurate mode packet generation, software should set IA32_RTIT_CTL.CYCEn=1. It is recommended that software also set TSCEn=1 anytime cycle-accurate mode is in use. With this, all CYC-eligible packets will be preceded by a CYC packet, the payload of which indicates the number of core clock cycles since the last CYC packet. In cases where multiple CYC-eligible packets are generated in a single cycle, only a single CYC will be generated before the CYC-eligible packets, otherwise each CYC-eligible packet will be preceded by its own CYC. The CYC-eligible packets are:

- TNT, TIP, TIP.PGE, TIP.PGD, MODE.EXEC, MODE.TSX, PIP, VMCS, OVF, MTC, TSC, PTWRITE, EXSTOP

TSC packets are generated when there is insufficient information to reconstruct wall-clock time, due to tracing being disabled (TriggerEn=0), or power down scenarios like a transition to a deep-sleep MWAIT C-state. In this case, the CYC that is generated along with the TSC will indicate the number of cycles actively tracing (those powered up, with TriggerEn=1) executed between the last CYC packet and the TSC packet. And hence the amount of time spent while tracing is inactive can be inferred from the difference in time between that expected based on the CYC value, and the actual time indicated by the TSC.

Additional CYC packets may be sent stand-alone, so that the processor can ensure that the decoder is aware of the number of cycles that have passed before the internal hardware counter wraps, or is reset due to other micro-architectural condition. There is no guarantee at what intervals these standalone CYC packets will be sent, except that they will be sent before the wrap occurs. An illustration is given below.

Example 33-1. An Illustrative CYC Packet Example

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>Instruction Snapshot</th>
<th>Generated Packets</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>call %eax</td>
<td>CYC(?) , TIP</td>
<td>?Elapsed cycles from the previous CYC unknown</td>
</tr>
<tr>
<td>x + 2</td>
<td>call %ebx</td>
<td>CYC(2), TIP</td>
<td>1 byte CYC packet; 2 cycles elapsed from the previous CYC</td>
</tr>
<tr>
<td>x + 8</td>
<td>jnz Foo (not taken)</td>
<td>CYC(6)</td>
<td>1 byte CYC packet</td>
</tr>
<tr>
<td>x + 9</td>
<td>ret (compressed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x + 12</td>
<td>jnz Bar (taken)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x + 16</td>
<td>ret (uncompressed)</td>
<td>TNT, CYC(8), TIP</td>
<td>1 byte CYC packet</td>
</tr>
<tr>
<td>x + 4111</td>
<td></td>
<td>CYC(4095)</td>
<td>2 byte CYC packet</td>
</tr>
<tr>
<td>x + 12305</td>
<td></td>
<td>CYC(8194)</td>
<td>3 byte CYC packet</td>
</tr>
<tr>
<td>x + 16332</td>
<td>mov cr3, %ebx</td>
<td>CYC(4027), PIP</td>
<td>2 byte CYC packet</td>
</tr>
</tbody>
</table>
33.3.6.1 Cycle Counter
The cycle counter is implemented in hardware (independent of the time stamp counter or performance monitoring counters), and is a simple incrementing counter that does not saturate, but rather wraps. The size of the counter is implementation specific.

The cycle counter is reset to zero any time that TriggerEn is cleared, and when a CYC packet is sent. The cycle counter will continue to count when ContextEn or FilterEn are cleared, and cycle packets will still be generated. It will not count during sleep states that result in Intel PT logic being powered-down, but will count up to the point where clocks are disabled, and resume counting once they are re-enabled.

33.3.6.2 Cycle Packet Semantics
Cycle-accurate mode adheres to the following protocol:

- All packets that precede a CYC packet represent instructions or events that took place before the CYC time.
- All packets that follow a CYC packet represent instructions or events that took place at the same time as, or after, the CYC time.
- The CYC-eligible packet that immediately follows a CYC packet represents an instruction or event that took place at the same time as the CYC time.

These items above give the decoder a means to apply CYC packets to a specific instruction in the assembly stream. Most packets represent a single instruction or event, and hence the CYC packet that precedes each of those packets represents the retirement time of that instruction or event. In the case of TNT packets, up to 6 conditional branches and/or compressed RETs may be contained in the packet. In this case, the preceding CYC packet provides the retirement time of the first branch in the packet. It is possible that multiple branches retired in the same cycle as that first branch in the TNT, but the protocol will not make that obvious. Also note that a MTC packet could be generated in the same cycle as the first JCC in the TNT packet. In this case, the CYC would precede both the MTC and the TNT, and apply to both.

Note that there are times when the cycle counter will stop counting, though cycle-accurate mode is enabled. After any such scenario, a CYC packet followed by TSC packet will be sent. See Section 33.8.3.2 to understand how to interpret the payload values.

Multi-packet Instructions or Events
Some operations, such as interrupts or task switches, generate multiple packets. In these cases, multiple CYC packets may be sent for the operation, preceding each CYC-eligible packet in the operation. An example, using a task switch on a software interrupt, is shown below.

Example 33-2. An Example of CYC in the Presence of Multi-Packet Operations

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>Instruction Snapshot</th>
<th>Generated Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>jnz Foo (not taken)</td>
<td>CYC(?)</td>
</tr>
<tr>
<td>x + 2</td>
<td>ret (compressed)</td>
<td></td>
</tr>
<tr>
<td>x + 8</td>
<td>jnz Bar (taken)</td>
<td></td>
</tr>
<tr>
<td>x + 9</td>
<td>jmp %eax</td>
<td>TNT, CYC(9), TIP</td>
</tr>
<tr>
<td>x + 12</td>
<td>jnz Bar (not taken)</td>
<td>CYC(3)</td>
</tr>
<tr>
<td>x + 32</td>
<td>int3 (task gate)</td>
<td>TNT, FUP, CYC(10), PIP, CYC(20), MODE.Exec, TIP</td>
</tr>
</tbody>
</table>

33.3.6.3 Cycle Thresholds
Software can opt to reduce the frequency of cycle packets, a trade-off to save bandwidth and intrusion at the expense of precision. This is done by utilizing a cycle threshold (see Section 33.2.8.2).

IA32_RTIT_CTL.CycThresh indicates to the processor the minimum number of cycles that must pass before the next CYC packet should be sent. If this value is 0, no threshold is used, and CYC packets can be sent every cycle in which a CYC-eligible packet is generated. If this value is greater than 0, the hardware will wait until the associated
number of cycles have passed since the last CYC packet before sending another. CPUID provides the threshold options for CycThresh, see Section 33.3.1.

Note that the cycle threshold does not dictate how frequently a CYC packet will be posted, it merely assigns the maximum frequency. If the cycle threshold is 16, a CYC packet can be posted no more frequently than every 16 cycles. However, once that threshold of 16 cycles has passed, it still requires a new CYC-eligible packet to be generated before a CYC will be inserted. Table 33-13 illustrates the threshold behavior.

<table>
<thead>
<tr>
<th>Time (cycles)</th>
<th>Instruction Snapshot</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>jmp %eax</td>
<td>CYC, TIP</td>
</tr>
<tr>
<td>x + 9</td>
<td>call %ebx</td>
<td>CYC, TIP</td>
</tr>
<tr>
<td>x + 15</td>
<td>call %ecx</td>
<td>CYC, TIP</td>
</tr>
<tr>
<td>x + 30</td>
<td>jmp %edx</td>
<td>CYC, TIP</td>
</tr>
<tr>
<td>x + 38</td>
<td>mov cr3,%eax</td>
<td>CYC, PIP</td>
</tr>
<tr>
<td>x + 46</td>
<td>jmp [%eax]</td>
<td>CYC, TIP</td>
</tr>
<tr>
<td>x + 64</td>
<td>call %edx</td>
<td>CYC, TIP</td>
</tr>
<tr>
<td>x + 71</td>
<td>jmp %edx</td>
<td>CYC, TIP</td>
</tr>
</tbody>
</table>

**33.3.7 Decoder Synchronization (PSB+)**

The PSB packet (Section 33.4.2.17) serves as a synchronization point for a trace-packet decoder. It is a pattern in the trace log for which the decoder can quickly scan to align packet boundaries. No legal packet combination can result in such a byte sequence. As such, it serves as the starting point for packet decode. To decode a trace log properly, the decoder needs more than simply to be aligned: it needs to know some state and potentially some timing information as well. The decoder should never need to retain any information (e.g., LastIP, call stack, compound packet event) across a PSB; all compound packet events will be completed before a PSB, and any compression state will be reset.

When a PSB packet is generated, it is followed by a PSBEND packet (Section 33.4.2.18). One or more packets may be generated in between those two packets, and these inform the decoder of the current state of the processor. These packets, known collectively as PSB+, should be interpreted as "status only", since they do not imply any change of state at the time of the PSB, nor are they associated directly with any instruction or event. Thus, the normal binding and ordering rules that apply to these packets outside of PSB+ can be ignored when these packets are between a PSB and PSBEND. They inform the decoder of the state of the processor at the time of the PSB.

PSB+ can include:
- Timestamp (TSC), if IA32_RTIT_CTL.TSCEn=1.
- Timestamp-MTC Align (TMA), if IA32_RTIT_CTL.TSCEn=1 && IA32_RTIT_CTL.MTCEn=1.
- Paging Information Packet (PIP), if ContextEn=1 and IA32_RTIT_CTL.OS=1. The non-root bit (NR) is set if the logical processor is in VMX non-root operation and the "conceal VMX from PT" VM-execution control is 0.
- VMCS packet, if either the logical is in VMX root operation or the logical processor is in VMX non-root operation and the "conceal VMX from PT" VM-execution control is 0.
- Core Bus Ratio (CBR).
- MODE.TSX, if ContextEn=1 and BranchEn = 1.
- MODE.Exec, if PacketEn=1.
- Flow Update Packet (FUP), if PacketEn=1.

PSB is generated only when TriggerEn=1; hence PSB+ has the same dependencies. The ordering of packets within PSB+ is not fixed. Timing packets such as CYC and MTC may be generated between PSB and PSBEND, and their meanings are the same as outside PSB+. 
A PSB+ can be lost in some scenarios. If IA32_RTIT_STATUS.TriggerEn is cleared just as the PSB threshold is reached, e.g., due to TraceEn being cleared, the PSB+ may not be generated. On processors that support PSB preservation (CPUID.(EAX=14H, ECX=0):EBX[bit 6] = 1), setting IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PSB+ that is pending at the time PT is disabled will be recorded by setting IA32_RTIT_STATUS.PendPSB[6] = 1. A PSB will be inserted, and PendPSB cleared, when PT is later re-enabled while PendPSB = 1.

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost. For this reason, the OVF packet should also be viewed as terminating PSB+. If IA32_RTIT_STATUS.TriggerEn is cleared just as the PSB threshold is reached, the PSB+ may not be generated. TriggerEn can be cleared by a WRMSR that clears IA32_RTIT_CTL.TraceEn, a VM exit that clears IA32_RTIT_CTL.TraceEn, an #SMI, or any time that either IA32_RTIT_STATUS.Stopped is set (e.g., by a TraceStop or ToPA stop condition) or IA32_RTIT_STATUS.Error is set (e.g., by an Intel PT output error). On processors that support PSB preservation (CPUID.(EAX=14H, ECX=0):EBX[bit 6] = 1), setting IA32_RTIT_CTL.InjectPsbPmiOnEnable[56] = 1 will ensure that a PSB+ that is pending at the time PT is disabled will be recorded by setting IA32_RTIT_STATUS.PendPSB[6] = 1. A PSB will then be pended when the saved PT context is later restored.

### 33.3.8 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor’s dedicated internal buffers are all full, an “internal buffer overflow” occurs. On such an overflow packet generation ceases (as packets would need to enter the processor’s internal buffer) until the overflow resolves. Once resolved, packet generation resumes.

When the buffer overflow is cleared, an OVF packet (Section 33.4.2.16) is generated, and the processor ensures that packets which follow the OVF are not compressed (IP compression or RET compression) against packets that were lost.

If IA32_RTIT_CTL.BranchEn = 1, the OVF packet will be followed by a FUP if the overflow resolves while PacketEn=1. If the overflow resolves while PacketEn = 0 no packet is generated, but a TIP.PGE will naturally be generated later, once PacketEn = 1. The payload of the FUP or TIP.PGE will be the Current IP of the first instruction upon which tracing resumes after the overflow is cleared. If the overflow resolves while PacketEn=1, only timing packets may come between the OVF and the FUP. If the overflow resolves while PacketEn=0, any other packets that are not dependent on PacketEn may come between the OVF and the TIP.PGE.

#### 33.3.8.1 Overflow Impact on Enables

The address comparisons to ADDRn ranges, for IP filtering and TraceStop (Section 33.2.5.3), continue during a buffer overflow, and TriggerEn, ContextEn, and FilterEn may change during a buffer overflow. Like other packets, however, any TIP.PGE or TIP.PGD packets that would have been generated will be lost. Further, IA32_RTIT_STATUS.PacketByteCnt will not increment, since it is only incremented when packets are generated.

If a TraceStop event occurs during the buffer overflow, IA32_RTIT_STATUS.Stopped will still be set, tracing will cease as a result. However, the TraceStop packet, and any TIP.PGD that result from the TraceStop, may be dropped.

#### 33.3.8.2 Overflow Impact on Timing Packets

Any timing packets that are generated during a buffer overflow will be dropped. If only a few MTC packets are dropped, a decoder should be able to detect this by noticing that the time value in the first MTC packet after the buffer overflow incremented by more than one. If the buffer overflow lasted long enough that 256 MTC packets are lost (and thus the MTC packet ‘wraps’ its 8-bit CTC value), then the decoder may be unable to properly understand the trace. This is not an expected scenario. No CYC packets are generated during overflow, even if the cycle counter wraps.

Note that, if cycle-accurate mode is enabled, the OVF packet will generate a CYC packet. Because the cycle counter counts during overflows, this CYC packet can provide the duration of the overflow. However, there is a risk that the cycle counter wrapped during the overflow, which could render this CYC misleading.
33.3.9 TNT Disable

Software can opt to omit TNT packets from control flow trace (BranchEn=1) by setting IA32_RTIT_CTL.DisTNT[bit 55]. This can dramatically reduce trace size. Results will vary by workload, but trace size reductions of 40-75% are typical, which will have a corresponding reduction in performance overhead and memory bandwidth consumption from Intel PT. However, omitting TNT packets means the decoder is not able to follow the full control flow trace, since conditional branch and compressed RET results won't be known. Thus, TNT Disable should be employed only for usages that do not depend on full control flow trace.

NOTE
To avoid loss of RET results with TNT Disable, software may wish to disable RET compression by setting IA32_RTIT_CTL.DisRETC[bit 11].

33.3.10 Operational Errors

Errors are detected as a result of packet output configuration problems, which can include output alignment issues, ToPA reserved bit violations, or overlapping packet output with restricted memory. See “ToPA Errors” in Section 33.2.7.2 for details on ToPA errors, and Section 33.2.7.4 for details on restricted memory errors. Operational errors are only detected and signaled when TraceEn=1.

When an operational error is detected, tracing is disabled and the error is logged. Specifically, IA32_RTIT_STATUS.Error is set, which will cause IA32_RTIT_STATUS.TriggerEn to be 0. This will disable generation of all packets. Some causes of operational errors may lead to packet bytes being dropped.

It should be noted that the timing of error detection may not be predictable. Errors are signaled when the processor encounters the problematic configuration. This could be as soon as packet generation is enabled but could also be later when the problematic entry or field needs to be used.

Once an error is signaled, software should disable packet generation by clearing TraceEn, diagnose and fix the error condition, and clear IA32_RTIT_STATUS.Error. At this point, packet generation can be re-enabled.

33.4 TRACE PACKETS AND DATA TYPES

This section details the data packets generated by Intel Processor Trace. It is useful for developers writing the interpretation code that will decode the data packets and apply it to the traced source code.

33.4.1 Packet Relationships and Ordering

This section introduces the concept of packet “binding”, which involves determining the IP in a binary disassembly at which the change indicated by a given packet applies. Some packets have the associated IP as the payload (FUP, TIP), while for others the decoder need only search for the next instance of a particular instruction (or instructions) to bind the packet (TNT). However, in many cases, the decoder will need to consider the relationship between packets, and to use this packet context to determine how to bind the packet.

Section 33.4.1.1 below provides detailed descriptions of the packets, including how packets bind to IPs in the disassembly, to other packets, or to nothing at all. Many packets listed are simple to bind, because they are generated in only a few scenarios. Those that require more consideration are typically part of “compound packet events”, such as interrupts, exceptions, and some instructions, where multiple packets are generated by a single operation (instruction or event). These compound packet events frequently begin with a FUP to indicate the source address (if it is not clear from the disassembly), and are concluded by a TIP or TIP.PGD packet that indicates the destination address (if one is provided). In this scenario, the FUP is said to be “coupled” with the TIP packet.

Other packets could be in between the coupled FUP and TIP packet. Timing packets, such as TSC, MTC, CYC, or CBR, could arrive at any time, and hence could intercede in a compound packet event. If an operation changes CR3 or the processor’s mode of execution, a state update packet (i.e., PIP or MODE) is generated. The state changes indicated by these intermediate packets should be applied at the IP of the TIP* packet. A summary of compound packet events is provided in Table 33-14; see Section 33.4.1.1 for more per-packet details and Section 33.7 for more detailed packet generation examples.
### 33.4.1.1 Packet Blocks

Packet blocks are a means to dump one or more groups of state values. Packet blocks begin with a Block Begin Packet (BBP), which indicates what type of state is held within the block. Following each BBP there may be one or more Block Item Packets (BIPs), which contain the state values. The block is terminated by either a Block End Packet (BEP) or another BBP indicating the start of a new block.

The BIP packet includes an ID value that, when combined with the Type field from the BBP that preceded it, uniquely identifies the state value held in the BIP payload. The size of each BIP packet payload is provided by the Size field in the preceding BBP packet.

Each block type can have up to 32 items defined for it. There is no guarantee, however, that each block of that type will hold all 32 items. For more details on which items to expect, see documentation on the specific block type of interest.

See the BBP packet description (Section 33.4.2.26) for details on packet block generation scenarios.

Packet blocks are entirely generated within an instruction or between instructions, which dictates the types of packets (aside from BIPs) that may be seen within a packet block. Packets that indicate control flow changes, or other indication of instruction completion, cannot be generated within a block. These are listed in the following table. Other packets, including timing packets, may occur between BBP and BEP.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Beginning</th>
<th>Middle</th>
<th>End</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional, uncompressed</td>
<td>FUP or none</td>
<td>Any combination of</td>
<td>TIP or TIP.PGD</td>
<td>FUP only for asynchronous events. Order of middle packets may vary.</td>
</tr>
<tr>
<td>control-flow transfer</td>
<td></td>
<td>PIP, VMCS, MODE.Exec, or none</td>
<td></td>
<td>PIP/VMCS/MODE only if the operation modifies the state tracked by these respective packets.</td>
</tr>
<tr>
<td>TSX Update</td>
<td>MODE.TSX, and</td>
<td>None</td>
<td>TIP, TIP.PGD, or none</td>
<td>FUP</td>
</tr>
<tr>
<td>(FUP or none)</td>
<td></td>
<td></td>
<td></td>
<td>TIP/TIP.PGD only for TSX abort cases.</td>
</tr>
<tr>
<td>Overflow</td>
<td>OVF</td>
<td>PSB, PSBEND, or none</td>
<td>FUP or TIP.PGE</td>
<td>FUP if overflow resolves while ContextEn=1, else TIP.PGE.</td>
</tr>
</tbody>
</table>

#### Decoder Implications

When a Block Begin Packet (BBP) is encountered, the decoder will need to decode some packets within the block differently from those outside a block. The Block Item Packet (BIP) header byte has the same encoding as a TNT packet outside of a block, but must be treated as a BIP header (with following payload) within one.

When an OVF packet is encountered, the decoder should treat that as a block ending condition. Packet generation will not resume within a block.
### 33.4.2 Packet Definitions

The following description of packet definitions are in tabular format. Figure 33-3 explains how to interpret them. Packet bits listed as “RSVD” are not guaranteed to be 0.

<table>
<thead>
<tr>
<th>Name</th>
<th>Packet name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Description of fields**
- **Byte Number**: (Not labeled in table)
- **Bit Number**: (Not labeled in table)
- **Header bits in Green**: (Not labeled in table)
- **Payload in White**: (Not labeled in table)

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>Depends on packet generation configuration enable controls or other bits (Section 33.2.6).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation Scenario</td>
<td>Which instructions, events, or other scenarios can cause this packet to be generated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Description of the packet, including the purpose it serves, meaning of the information or payload, etc</th>
</tr>
</thead>
</table>

| Application | How a decoder should apply this packet. It may bind to a specific instruction from the binary, or to another packet in the stream, or have other implications on decode |

**Figure 33-3. Interpreting Tabular Definition of Packet Format**
### 33.4.2.1 Taken/Not-taken (TNT) Packet

#### Table 33-16. TNT Packet Definition

<table>
<thead>
<tr>
<th>Packet Format</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short TNT 1</td>
<td>0</td>
<td>1</td>
<td>B₁</td>
<td>B₂</td>
<td>B₃</td>
<td>B₄</td>
<td>B₅</td>
<td>B₆</td>
</tr>
<tr>
<td>Long TNT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Short TNT 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Short TNT 3</td>
<td>B₄₀</td>
<td>B₄₁</td>
<td>B₄₂</td>
<td>B₄₃</td>
<td>B₄₄</td>
<td>B₄₅</td>
<td>B₄₆</td>
<td>B₄₇</td>
</tr>
<tr>
<td>Short TNT 4</td>
<td>B₃₂</td>
<td>B₃₃</td>
<td>B₃₄</td>
<td>B₃₅</td>
<td>B₃₆</td>
<td>B₃₇</td>
<td>B₃₈</td>
<td>B₃₉</td>
</tr>
<tr>
<td>Short TNT 5</td>
<td>B₂₄</td>
<td>B₂₅</td>
<td>B₂₆</td>
<td>B₂₇</td>
<td>B₂₈</td>
<td>B₂₉</td>
<td>B₃₀</td>
<td>B₃₁</td>
</tr>
<tr>
<td>Short TNT 6</td>
<td>B₁₆</td>
<td>B₁₇</td>
<td>B₁₈</td>
<td>B₁₉</td>
<td>B₂₀</td>
<td>B₂₁</td>
<td>B₂₂</td>
<td>B₂₃</td>
</tr>
<tr>
<td>Short TNT 7</td>
<td>B₈</td>
<td>B₉</td>
<td>B₁₀</td>
<td>B₁₁</td>
<td>B₁₂</td>
<td>B₁₃</td>
<td>B₁₄</td>
<td>B₁₅</td>
</tr>
</tbody>
</table>

B₁…BN represent the last N conditional branch or compressed RET (Section 33.4.2.2) results, such that B₁ is oldest and BN is youngest. The short TNT packet can contain from 1 to 6 TNT bits. The long TNT packet can contain from 1 to 47 TNT bits.

Irrespective of how many TNT bits is in a packet, the last valid TNT bit is followed by a trailing 1, or Stop bit, as shown above. If the TNT packet is not full (fewer than 6 TNT bits for the Short TNT, or fewer than 47 TNT bits for the Long TNT), the Stop bit moves up, and the trailing bits of the packet are filled with 0s. Examples of these “partial TNTs” are shown below. An implementation may choose to use long TNTs, short TNTs, or both.

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>PacketEn &amp;&amp; ~IA32_RTIT_CTL.DisTNT</th>
</tr>
</thead>
</table>
| Generation Scenario | On a conditional branch or compressed RET, if it fills the TNT. Also, partial TNTs may be generated at any time, as a result of other packets being generated, or certain micro-architectural conditions occurring, before the TNT is full.
The IP payload in a TIP, FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the mode of execution, and the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out, via any of FUP, TIP, TIP.PGE, or TIP.PGD. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the “Last IP” that was encoded in trace packets, thus the decoder will need to keep track of the “Last IP” state in

<table>
<thead>
<tr>
<th>Packet Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>PacketEn</th>
<th>Generation Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, VM exit, VM entry, TSX abort, EENTER, EEXIT, ERESUME, AEX.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides the target for some control flow transfers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload. The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.</td>
</tr>
</tbody>
</table>

NOTES:
1. EENTER, EEXIT, ERESUME, AEX would be possible only for a debug enclave.
software, to match fidelity with packets generated by hardware. “Last IP” is initialized to zero, hence if the first IP in the trace may be compressed if the upper bytes are zeroes.

The “IPBytes” field of the IP packets (FUP, TIP, TIP.PGE, TIP.PGD) serves to indicate how many bytes of payload are provided, and how the decoder should fill in any suppressed bytes. The algorithm for reconstructing the IP for a TIP/FUP packet is shown in the table below.

### Table 33-18. FUP/TIP IP Reconstruction

<table>
<thead>
<tr>
<th>IPBytes</th>
<th>Uncompressed IP Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000b</td>
<td>None, IP is out of context</td>
</tr>
<tr>
<td>001b</td>
<td>Last IP[63:16]</td>
</tr>
<tr>
<td>010b</td>
<td>Last IP[63:32]</td>
</tr>
<tr>
<td>011b</td>
<td>IP Payload[47:0]</td>
</tr>
<tr>
<td>100b</td>
<td>Last IP[63:48]</td>
</tr>
<tr>
<td>101b</td>
<td>Reserved</td>
</tr>
<tr>
<td>110b</td>
<td>IP Payload[63:0]</td>
</tr>
<tr>
<td>111b</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The processor-internal Last IP state is guaranteed to be reset to zero when a PSB is sent out. This means that the IP that follows the PSB with either be un-compressed (011b or 110b, see Table 33-18), or compressed against zero.

At times, “IPbytes” will have a value of 0. As shown above, this does not mean that the IP payload matches the full address of the last IP, but rather that the IP for this packet was suppressed. This is used for cases where the IP that applies to the packet is out of context. An example is the TIP.PGD sent on a SYSCALL, when tracing only USR code.

Indirect Transfer Compression for Returns (RET)

In addition to IP compression, TIP packets for near return (RET) instructions can also be compressed. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is unneeded, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own.

When a RET is compressed, a Taken indication is added to the TNT buffer. Because the RET generates no TIP packet, it also does not update the internal Last IP value, and thus the decoder should treat it the same way. If the RET is not compressed, it will generate a TIP packet (just like when RET compression is disabled, via IA32_RTIT_CTL.DisRETC).

A CALL/RET stack can be maintained by the decoder by doing the following:

1. Allocate space to store 64 RET targets.
2. For near CALLs, push the Next IP onto the stack. Once the stack is full, new CALLs will force the oldest entry off the end of the stack, such that only the youngest 64 entries are stored. Note that this excludes zero-length CALLs, which are direct near CALLs with displacement zero (to the next IP). These CALLs typically don’t have matching RETs.
3. For near RETs, pop the top (youngest) entry off the stack. This will be the expected target of the RET.

In cases where a RET is compressed, the RET target is guaranteed to match the expected target from 3) above. If the target is not compressed, a TIP packet will be generated with the RET target, which may differ from the expected target in some cases.
The hardware ensures that packets read by the decoder will always have seen the CALL that corresponds to any compressed RET. The processor will never compress a RET across a PSB, a buffer overflow, or scenario where PacketEn=0. This means that a RET whose corresponding CALL executed while PacketEn=0, or before the last PSB, etc., will not be compressed.

If the CALL/RET stack is manipulated or corrupted by software, and thereby causes a RET to transfer control to a target that is inconsistent with the CALL/RET stack, then the RET will not be compressed, and will produce a TIP packet. This can happen, for example, if software executes a PUSH instruction to push a target onto the stack, and a later RET uses this target.

For processors that employ deferred TIPs (Section 33.4.2.3), an uncompressed RET will not be deferred, and hence will force out any accumulated TNTs or TIPs. This serves to avoid ambiguity, and make clear to the decoder whether the near RET was compressed, and hence a bit in the in-progress TNT should be consumed, or uncompressed, in which case there will be no in-progress TNT and thus a TIP should be consumed.

Note that in the unlikely case that a RET executes in a different execution mode than the associated CALL, the decoder will need to model the same behavior with its CALL stack. For instance, if a CALL executes in 64-bit mode, a 64-bit IP value will be pushed onto the software stack. If the corresponding RET executes in 32-bit mode, then only the lower 32 target bits will be popped off of the stack, which may mean that the RET does not go to the CALL’s Next IP. This is architecturally correct behavior, and this RET could be compressed, thus the decoder should match this behavior.

33.4.2.3 Deferred TIPs

The processor may opt to defer sending out the TNT when TIPs are generated. Thus, rather than sending a partial TNT followed by a TIP, both packets will be deferred while the TNT accumulates more Jcc/RET results. Any number of TIP packets may be accumulated this way, such that only once the TNT is filled, or once another packet (e.g., FUP) is generated, the TNT will be sent, followed by all the deferred TIP packets, and finally terminated by the other packet(s) that forced out the TNT and TIP packets. Generation of many other packets (see list below) will force out the TNT and any accumulated TIP packets. This is an optional optimization in hardware to reduce the bandwidth consumption, and hence the performance impact, incurred by tracing.

<table>
<thead>
<tr>
<th>Code Flow</th>
<th>Packets, Non-Deferred TIPS</th>
<th>Packets, Deferred TIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000 cmp %rcx, 0</td>
<td>TNT(0b0), TIP(0x1308)</td>
<td></td>
</tr>
<tr>
<td>0x1004 jnz Foo // not-taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1008 jmp %rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1308 cmp %rcx, 1</td>
<td>TNT(0b010), TIP(0x1100)</td>
<td></td>
</tr>
<tr>
<td>0x130c jnz Bar // not-taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1310 cmp %rcx, 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1314 jnz Baz // taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1500 cmp %eax, 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1504 jg Exit // not-taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1508 jmp %r15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1100 cmp %rbx, 1</td>
<td>TNT(0b0), FUP(0x110c), TIP(0xcc00)</td>
<td>TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)</td>
</tr>
<tr>
<td>0x1104 jg Start // not-taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1108 add %rcx, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110c ... // an asynchronous Interrupt arrives</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INThandler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xcc00 pop %rdx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Packet Generation Enable (TIP.PGE) Packet

#### Table 33-20. TIP.PGE Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Packet Format | Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins. This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples:  
  - **TriggerEn**: This is set on software write to set IA32_RTIT_CTL.TraceEn as long as the Stopped and Error bits in IA32_RTIT_STATUS are clear. The IP payload will be the Next IP of the WRMSR.  
  - **FilterEn**: This is set when software jumps into the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn[AB], see Section 33.2.5.3. The IP payload will be the target of the branch.  
  - **ContextEn**: This is set on a CPL change, a CR3 write or any other means of changing ContextEn. The IP payload will be the Next IP of the instruction that changes context if it is not a branch, otherwise it will be the target of the branch. |

| Dependencies | Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn |

#### Application

TIP.PGE packets bind to the instruction at the IP given in the payload.
## 33.4.2.5 Packet Generation Disable (TIP.PGD) Packet

### Table 33-21. TIP.PGD Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Target IP - Packet Generation Disable (TIP.PGD) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>IPBytes</td>
</tr>
<tr>
<td>0</td>
<td>TargetIP[7:0]</td>
</tr>
<tr>
<td>1</td>
<td>TargetIP[15:8]</td>
</tr>
<tr>
<td>2</td>
<td>TargetIP[23:16]</td>
</tr>
<tr>
<td>3</td>
<td>TargetIP[31:24]</td>
</tr>
<tr>
<td>4</td>
<td>TargetIP[39:32]</td>
</tr>
<tr>
<td>5</td>
<td>TargetIP[47:40]</td>
</tr>
<tr>
<td>6</td>
<td>TargetIP[55:48]</td>
</tr>
<tr>
<td>7</td>
<td>TargetIP[63:56]</td>
</tr>
</tbody>
</table>

**Dependencies**  
PacketEn transitions to 0  
Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn= 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn. PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:

- **TriggerEn**: This is cleared on software write to clear IA32_RTIT_CTL.TraceEn, or when IA32_RTIT_STATUS.Stopped is set, or on operational error. The IP payload will be suppressed in this case, and the “IPBytes” field will have the value 0.
- **FilterEn**: This is cleared when software jumps out of the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see Section 33.2.5.3. The IP payload will depend on the type of the branch. For conditional branches, the payload is suppressed (IPBytes ≠ 0), and in this case the destination can be inferred from the disassembly. For any other type of branch, the IP payload will be the target of the branch.
- **ContextEn**: This can happen on a CPL change, a CR3 write or any other means of changing ContextEn. See Section 33.2.5.3 for details. In this case, when ContextEn is cleared, there will be no IP payload. The “IPBytes” field will have value 0.

**Generation Scenario**  
Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disabled packet generation and clears PacketEn.

**Description**  
Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNT bit will be replaced with TIP.PGD. The payload of the TIP.PGD will be the target of the branch, unless the result of the instruction causes TraceEn or ContextEn to be cleared (i.e., SYSCALL when IA32_RTIT_CTL.OS=0). In the case where a conditional branch clears FilterEn and hence PacketEn, there will be no TNT bit for this branch, replaced instead by the TIP.PGD.

**Application**  
TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce. In cases where there is an unbound FUP preceding the TIP.PGD, then a TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. The TIP.PGD may or may not have an IP payload, depending on whether the operation cleared ContextEn.

If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction.
33.4.2.6 Flow Update (FUP) Packet

Table 33-22. FUP Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Flow Update (FUP) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7  6  5  4  3  2  1  0</td>
</tr>
<tr>
<td>0</td>
<td>IPBytes</td>
</tr>
<tr>
<td>1</td>
<td>IP[7:0]</td>
</tr>
<tr>
<td>2</td>
<td>IP[15:8]</td>
</tr>
<tr>
<td>3</td>
<td>IP[23:16]</td>
</tr>
<tr>
<td>4</td>
<td>IP[31:24]</td>
</tr>
<tr>
<td>5</td>
<td>IP[39:32]</td>
</tr>
<tr>
<td>6</td>
<td>IP[47:40]</td>
</tr>
<tr>
<td>7</td>
<td>IP[55:48]</td>
</tr>
<tr>
<td>8</td>
<td>IP[63:56]</td>
</tr>
</tbody>
</table>

Dependencies: TriggerEn && ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 33.2.5, Section 33.4.2.21, and Section 33.4.2.22 for details.)

Generation Scenario: Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit, #MC, PSB+, XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, EENTER, EEXIT, ERESUME, EEE, AEX, INTO, INT1, INT3, INT n, a WRMSR that disables packet generation.

Description: Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.

Application: FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets. In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 33.4.2.8 for details. Otherwise, FUPs are part of compound packet events (see Section 33.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) packet will provide the destination IP. Other packets may be included in the compound event between the FUP and TIP.

NOTES:
1. EENTER, EEXIT, ERESUME, EEE, AEX apply only if Intel Software Guard Extensions is supported.

FUP IP Payload

Flow Update Packet gives the source address of an instruction when it is needed. In general, branch instructions do not need a FUP, because the source address is clear from the disassembly. For asynchronous events, however, the source address cannot be inferred from the source, and hence a FUP will be sent. Table 33-23 illustrates cases where FUPs are sent, and which IP can be expected in those cases.
On a canonical fault due to sequentially fetching an instruction in non-canonical space (as opposed to jumping to non-canonical space), the IP of the fault (and thus the payload of the FUP) will be a non-canonical address. This is consistent with what is pushed on the stack for such faulting cases.

If there are post-commit task switch faults, the IP value of the FUP will be the original IP when the task switch started. This is the same value as would be seen in the LBR_FROM field. But it is a different value as is saved on the stack or VMCS.

### Table 33-23. FUP Cases and IP Payload

<table>
<thead>
<tr>
<th>Event</th>
<th>Flow Update IP</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Interrupt, NMI/SMI, Traps, Machine Check (trap-like), INIT/SIPI</td>
<td>Address of next instruction (Next IP) that would have been executed</td>
<td>Functionally, this matches the LBR_FROM field value and also the EIP value which is saved onto the stack.</td>
</tr>
<tr>
<td>Exceptions/Faults, Machine check (fault-like)</td>
<td>Address of the instruction which took the exception/fault (Current IP)</td>
<td>This matches the similar functionality of LBR_FROM field value and also the EIP value which is saved onto the stack.</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>Address of the software interrupt instruction (Current IP)</td>
<td>This matches the similar functionality of LBR_FROM field value and also the EIP value which is saved onto the stack.</td>
</tr>
<tr>
<td>EENTER, EEXIT, ERESUME, Enclave Exiting Event (EEE), AEX&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Current IP of the instruction</td>
<td>This matches the LBR_FROM field value and also the EIP value which is saved onto the stack.</td>
</tr>
<tr>
<td>XACQUIRE</td>
<td>Address of the X* instruction</td>
<td></td>
</tr>
<tr>
<td>XRELEASE, XBEGIN, XEND, XABORT, other transactional abort</td>
<td>Current IP</td>
<td></td>
</tr>
<tr>
<td>#SMI</td>
<td>IP that is saved into SMRAM</td>
<td></td>
</tr>
<tr>
<td>WRMSR that clears TraceEn, PSB+</td>
<td>Current IP</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**

1. Information on EENTER, EEXIT, ERESUME, EEE, Asynchronous Enclave eXit (AEX) can be found in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3D.
### 33.4.2.7 Paging Information (PIP) Packet

#### Table 33-24. PIP Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Paging Information (PIP) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>2 CR3[11:5] or 0</td>
<td>RSVD/NR</td>
</tr>
<tr>
<td>3 CR3[19:12]</td>
<td></td>
</tr>
<tr>
<td>4 CR3[27:20]</td>
<td></td>
</tr>
<tr>
<td>5 CR3[35:28]</td>
<td></td>
</tr>
<tr>
<td>6 CR3[43:36]</td>
<td></td>
</tr>
<tr>
<td>7 CR3[51:44]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn &amp;&amp; ContextEn &amp;&amp; IA32_RTIT_CTL.OS</th>
<th>Generation Scenario</th>
<th>MOV CR3, Task switch, INIT, SIPI, PSB+, VM exit, VM entry</th>
</tr>
</thead>
</table>
| Description           | The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other paging modes (32-bit and 4-level paging\(^1\)), these bits are 0. This packet holds the CR3 address value. It will be generated on operations that modify CR3:  
  - MOV CR3 operation  
  - Task Switch  
  - INIT and SIPI  
  - VM exit, if "conceal VMX from PT" VM-exit control is 0 (see Section 33.5.1)  
  - VM entry, if "conceal VMX from PT" VM-entry control is 0  
  PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section 33.2.9.3 for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.  
  The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.  
  The PIP packet contains the new CR3 value when CR3 is written.  
  PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the "conceal VMX from PT" VM-execution control is 0 (see Section 33.5.1). All other PIPs clear the NR bit. |
| Application           | The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time. When a PIP is encountered, a decoder should do the following:  
  1) If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this PIP is part of a compound packet event (Section 33.4.1). Find the ending TIP and apply the new CR3/NR values to the TIP payload IP.  
  2) Otherwise, look for the next MOV CR3, far branch, or VMRESUME/VMLAUNCH in the disassembly, and apply the new CR3 to the next (or target) IP.  
  For examples of the packets generated by these flows, see Section 33.7. |

**NOTES:**

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

### 33.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.
The MODE Leaf ID indicates which set of mode bits are held in the lower bits.

### MODE.Exec Packet

#### Table 33-25. General Form of MODE Packets

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Leaf ID</td>
<td>Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table 33-26. MODE.Exec Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>MODE.Exec Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

MODE Leaf ID is '0000'.

**Dependencies**

- **TriggerEn**
- **ContextEn**
- **FilterEn**
- **Generation Scenario**
  - Any operation that changes the CS.L, CS.D, or EFER.LMA, if IA32_RTIT_CTL.BranchEn=1.
  - Any operation that changes RFLAGS.IF, if IA32_RTIT_CTL.EventEn=1.
  - Any TIP.PGE scenario, such that any of the mode bits tracked may have changed since the last MODE.Exec.

**Description**

Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L & IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary. Further, if CPUID.0x14.0.EBX[6]=1 ("Event Trace Support"), it indicates when interrupts are masked by providing the RFLAGS.IF value.

MODE.Exec is sent at the time of a mode change, if dependencies are met at the time, otherwise it is sent when tracing resumes. In the former case, the MODE.Exec packet is generated along with other packets that result from the operation that changes the mode, and is guaranteed to be followed by a TIP or TIP.PGE for branch operations, or a FUP for non-branch operations (CLI, STI, or POPF if EventEn=1). In cases where the mode changes while filtering dependencies are not met, the processor ensures that the decoder doesn’t lose track of the mode by sending any needed MODE.Exec once tracing resumes (preceding the TIP.PGE, if BranchEn=1). The processor may opt to suppress the MODE.Exec when tracing resumes if the mode matches that of the last MODE.Exec packet. MODE.Exec packets are generated on CS.L, CS.D, or EFER.LMA changes only if control flow tracing is enabled (BranchEn=1). This is essential for the decoder to properly disassemble the associated binary.

<table>
<thead>
<tr>
<th>CS.D</th>
<th>(CS.L &amp; IA32_EFER.LMA)</th>
<th>Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>64-bit mode</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>32-bit mode</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>16-bit mode</td>
</tr>
</tbody>
</table>

MODE.Exec packets are generated on interrupt flag (RFLAGS.IF) changes only if event tracing is enabled (EventEn=1).

**Application**

MODE.Exec always precedes an IP packet (TIP, TIP.PGE, or FUP). The mode change applies to the IP address in the payload of the IP packet. When MODE.Exec is followed by a FUP, it is a stand-alone FUP and should be consumed by the MODE.Exec.
## MODE.TSX Packet

### Table 33-27. MODE.TSX Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>MODE.TSX Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7   6   5   4   3   2   1   0</td>
</tr>
<tr>
<td>0</td>
<td>1   0   0   1   1   0   0   1</td>
</tr>
<tr>
<td>1</td>
<td>0   0   1   0   0   0   0   TXAbort InTX</td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>TriggerEn &amp;&amp; ContextEn</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Indicates when a TSX transaction (either HLE or RTM) begins, commits, or aborts. Instructions executed transactionally will be “rolled back” if the transaction is aborted.</td>
</tr>
<tr>
<td>TXAbort</td>
<td>InTX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Application**

If PacketEn=1, MODE.TSX always immediately precedes a FUP. If the TXAbort bit is zero, then the mode change applies to the IP address in the payload of the FUP. If TXAbort=1, then the FUP will be followed by a TIP, and the mode change will apply to the IP address in the payload of the TIP.

MODE.TSX packets may be generated when PacketEn=0, due to FilterEn=0. In this case, only the last MODE.TSX generated before TIP.PGE need be applied.
### 33.4.2.9 TraceStop Packet

#### Table 33-28. TraceStop Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>TraceStop Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td><img src="image" alt="Binary Format" /></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>TriggerEn &amp;&amp; ContextEn</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Indicates when software has entered a user-configured TraceStop region. When the IP matches a TraceStop range while ContextEn and TriggerEn are set, a TraceStop action occurs. This disables tracing by setting IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn, and causes a TraceStop packet to be generated. The TraceStop action also forces FilterEn to 0. Note that TraceStop may not force a flush of internally buffered packets, and thus trace packet generation should still be manually disabled by clearing IA32_RTIT_CTL.TraceEn before examining output. See Section 33.2.5.3 for more details.</td>
</tr>
<tr>
<td><strong>Application</strong></td>
<td>If TraceStop follows a TIP.PGD (before the next TIP.PGE), then it was triggered either by the instruction that cleared PacketEn, or it was triggered by some later instruction that executed while FilterEn=0. In either case, the TraceStop can be applied at the IP of the TIP.PGD (if any). If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.</td>
</tr>
</tbody>
</table>

### 33.4.2.10 Core:Bus Ratio (CBR) Packet

#### Table 33-29. CBR Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Core:Bus Ratio (CBR) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td><img src="image" alt="Binary Format" /></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>TriggerEn</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Indicates the core:bus ratio of the processor core. Useful for correlating wall-clock time and cycle time.</td>
</tr>
<tr>
<td><strong>Application</strong></td>
<td>The CBR packet indicates the point in the trace when a frequency transition has occurred. On some implementations, software execution will continue during transitions to a new frequency, while on others software execution ceases during frequency transitions. There is not a precise IP provided, to which to bind the CBR packet.</td>
</tr>
</tbody>
</table>
### 33.4.2.11 Timestamp Counter (TSC) Packet

#### Table 33-30. TSC Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Timestamp Counter (TSC) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>Dependencies</td>
<td>IA32_RTIT_CTL.TSCEn &amp;&amp; TriggerEn</td>
</tr>
<tr>
<td>Generation Scenario</td>
<td>Sent after any event that causes the processor clocks or Intel PT timing packets (such as MTC or CYC) to stop. This may include P-state changes, wake from C-state, or clock modulation. Also on transition of TraceEn from 0 to 1.</td>
</tr>
<tr>
<td>Description</td>
<td>When enabled by software, a TSC packet provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other timestamped logs.</td>
</tr>
<tr>
<td>Application</td>
<td>TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet.</td>
</tr>
</tbody>
</table>
33.4.2.12 Mini Time Counter (MTC) Packet

Table 33-31. MTC Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Mini time Counter (MTC) Packet</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Packet Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0 0 1 0 1 1 0 0 1</td>
</tr>
<tr>
<td>1 CTC[N+7:N]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>Generation Scenario</th>
<th>Periodic, based on the core crystal clock, or Always Running Timer (ART).</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_RTIT_CTL.MTCEn &amp;&amp; TriggerEn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description
When enabled by software, an MTC packet provides a periodic indication of wall-clock time. The 8-bit CTC (Common Timestamp Copy) payload value is set to (ART >> N) & FFH. The frequency of the ART is related to the Maximum Non-Turbo frequency, and the ratio can be determined from CPUID leaf 15H, as described in Section 33.8.3. Software can select the threshold N, which determines the MTC frequency by setting the IA32_RTIT_CTL.MTCFreq field (see Section 33.2.8.2) to a supported value using the lookup enumerated by CPUID (see Section 33.3.1). See Section 33.8.3 for details on how to use the MTC payload to track TSC time.

MTC provides 8 bits from the ART, starting with the bit selected by MTCFreq to dictate the frequency of the packet. Whenever that 8-bit range being watched changes, an MTC packet will be sent out with the new value of that 8-bit range. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. The decoder can infer the truncated bits, CTC[N-1:0], are 0 at the time of the MTC packet.

There are cases in which MTC packet can be dropped, due to overflow or other micro-architectural conditions. The decoder should be able to recover from such cases by checking the 8-bit payload of the next MTC packet, to determine how many MTC packets were dropped. It is not expected that >256 consecutive MTC packets should ever be dropped.

Application
MTC does not precisely indicate the time of any other packet, nor does it bind to any IP. However, all preceding packets represent instructions or events that executed before the indicated ART time, and all subsequent packets represent instructions that executed after, or at the same time as, the ART time.
### TSC/MTC Alignment (TMA) Packet

#### Table 33-32. TMA Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>TSC/MTC Alignment (TMA) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td><strong>Dependency</strong></td>
<td>IA32_RTIT_CTL.MTCEn &amp;&amp;</td>
</tr>
<tr>
<td></td>
<td>IA32_RTIT_CTL.TSCEn &amp;&amp;</td>
</tr>
<tr>
<td></td>
<td>TriggerEn</td>
</tr>
<tr>
<td><strong>Generation Scenario</strong></td>
<td>Sent with any TSC packet.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The TMA packet serves to provide the information needed to allow the decoder to correlate MTC packets with TSC packets. With this packet, when a MTC packet is encountered, the decoder can determine how many timestamp counter ticks have passed since the last TSC or MTC packet. See Section 33.8.3.2 for details on how to make this calculation.</td>
</tr>
<tr>
<td><strong>Application</strong></td>
<td>TMA is always sent immediately following a TSC packet, and the payload values are consistent with the TSC payload value. Thus the application of TMA matches that of TSC.</td>
</tr>
</tbody>
</table>
### 33.4.2.14 Cycle Count (CYC) Packet

#### Table 33-33. Cycle Count Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>Cycle Count (CYC) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Cycle Counter[4:0]</td>
<td></td>
</tr>
<tr>
<td>Exp</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Cycle Counter[11:5]</td>
</tr>
<tr>
<td>Exp</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cycle Counter[18:12]</td>
</tr>
<tr>
<td>Exp</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>... (if Exp = 1 in the previous byte)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>IA32_RTIT_CTL.CYCEn&amp;&amp;TriggerEn</th>
<th>Generation Scenario</th>
<th>Can be sent at any time, though a maximum of one CYC packet is sent per core clock cycle. See Section 33.3.6 for CYC-eligible packets.</th>
</tr>
</thead>
</table>

| Description                         | The Cycle Counter field increments at the same rate as the processor core clock ticks, but with a variable length format (using a trailing EXP bit field) and a range-capped byte length. If the CYC value is less than 32, a 1-byte CYC will be generated, with Exp=0. If the CYC value is between 32 and 4095 inclusive, a 2-byte CYC will be generated, with byte 0 Exp=1 and byte 1 Exp=0. And so on. CYC provides the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets, at the expense of some precision. These settings are configured using the IA32_RTIT_CTL.CycThresh field (see Section 33.2.8.2). For details on Cycle-Accurate Mode, IPC calculation, etc, see Section 33.3.6. When CycThresh=0, and hence no threshold is in use, then a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. The CYC packet will precede the other packets generated in the cycle, and provides the precise cycle time of the packets that follow. In addition to these CYC packets generated with other packets, CYC packets can be sent stand-alone. These packets serve simply to update the decoder with the number of cycles passed, and are used to ensure that a wrap of the processor’s internal cycle counter doesn’t cause cycle information to be lost. These stand-alone CYC packets do not indicate the cycle time of any other packet or operation, and will be followed by another CYC packet before any other CYC-eligible packet is seen. When CycThresh>0, CYC packets are generated only after a minimum number of cycles have passed since the last CYC packet. Once this threshold has passed, the behavior above resumes, where CYC will either be sent in the next cycle that produces other CYC-eligible packets, or could be sent stand-alone. When using CYC thresholds, only the cycle time of the operation (instruction or event) that generates the CYC packet is truly known. Other operations simply have their execution time bounded: they completed at or after the last CYC time, and before the next CYC time. |
| Application             | CYC provides the offset cycle time (since the last CYC packet) for the CYC-eligible packet that follows. If another CYC is encountered before the next CYC-eligible packet, the cycle values should be accumulated and applied to the next CYC-eligible packet. If a CYC packet is generated by a TNT, note that the cycle time provided by the CYC packet applies to the first branch in the TNT packet. |
33.4.2.15 VMCS Packet

Table 33-34. VMCS Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>VMCS Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td>7</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 0 1 0 0 0</td>
<td>VMCS pointer [19:12]</td>
</tr>
<tr>
<td>2</td>
<td>VMCS pointer [27:20]</td>
</tr>
<tr>
<td>3</td>
<td>VMCS pointer [35:28]</td>
</tr>
<tr>
<td>4</td>
<td>VMCS pointer [43:36]</td>
</tr>
<tr>
<td>5</td>
<td>VMCS pointer [51:44]</td>
</tr>
</tbody>
</table>

Dependencies TriggerEn && ContextEn; Also in VMX operation.

Generation Scenario Generated on successful VMPTRLD, and optionally on PSB+, SMM VM exits, and VM entries that return from SMM (see Section 33-53).

Description The VMCS packet provides a VMCS pointer for a decoder to determine the transition of code contexts:
- On a successful VMPTRLD (i.e., a VMPTRLD that doesn’t fault, fail, or VM exit), the VMCS packet contains the logical processor’s VMCS pointer established by VMPTRLD (for subsequent execution of a VM guest context).
- An SMM VM exit loads the logical processor’s VMCS pointer with the SMM-transfer VMCS pointer. If the “conceal VMX from PT” VM-exit control is 0 (see Section 33.5.1), a VMCS packet provides this pointer. See Section 33.6 on tracing inside and outside STM.
- A VM entry that returns from SMM loads the logical processor’s VMCS pointer from a field in the SMM-transfer VMCS. If the “conceal VMX from PT” VM-entry control is 0, a VMCS packet provides this pointer. Whether the VM entry is to VMX root operation or VMX non-root operation is indicated by the PIP.NR bit. A VMCS packet generated before a VMCS pointer has been loaded, or after the VMCS pointer has been cleared will set all 64 bits in the VMCS pointer field.
- VMCS packets will not be seen on processors with IA32_VMX_MISC[bit 14]=0, as these processors do not allow TraceEn to be set in VMX operation.

Application The purpose of the VMCS packet is to help the decoder uniquely identify changes in the executing software context in situations that CR3 may not be unique. When a VMCS packet is encountered, a decoder should do the following:
- If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this VMCS is part of a compound packet event (Section 33.4.1). Find the ending TIP and apply the new VMCS base pointer value to the TIP payload IP.
- Otherwise, look for the next VMPTRLD, VMRESUME, or VMLAUNCH in the disassembly, and apply the new VMCS base pointer on the next VM entry.
For examples of the packets generated by these flows, see Section 33.7.
### 33.4.2.16 Overflow (OVF) Packet

<table>
<thead>
<tr>
<th>Name</th>
<th>Overflow (OVF) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1 0 0 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn</th>
<th>Generation Scenario</th>
<th>On resolution of internal buffer overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**
OVF simply indicates to the decoder that an internal buffer overflow occurred, and packets were likely lost. If BranchEN= 1, OVF is followed by a FUP or TIP.PGE which will provide the IP at which packet generation resumes. See Section 33.3.8.

**Application**
When an OVF packet is encountered, the decoder should skip to the IP given in the subsequent FUP or TIP.PGE. The cycle counter for the CYC packet will be reset at the time the OVF packet is sent. Software should reset its call stack depth on overflow, since no RET compression is allowed across an overflow. Similarly, any IP compression that follows the OVF is guaranteed to use as a reference LastIP the IP payload of an IP packet that preceded the overflow.

### 33.4.2.17 Packet Stream Boundary (PSB) Packet

<table>
<thead>
<tr>
<th>Name</th>
<th>Packet Stream Boundary (PSB) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>2 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>3 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>4 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>5 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>6 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>7 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>8 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>9 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>10 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>11 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>12 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>13 1 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>14 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>15 1 0 0 0 0 0 1</td>
<td></td>
</tr>
</tbody>
</table>
33.4.2.18 PSBEND Packet

Table 33-37. PSBEND Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>PSBEND Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn</th>
<th>Generation Scenario</th>
<th>Always follows PSB packet, separated by PSB+ packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>PSBEND is simply a terminator for the series of “status only” (PSB+) packets that follow PSB (Section 33.3.7).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>When a PSBEND packet is seen, the decoder should cease to treat packets as “status only”.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 33.4.2.19 Maintenance (MNT) Packet

**Table 33-38. MNT Packet Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>Maintenance (MNT) Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**Dependencies** | TriggerEn | Generation Scenario | Implementation specific

**Description** | This packet is generated by hardware, the payload meaning is model-specific.

**Application** | Unless a decoder has been extended for a particular family/model/stepping to interpret MNT packet payloads, this packet should simply be ignored. It does not bind to any IP.

### 33.4.2.20 PAD Packet

**Table 33-39. PAD Packet Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>PAD Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Dependencies** | TriggerEn | Generation Scenario | Implementation specific

**Description** | PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.

**Application** | Ignore PAD packets.
### 33.4.2.21  PTWRITE (PTW) Packet

**Table 33-40. PTW Packet Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>PTW Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packet Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:

<table>
<thead>
<tr>
<th>PayloadBytes</th>
<th>Bytes of Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>'00</td>
<td>4</td>
</tr>
<tr>
<td>'01</td>
<td>8</td>
</tr>
<tr>
<td>'10</td>
<td>Reserved</td>
</tr>
<tr>
<td>'11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.

**Dependencies**

TriggerEn && ContextEn && FilterEn && PTWEn

**Scenario**

PTWRITE Instruction

**Description**

Contains the value held in the PTWRITE operand. This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.

**Application**

Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.
## 33.4.2.22 Execution Stop (EXSTOP) Packet

### Table 33-41. EXSTOP Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>EXSTOP Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>IP</td>
</tr>
<tr>
<td>IP bit indicates if a FUP will follow.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn &amp;&amp; PwrEvtEn</th>
<th>Generation Scenario</th>
<th>C-state entry, P-state change, or other processor clock power-down. Includes:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Entry to C-state deeper than C0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TM1/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• STPCLK#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Frequency change due to IA32_CLOCK_MODULATION, Turbo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>This packet indicates that software execution has stopped due to processor clock powerdown. Later packets will indicate when execution resumes.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If EXSTOP is generated while ContextEn is set, the IP bit will be set, and EXSTOP will be followed by a FUP packet containing the IP at which execution stopped. More precisely, this will be the IP of the oldest instruction that has not yet completed.</td>
</tr>
<tr>
<td></td>
<td>This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td>
</tr>
</tbody>
</table>

| Application | If a FUP follows EXSTOP (hence IP bit set), the EXSTOP can be bound to the FUP IP. Otherwise the IP is not known. Time of powerdown can be inferred from the preceding CYC, if CYCEn=1. Combined with the TSC at the time of wake (if TSCEn=1), this can be used to determine the duration of the powerdown. |
### 33.4.2.23 MWAIT Packet

<table>
<thead>
<tr>
<th>Name</th>
<th>MWAIT Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 0 0 0 0 1 0</td>
</tr>
<tr>
<td>2</td>
<td>MWAIT Hints[7:0]</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn &amp; PwrEvtEn &amp; ContextEn</th>
<th>Generation Scenario</th>
<th>MWAIT, UMWAIT, or TPAUSE instructions, or I/O redirection to MWAIT, that complete without fault or VMexit.</th>
</tr>
</thead>
</table>

**Description**

Indicates that an MWAIT operation to C-state deeper than C0.0 completed. The MWAIT hints and extensions passed in by software are exposed in the payload. For UMWAIT and TPAUSE, the EXT field holds the input register value that determines the optimized state requested.

For entry to some highly optimized C0 sub-C-states, such as C0.1, no MWAIT packet is generated.

This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.

**Application**
The binding for the upcoming EXSTOP packet also applies to the MWAIT packet. See Section 33.4.2.22.
33.4.2.24 Power Entry (PWRE) Packet

Table 33-43. PWRE Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>PWRE Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>HW</td>
</tr>
<tr>
<td>3</td>
<td>Resolved Thread C-State</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn &amp;&amp; PwrEvtEn</th>
<th>Generation Scenario</th>
<th>Transition to a C-state deeper than C0.0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Indicates processor entry to the resolved thread C-state and sub C-state indicated. The processor will remain in this C-state until either another PWRE indicates the processor has moved to a C-state deeper than C0.0, or a PWRX packet indicates a return to C0.0. For entry to some highly optimized C0 sub-C-states, such as C0.1, no PWRE packet is generated. Note that some CPUs may allow MWAIT to request a deeper C-state than is supported by the core. These deeper C-states may have platform-level implications that differentiate them. However, the PWRE packet will provide only the resolved thread C-state, which will not exceed that supported by the core. If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, UMWAIT, TPAUSE, HLT, or shutdown), the HW bit will be set to indicate this. Hardware Duty Cycling (see Section 15.5, “Hardware Duty Cycling (HDC),” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B) is an example of such a case.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>When transitioning from C0.0 to a deeper C-state, the PWRE packet will be followed by an EXSTOP. If that EXSTOP packet has the IP bit set, then the following FUP will provide the IP at which the C-state entry occurred. Subsequent PWRE packets generated before the next PWRX should bind to the same IP.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 33.4.2.25 Power Exit (PWRX) Packet

#### Table 33-44. PWRX Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>PWRX Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Last Core C-State</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>Generation Scenario</th>
<th>Transition from a C-state deeper than C0.0 to C0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriggerEn &amp;&amp; PwrEvtEn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Indicates processor return to thread C0 from a C-state deeper than C0.0.
For return from some highly optimized C0 sub-C-states, such as C0.1, no PWRX packet is generated. The Last Core C-State field provides the MWAIT encoding for the core C-state at the time of the wake. The Deepest Core C-State provides the MWAIT encoding for the deepest core C-state achieved during the sleep session, or since leaving thread C0. MWAIT encodings for C-states can be found in Table 4-11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. Note that these values reflect only the core C-state, and hence will not exceed the maximum supported core C-state, even if deeper C-states can be requested. The Wake Reason field is one-hot, encoded as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interrupt</td>
<td>Wake due to external interrupt received.</td>
</tr>
<tr>
<td>1</td>
<td>Timer Deadline</td>
<td>Wake due to timer expiration, such as UMWAIT/TPAUSE TSC-quanta.</td>
</tr>
<tr>
<td>2</td>
<td>Store to Monitored Address</td>
<td>Wake due to store to monitored address.</td>
</tr>
<tr>
<td>3</td>
<td>Hw Wake</td>
<td>Wake due to hardware autonomous condition, such as HDC.</td>
</tr>
</tbody>
</table>

**Application**
PwRX will always apply to the same IP as the PwRE. The time of wake can be discerned from (optional) timing packets that precede PwRX.
### 33.4.2.26 Block Begin Packet (BBP)

**Table 33-45. Block Begin Packet Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>BBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td><img src="image" alt="Hexadecimal Format" /></td>
</tr>
<tr>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td></td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td>0x02..0x03</td>
</tr>
<tr>
<td></td>
<td>0x04</td>
</tr>
<tr>
<td></td>
<td>0x05</td>
</tr>
<tr>
<td></td>
<td>0x06..0x07</td>
</tr>
<tr>
<td></td>
<td>0x08</td>
</tr>
<tr>
<td></td>
<td>0x09</td>
</tr>
<tr>
<td></td>
<td>0x0A</td>
</tr>
<tr>
<td></td>
<td>0x0B..0x0F</td>
</tr>
<tr>
<td></td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>0x11..0x1F</td>
</tr>
</tbody>
</table>

**Dependencies**
- TriggerEn
- Generation Scenario
- PEBS event, if IA32_PEBS_ENABLE.OUTPUT=1.

**Description**
This packet indicates the beginning of a block of packets which are collectively tied to a single event or instruction. The size of the block item payloads within this block is provided by the Size (SZ) bit:
- SZ=0: 8-byte block items
- SZ=1: 4-byte block items

The meaning of the BIP payloads is provided by the Type field:

- 0x00: Reserved
- 0x01: General-Purpose Registers
- 0x02..0x03: Reserved
- 0x04: PEBS Basic
- 0x05: PEBS Memory
- 0x06..0x07: Reserved
- 0x08: LBR Block 0
- 0x09: LBR Block 1
- 0x0A: LBR Block 2
- 0x0B..0x0F: Reserved
- 0x10: XMM Registers
- 0x11..0x1F: Reserved

**Application**
A BBP will always be followed by a Block End Packet (BEP), and when the block is generated while ContextEn=1 that BEP will have IP=1 and be followed by a FUP that provides the IP to which the block should be bound. Note that, in addition to BEP, a block can be terminated by a BBP (indicating the start of a new block) or an OVF packet.
33.4.2.27 Block Item Packet (BIP)

Table 33-46. Block Item Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>BIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the preceding BBP.SZ=0:</td>
</tr>
<tr>
<td></td>
<td>If the preceding BBP.SZ=1:</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>ID[5:0]</td>
</tr>
<tr>
<td>1</td>
<td>Payload[7:0]</td>
</tr>
<tr>
<td>2</td>
<td>Payload[15:8]</td>
</tr>
<tr>
<td>3</td>
<td>Payload[23:16]</td>
</tr>
<tr>
<td>4</td>
<td>Payload[31:24]</td>
</tr>
<tr>
<td>5</td>
<td>Payload[39:32]</td>
</tr>
<tr>
<td>6</td>
<td>Payload[47:40]</td>
</tr>
<tr>
<td>7</td>
<td>Payload[55:48]</td>
</tr>
<tr>
<td>8</td>
<td>Payload[63:56]</td>
</tr>
</tbody>
</table>

Dependencies | TriggerEn | Generation Scenario | See BBP. |
Description | The size of the BIP payload is determined by the Size field in the preceding BBP packet. The BIP header provides the ID value that, when combined with the Type field from the preceding BBP, uniquely identifies the state value held in the BIP payload. See Table 33-47 below for the complete list. |
Application | See BBP. |

BIP State Value Encodings
The table below provides the encoding values for all defined block items. State items that are larger than 8 bytes, such as XMM register values, are broken into multiple 8-byte components. BIP packets with Size=1 (4 byte payload) will provide only the lower 4 bytes of the associated state value.

Table 33-47. BIP Encodings

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>State Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x00</td>
<td>R/EFLAGS</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>R/EIP</td>
</tr>
<tr>
<td>0x01</td>
<td>0x02</td>
<td>R/EAX</td>
</tr>
<tr>
<td>0x01</td>
<td>0x03</td>
<td>R/ECX</td>
</tr>
</tbody>
</table>
### Table 33-47. BIP Encodings (Contd.)

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>State Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x04</td>
<td>R/EDX</td>
</tr>
<tr>
<td>0x01</td>
<td>0x05</td>
<td>R/EBX</td>
</tr>
<tr>
<td>0x01</td>
<td>0x06</td>
<td>R/ESP</td>
</tr>
<tr>
<td>0x01</td>
<td>0x07</td>
<td>R/EBP</td>
</tr>
<tr>
<td>0x01</td>
<td>0x08</td>
<td>R/ESI</td>
</tr>
<tr>
<td>0x01</td>
<td>0x09</td>
<td>R/EDI</td>
</tr>
<tr>
<td>0x01</td>
<td>0x0A</td>
<td>R8</td>
</tr>
<tr>
<td>0x01</td>
<td>0x0B</td>
<td>R9</td>
</tr>
<tr>
<td>0x01</td>
<td>0x0C</td>
<td>R10</td>
</tr>
<tr>
<td>0x01</td>
<td>0x0D</td>
<td>R11</td>
</tr>
<tr>
<td>0x01</td>
<td>0x0E</td>
<td>R12</td>
</tr>
<tr>
<td>0x01</td>
<td>0x0F</td>
<td>R13</td>
</tr>
<tr>
<td>0x01</td>
<td>0x10</td>
<td>R14</td>
</tr>
<tr>
<td>0x01</td>
<td>0x11</td>
<td>R15</td>
</tr>
</tbody>
</table>

#### PEBS Basic Info (Section 20.9.2.2.1)

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>0x00</td>
<td>Instruction Pointer</td>
</tr>
<tr>
<td>0x04</td>
<td>0x01</td>
<td>Applicable Counters</td>
</tr>
<tr>
<td>0x04</td>
<td>0x02</td>
<td>Timestamp</td>
</tr>
</tbody>
</table>

#### PEBS Memory Info (Section 20.9.2.2.2)

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05</td>
<td>0x00</td>
<td>MemAccessAddress</td>
</tr>
<tr>
<td>0x05</td>
<td>0x01</td>
<td>MemAuxInfo</td>
</tr>
<tr>
<td>0x05</td>
<td>0x02</td>
<td>MemAccessLatency</td>
</tr>
<tr>
<td>0x05</td>
<td>0x03</td>
<td>TSXAuxInfo</td>
</tr>
</tbody>
</table>

#### LBR_0

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>0x00</td>
<td>LBR[TOS-0]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x01</td>
<td>LBR[TOS-0]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x02</td>
<td>LBR[TOS-0]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x03</td>
<td>LBR[TOS-1]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>LBR[TOS-1]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x05</td>
<td>LBR[TOS-1]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x06</td>
<td>LBR[TOS-2]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x07</td>
<td>LBR[TOS-2]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x08</td>
<td>LBR[TOS-2]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x09</td>
<td>LBR[TOS-3]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x0A</td>
<td>LBR[TOS-3]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x0B</td>
<td>LBR[TOS-3]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x0C</td>
<td>LBR[TOS-4]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x0D</td>
<td>LBR[TOS-4]_TO_IP</td>
</tr>
<tr>
<td>BBP.Type</td>
<td>BIP.ID</td>
<td>State Value</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-------------------</td>
</tr>
<tr>
<td>0x08</td>
<td>0x0E</td>
<td>LBR[TOS-4]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x0F</td>
<td>LBR[TOS-5]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x10</td>
<td>LBR[TOS-5]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x11</td>
<td>LBR[TOS-5]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x12</td>
<td>LBR[TOS-6]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x13</td>
<td>LBR[TOS-6]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x14</td>
<td>LBR[TOS-6]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x15</td>
<td>LBR[TOS-7]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x16</td>
<td>LBR[TOS-7]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x17</td>
<td>LBR[TOS-7]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x18</td>
<td>LBR[TOS-8]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x19</td>
<td>LBR[TOS-8]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x1A</td>
<td>LBR[TOS-8]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x1B</td>
<td>LBR[TOS-9]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x1C</td>
<td>LBR[TOS-9]_TO_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x1D</td>
<td>LBR[TOS-9]_INFO</td>
</tr>
<tr>
<td>0x08</td>
<td>0x1E</td>
<td>LBR[TOS-10]_FROM_IP</td>
</tr>
<tr>
<td>0x08</td>
<td>0x1F</td>
<td>LBR[TOS-10]_TO_IP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LBR_1</td>
</tr>
<tr>
<td>0x09</td>
<td>0x00</td>
<td>LBR[TOS-10]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x01</td>
<td>LBR[TOS-11]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x02</td>
<td>LBR[TOS-11]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x03</td>
<td>LBR[TOS-11]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x04</td>
<td>LBR[TOS-12]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x05</td>
<td>LBR[TOS-12]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x06</td>
<td>LBR[TOS-12]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x07</td>
<td>LBR[TOS-13]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x08</td>
<td>LBR[TOS-13]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x09</td>
<td>LBR[TOS-13]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x0A</td>
<td>LBR[TOS-14]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x0B</td>
<td>LBR[TOS-14]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x0C</td>
<td>LBR[TOS-14]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x0D</td>
<td>LBR[TOS-15]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x0E</td>
<td>LBR[TOS-15]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x0F</td>
<td>LBR[TOS-15]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x10</td>
<td>LBR[TOS-16]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x11</td>
<td>LBR[TOS-16]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x12</td>
<td>LBR[TOS-16]_INFO</td>
</tr>
</tbody>
</table>
### Table 33-47. BIP Encodings (Contd.)

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>State Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x09</td>
<td>0x13</td>
<td>LBR[TOS-17]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x14</td>
<td>LBR[TOS-17]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x15</td>
<td>LBR[TOS-17]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x16</td>
<td>LBR[TOS-18]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x17</td>
<td>LBR[TOS-18]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x18</td>
<td>LBR[TOS-18]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x19</td>
<td>LBR[TOS-19]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x1A</td>
<td>LBR[TOS-19]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x1B</td>
<td>LBR[TOS-19]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x1C</td>
<td>LBR[TOS-20]_FROM_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x1D</td>
<td>LBR[TOS-20]_TO_IP</td>
</tr>
<tr>
<td>0x09</td>
<td>0x1E</td>
<td>LBR[TOS-20]_INFO</td>
</tr>
<tr>
<td>0x09</td>
<td>0x1F</td>
<td>LBR[TOS-21]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x00</td>
<td>LBR[TOS-21]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x01</td>
<td>LBR[TOS-21]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x02</td>
<td>LBR[TOS-22]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x03</td>
<td>LBR[TOS-22]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x04</td>
<td>LBR[TOS-22]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x05</td>
<td>LBR[TOS-23]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x06</td>
<td>LBR[TOS-23]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x07</td>
<td>LBR[TOS-23]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x08</td>
<td>LBR[TOS-24]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x09</td>
<td>LBR[TOS-24]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0A</td>
<td>LBR[TOS-24]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0B</td>
<td>LBR[TOS-25]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0C</td>
<td>LBR[TOS-25]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0D</td>
<td>LBR[TOS-25]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0E</td>
<td>LBR[TOS-26]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x0F</td>
<td>LBR[TOS-26]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x10</td>
<td>LBR[TOS-26]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x11</td>
<td>LBR[TOS-27]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x12</td>
<td>LBR[TOS-27]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x13</td>
<td>LBR[TOS-27]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x14</td>
<td>LBR[TOS-28]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x15</td>
<td>LBR[TOS-28]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x16</td>
<td>LBR[TOS-28]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x17</td>
<td>LBR[TOS-29]_FROM_IP</td>
</tr>
</tbody>
</table>
### Table 33-47. BIP Encodings (Contd.)

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>State Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0A</td>
<td>0x18</td>
<td>LBR[TOS-29]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x19</td>
<td>LBR[TOS-29]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x1A</td>
<td>LBR[TOS-30]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x1B</td>
<td>LBR[TOS-30]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x1C</td>
<td>LBR[TOS-30]_INFO</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x1D</td>
<td>LBR[TOS-31]_FROM_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x1E</td>
<td>LBR[TOS-31]_TO_IP</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x1F</td>
<td>LBR[TOS-31]_INFO</td>
</tr>
</tbody>
</table>

**XMM Registers**

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>State Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>0x00</td>
<td>XMM0_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>XMM0_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x02</td>
<td>XMM1_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x03</td>
<td>XMM1_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x04</td>
<td>XMM2_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x05</td>
<td>XMM2_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x06</td>
<td>XMM3_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x07</td>
<td>XMM3_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x08</td>
<td>XMM4_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x09</td>
<td>XMM4_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0A</td>
<td>XMM5_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0B</td>
<td>XMM5_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0C</td>
<td>XMM6_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0D</td>
<td>XMM6_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0E</td>
<td>XMM7_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0F</td>
<td>XMM7_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x10</td>
<td>XMM8_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x11</td>
<td>XMM8_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x12</td>
<td>XMM9_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x13</td>
<td>XMM9_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x14</td>
<td>XMM10_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x15</td>
<td>XMM10_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x16</td>
<td>XMM11_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x17</td>
<td>XMM11_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x18</td>
<td>XMM12_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x19</td>
<td>XMM12_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x1A</td>
<td>XMM13_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x1B</td>
<td>XMM13_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x1C</td>
<td>XMM14_Q0</td>
</tr>
</tbody>
</table>
### Table 33-47. BIP Encodings (Contd.)

<table>
<thead>
<tr>
<th>BBP.Type</th>
<th>BIP.ID</th>
<th>State Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>0x1D</td>
<td>XMM14_Q1</td>
</tr>
<tr>
<td>0x10</td>
<td>0x1E</td>
<td>XMM15_Q0</td>
</tr>
<tr>
<td>0x10</td>
<td>0x1F</td>
<td>XMM15_Q1</td>
</tr>
</tbody>
</table>

### 33.4.2.28 Block End Packet (BEP)

**Table 33-48. Block End Packet Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>BEP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>IP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>TriggerEn</th>
<th>Generation Scenario</th>
<th>See BBP.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Indicates the end of a packet block. The IP bit indicates if a FUP will follow, and will be set if ContextEn=1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>The block, from initial BBP to the BEP, binds to the FUP IP, if IP=1, and consumes the FUP.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 33.4.2.29 Control Flow Event (CFE) Packet

**Table 33-49. Control Flow Event Packet Definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>CFE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Packet Format</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>IP</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

IP bit indicates if a stand-alone FUP will follow.

**Dependencies**

| IA32_RTIT_CTL.EventEn && TriggerEn && ContextEn |
|---|---|

On ContextEn transitions, the CFE will be generated regardless of direction (\(1 \rightarrow 0\) or \(0 \rightarrow 1\)). VM exit is an exception, where CFE.VMEXIT depends only on the prior value of ContextEn.

**Generation Scenario**

- Software interrupt, external interrupt, user interrupt, or exception, including those injected on VM entry.
- INIT, SIPI, SMI, RSM, IRET, Shutdown.
- VM exit, if "Conceal VMX in PT" VMCS exit control is 0.
- VM entry, if "Conceal VMX in PT" VMCS entry control is 0.
- TSX Abort.

**Description**

This packet indicates that an asynchronous event or related event (see list above) has occurred. The type of event is provided in the packet (see Table 33-50 below), and, if the IP bit is set, the IP at which the event occurred is provided in a stand-alone FUP packet that follows. Further, in the case of an interrupt or exception, the vector field provides the vector of the event.

- The IP bit will be set only when ContextEn=1 before the event is taken, and either BranchEn=0 or else no FUP is generated for this event by BranchEn=1. There are some cases, such as SIPI and RSM, where no FUP is generated. Note that events that are not delivered to software, such as nested events or events which cause a VM exit, do not generate CFE packets.

**Application**

- If the IP bit is set, a FUP will follow that is stand-alone (not part of a compound packet event), and the CFE consumes the FUP. If the IP bit is not set, the CFE binds to the next FUP if PacketEn=1 (hence the CFE comes after a TIP.PGE but before the next TIP.PGD), and is stand-alone if PacketEn=0.

**CFE Packet Type and Vector Fields**

Every CFE has a Type field, which provides the type of event which generated the packet. For a subset of CFE Types, the CFE.Vector field may be valid. Details on these fields, as well as the IP to be expected in any following FUP packet, are provided in the table below.

**Table 33-50. CFE Packet Type and Vector Fields Details**

<table>
<thead>
<tr>
<th>CFE Subtype</th>
<th>Type</th>
<th>Vector</th>
<th>FUP IP</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTR</td>
<td>0x1</td>
<td>Event Vector</td>
<td>Varies</td>
<td>Used for interrupts (external and software), Exceptions, Faults, and NMI. FUP contains that address of the instruction that has not completed (NLIP for trap events, CLIP for fault events).</td>
</tr>
<tr>
<td>IRET</td>
<td>0x2</td>
<td>Invalid</td>
<td>CLIP</td>
<td></td>
</tr>
<tr>
<td>SMI</td>
<td>0x3</td>
<td>Invalid</td>
<td>NLIP</td>
<td></td>
</tr>
<tr>
<td>RSM</td>
<td>0x4</td>
<td>Invalid</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>SIPI</td>
<td>0x5</td>
<td>SIPI Vector</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td>0x6</td>
<td>Invalid</td>
<td>NLIP</td>
<td></td>
</tr>
<tr>
<td>VMENTRY</td>
<td>0x7</td>
<td>Invalid</td>
<td>CLIP</td>
<td>FUP contains IP of VMLAUNCH/VMRESUME.</td>
</tr>
<tr>
<td>CFE Subtype</td>
<td>Type</td>
<td>Vector</td>
<td>FUP IP</td>
<td>Details</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
<td>-----------------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VMEXIT</td>
<td>0x8</td>
<td>Invalid</td>
<td>Varies</td>
<td>FUP IP varies depending on type of VM exit, but will be the address of the instruction that has not completed. Will be consistent with Guest IP saved in VMCS.</td>
</tr>
<tr>
<td>VMEXIT_INTR</td>
<td>0x9</td>
<td>Event Vector</td>
<td>Varies</td>
<td>Sent in cases where VM exit was caused by an INTR event (interrupt, exception, fault, or NMI). Vector provided is for the event which caused the VM exit. FUP IP behavior matches that of INTR type above.</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>0xa</td>
<td>Invalid</td>
<td>Varies</td>
<td>FUP IP varies depending on the type of event that caused shutdown, but will be the address of the instruction that has not completed.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xb</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>UINTR</td>
<td>0xc</td>
<td>User Interrupt Vector</td>
<td>NLIP</td>
<td>User interrupt delivered.</td>
</tr>
<tr>
<td>UIRET</td>
<td>0xd</td>
<td>Invalid</td>
<td>CLIP</td>
<td>Exiting from user interrupt routine.</td>
</tr>
<tr>
<td>Reserved</td>
<td>0xe..0x1f</td>
<td>N/A</td>
<td>N/A</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
33.4.2.30 Event Data (EVD) Packet

### Table 33-51. Event Data Packet Definition

<table>
<thead>
<tr>
<th>Name</th>
<th>EVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Format</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Payload[7:0]</td>
</tr>
<tr>
<td>4</td>
<td>Payload[15:8]</td>
</tr>
<tr>
<td>5</td>
<td>Payload[23:16]</td>
</tr>
<tr>
<td>6</td>
<td>Payload[31:24]</td>
</tr>
<tr>
<td>7</td>
<td>Payload[39:32]</td>
</tr>
<tr>
<td>8</td>
<td>Payload[47:40]</td>
</tr>
<tr>
<td>9</td>
<td>Payload[55:48]</td>
</tr>
<tr>
<td>10</td>
<td>Payload[63:56]</td>
</tr>
</tbody>
</table>

**Dependencies**

IA32_RTIT_CTL.EventEn && TriggerEn && ContextEn

**Generation Scenario**

Page fault, including those injected on VM entry. VM exit, if “Suppress VMX packets on exit” VMCS exit control is 0.

**Description**

Provides additional data about the event that caused the following CFE. The Payload field is dictated by the Type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>'000000</td>
<td>Page Fault Linear Address, same as CR2 (PFA)</td>
</tr>
<tr>
<td>'000001</td>
<td>VMX Exit Qualification (VMXQ)</td>
</tr>
<tr>
<td>'000010</td>
<td>VMX Exit Reason (VMXR)</td>
</tr>
<tr>
<td>'000011 - '111111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

EVD packets are never generated in cases where a CFE is not.

**Application**

EVD packets bind to the same IP (if any) as the subsequent CFE packet.

### 33.5 TRACING IN VMX OPERATION

On processors that IA32_VMX_MISC[bit 14] reports 1, TraceEn can be set in VMX operation. The VMM can configure specific VMX controls to control what virtualization-specific data is included within the trace packets (see Section 33.5.1 for details). The VMM can also configure the VMCS to limit tracing to non-root operation, or to trace across both root and non-root operation. The VMCS controls exist to simplify virtualization of Intel PT for guest use, including the “Clear IA32_RTIT_CTL” exit control (See Section 25.7.1), “Load IA32_RTIT_CTL” entry control (See Section 25.8.1), and “Intel PT uses guest physical addresses” execution control (See Section 26.5.3).

For older processors that do not support these VMCS controls, the MSR-load areas used by VMX transitions can be employed by the VMM to restrict tracing to the desired context. See Section 33.5.2 for details. Tracing with SMM Transfer Monitor is described in Section 33.6.
33.5.1 VMX-Specific Packets and VMCS Controls

In all of the usages of VMX and Intel PT, a decoder in the host or VMM context can identify the occurrences of VMX transitions with the aid of VMX-specific packets. There are four kinds of packets relevant to VMX:

- **VMCS packet.** The VMX transitions of individual VMs can be distinguished by a decoder using the VMCS-pointer field in a VMCS packet. A VMCS packet is sent on a successful execution of VMPTRLD, and its VMCS-pointer field stores the VMCS pointer loaded by that execution. See Section 33.4.2.15 for details.

- **The NR (non-root) bit in a PIP packet.** Normally, the NR bit is set in any PIP packet generated in VMX non-root operation. In addition, PIP packets are generated with each VM entry and VM exit. Thus a transition of the NR bit from 0 to 1 indicates the occurrence of a VM entry, and a transition of 1 to 0 indicates the occurrence of a VM exit.

- **CFE packet.** Identifies VM exit and VM entry operations.

- **EVD packet.** Provides the exit reason and exit qualification for VM exits.

There are VMX controls that a VMM can set to conceal some of this VMX-specific information (by suppressing its recording) and thereby prevent it from leaking across virtualization boundaries. There is one of these controls (each of which is called “conceal VMX from PT”) of each type of VMX control.

<table>
<thead>
<tr>
<th>Type of VMX Control</th>
<th>Bit Position</th>
<th>Value</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secondary processor-based VM-execution control</td>
<td>19</td>
<td>0</td>
<td>Each PIP generated in VM non-root operation will set the NR bit. PSB+ in VMX non-root operation will include the VMCS packet, to ensure that the decoder knows which guest is currently in use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Each PIP generated in VMX non-root operation will clear the NR bit. PSB+ in VMX non-root operation will not include the VMCS packet.</td>
</tr>
<tr>
<td>VM-exit control</td>
<td>24</td>
<td>0</td>
<td>Each VM exit generates a PIP in which the NR bit is clear, and a CFE/EVD if Event Trace is enabled. In addition, SMM VM exits generate VMCS packets.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>VM exits do not generate PIPs, CFEs, or EVDs, and no VMCS packets are generated on SMM VM exits.</td>
</tr>
<tr>
<td>VM-entry control</td>
<td>17</td>
<td>0</td>
<td>Each VM entry generates a PIP in which the NR bit is set (except VM entries that return from SMM to VMX root operation), and a CFE if Event Trace is enabled. In addition, VM entries that return from SMM generate VMCS packets.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>VM entries do not generate PIPs or CFEs, and no VMCS packets are generated on VM entries that return from SMM.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. These are the positions of the control bits in the relevant VMX control fields.

The 0-settings of these VMX controls enable all VMX-specific packet information. The scenarios that would use these default settings also do not require the VMM to use VMX MSR-load areas to enable and disable trace-packet generation across VMX transitions.

If IA32_VMX_MISC[bit 14] reports 0, the 1-settings of the VMX controls in Table 33-52 are not supported, and VM entry will fail on any attempt to set them.

33.5.2 Managing Trace Packet Generation Across VMX Transitions

In tracing scenarios that collect packets for both VMX root operation and VMX non-root operation, a host executive can manage the MSRs associated with trace packet generation directly. The states of these MSRs need not be modified across VMX transitions.

For tracing scenarios that collect packets only within VMX root operation or only within VMX non-root operation, the VMM can toggle IA32_RTIT_CTL.TraceEn on VMX transitions.
33.5.2.1 System-Wide Tracing

When a host or VMM configures Intel PT to collect trace packets of the entire system, it can leave the relevant VMX controls clear to allow VMX-specific packets to provide information across VMX transitions.

The decoder will desire to identify the occurrence of VMX transitions. The packets of interests to a decoder are shown in Table 33-53.

<table>
<thead>
<tr>
<th>Event</th>
<th>Packets</th>
<th>Enable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM exit</td>
<td>EVD.VMXR, EVD.VMXPQ, CFE.VMEXIT*</td>
<td>EventEn</td>
<td>The CFE identifies the transfer as a VM exit, while the associated EVDs provide the exit reason and exit qualification.</td>
</tr>
<tr>
<td></td>
<td>FUP(GuestIP)</td>
<td>BranchEn or EventEn</td>
<td>The FUP indicates at which point in the guest flow the VM exit occurred. This is important, since VM exit can be an asynchronous event. The IP will match that written into the VMCS.</td>
</tr>
<tr>
<td></td>
<td>PIP(HostCR3, NR=0)</td>
<td></td>
<td>The PIP packet provides the new host CR3 value, as well as indication that the logical processor is entering VMX root operation. This allows the decoder to identify the change of executing context from guest to host and load the appropriate set of binaries to continue decode.</td>
</tr>
<tr>
<td></td>
<td>TIP(HostIP)</td>
<td>BranchEn</td>
<td>The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 33.4.2.8). This is generated only in cases where CS.D or (CS.L &amp; EFER.LMA) change during the transition.</td>
</tr>
<tr>
<td>VM entry</td>
<td>CFEVMENTRY, FUP(CLIP)</td>
<td>EventEn</td>
<td>The CFE identifies the transfer as a VM entry, while the FUP identifies the VMLAUNCH/VMRESUME IP.</td>
</tr>
<tr>
<td></td>
<td>PIP(GuestCR3, NR=1)</td>
<td>BranchEn</td>
<td>The PIP packet provides the new guest CR3 value, as well as indication that the logical processor is entering VMX non-root operation. This allows the decoder to identify the change of executing context from host to guest and load the appropriate set of binaries to continue decode.</td>
</tr>
<tr>
<td></td>
<td>TIP(GuestIP)</td>
<td>BranchEn</td>
<td>The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX non-root operation. This should match the RIP loaded from the VMCS. Note, this packet could be preceded by a MODE.Exec packet (Section 33.4.2.8). This is generated only in cases where CS.D or (CS.L &amp; EFER.LMA) change during the transition.</td>
</tr>
</tbody>
</table>

Since the VMX controls that suppress packet generation are cleared, a VMCS packet will be included in all PSB+ for this usage scenario. Additionally, VMPTRLD will generate such a packet. Thus the decoder can distinguish the execution context of different VMs.

When the host VMM configures a system to collect trace packets in this scenario, it should emulate CPUID to report CPUID.(EAX=07H, ECX=0):EBX[bit 26] as 0 to guests, indicating to guests that Intel PT is not available.

***VMX TSC Manipulation***

The TSC packets generated while in VMX non-root operation will include any changes resulting from the use of a VMM’s use of the TSC offsetting or TSC scaling VMX controls (see Chapter 26, “VMX Non-Root Operation“). In this system-wide usage model, the decoder may need to account for the effect of per-VM adjustments in the TSC packets generated in VMX non-root operation and the absence of TSC adjustments in TSC packets generated in VMX root operation. The VMM can supply this information to the decoder.

33.5.2.2 Guest-Only Tracing

A VMM can configure trace-packet generation while in VMX non-root operation for guests executing normally. This is accomplished by utilizing VMCS controls to manipulate the guest IA32_RTIT_CTL value on VMX transitions. For
older processors that do not support these VMCS controls, a VMM can use the VMX MSR-load areas on VM exits (see Section 25.7.2, “VM-Exit Controls for MSRs”) and VM entries (see Section 25.8.2, “VM-Entry Controls for MSRs”) to limit trace-packet generation to the guest environment. 

For this usage, VM entry is programmed to enable trace packet generation, while VM exit is programmed to clear IA32_RTIT_CTL.TraceEn so as to disable trace-packet generation in the host. Further, if it is preferred that the guest packet stream contain no indication that execution was in VMX non-root operation, the VMM should set to 1 all the VMX controls enumerated in Table 33-52.

### 33.5.2.3 Emulation of Intel PT Traced State

If a VMM emulates an element of processor state by taking a VM exit on reads and/or writes to that piece of state, and the state element impacts Intel PT packet generation or values, it may be incumbent upon the VMM to insert or modify the output trace data.

If a VM exit is taken on a guest write to CR3 (including “MOV CR3” as well as task switches), the PIP packet normally generated on the CR3 write will be missing. To avoid decoder confusion when the guest trace is decoded, the VMM should emulate the missing PIP by writing it into the guest output buffer. If the guest CR3 value is manipulated, the VMM may also need to manipulate the IA32_RTIT_CR3_MATCH value, in order to ensure the trace behavior matches the guest’s expectation.

Similarly, if a VMM emulates the TSC value by taking a VM exit on RDTSC, the TSC packets generated in the trace may mismatch the TSC values returned by the VMM on RDTSC. To ensure that the trace can be properly aligned with software logs based on RDTSC, the VMM should either make corresponding modifications to the TSC packet values in the guest trace, or use mechanisms such as TSC offsetting or TSC scaling in place of exiting.

### 33.5.2.4 TSC Scaling

When TSC scaling is enabled for a guest using Intel PT, the VMM should ensure that the value of Maximum Non-Turbo Ratio[15:8] in MSRPLATFORM_INFO (MSR 0CEH) and the TSC/“core crystal clock” ratio (EBX/EAX) in CPUID leaf 15H are set in a manner consistent with the resulting TSC rate that will be visible to the VM. This will allow the decoder to properly apply TSC packets, MTC packets (based on the core crystal clock or ART, whose frequency is indicated by CPUID leaf 15H), and CBR packets (which indicate the ratio of the processor frequency to the Max Non-Turbo frequency). Absent this, or separate indication of the scaling factor, the decoder will be unable to properly track time in the trace. See Section 33.8.3 for details on tracking time within an Intel PT trace.

### 33.5.2.5 Failed VM Entry

The packets generated by a failed VM entry depend both on the VMCS configuration, as well as on the type of failure. The results to expect are summarized in the table below. Note that packets in italics may or may not be generated, depending on implementation choice, and the point of failure.

<table>
<thead>
<tr>
<th>Usage Model</th>
<th>Entry Configuration</th>
<th>Early Failure (fall through to next IP)</th>
<th>Late Failure (VM exit like)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System-Wide</td>
<td>No use of &quot;Load IA32_RTIT_CTL&quot; entry control or VM-entry MSR-load area</td>
<td>TIP (NextIP)</td>
<td>CFE.VMENTRY, FUP(CLIP) if EventEn=1 PIP(Guest CR3, NR=1), TraceEn 0→1 Packets (See Section 33.2.8.3), PIP(HostCR3, NR=0), TIP(HostIP)</td>
</tr>
<tr>
<td>VMM Only</td>
<td>&quot;Load IA32_RTIT_CTL&quot; entry control or VM-entry MSR-load area used to clear TraceEn</td>
<td>TIP (NextIP)</td>
<td>TraceEn 0→1 Packets (See Section 33.2.8.3), TIP(HostIP)</td>
</tr>
<tr>
<td>VM Only</td>
<td>&quot;Load IA32_RTIT_CTL&quot; entry control or VM-entry MSR-load area used to set TraceEn</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
33.5.2.6 VMX Abort

VMX abort conditions take the processor into a shutdown state. On a VM exit that leads to VMX abort, some packets (FUP, PIP) may be generated, but any expected TIP, TIP.PGE, or TIP.PGD may be dropped.

33.6 TRACING AND SMM TRANSFER MONITOR (STM)

The SMM-transfer monitor (STM) is a VMM that operates inside SMM while in VMX root operation. An STM operates in conjunction with an executive monitor. The latter operates outside SMM and in VMX root operation. Transitions from the executive monitor or its VMs to the STM are called SMM VM exits. The STM returns from SMM via a VM entry to the VM in VMX non-root operation or the executive monitor in VMX root operation.

Intel PT supports tracing in an STM similar to tracing support for VMX operation as described above in Section 33.5. As a result, on a SMM VM exit resulting from #SMI, TraceEn is neither saved nor cleared by default. Software can save the state of the trace configuration MSRs and clear TraceEn using the MSR load/save lists.

Within Event Trace, SMM VM exits generate packets indicating both an #SMI and a VM exit. Similarly, VM entries that return from SMM generate packets that indicate both an RSM and a VM entry. SMM VM exits initiated by the VMCALL instruction do not generate any CFE packet, though the subsequent VM entry returning from SMM will generate a CFE.RSM.

33.7 PACKET GENERATION SCENARIOS

The following tables provide examples of packet generation for various operations. The following acronyms are used in the packet examples below:

- CLIP - Current LIP
- NLIP - Next Sequential LIP
- BLIP - Branch Target LIP

Table 33-55 illustrates the packets generated by a series of example operations, assuming that PacketEn (TriggerEn && ContextEn && FilterEn && BranchEn) is set before and after the operation.

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Normal non-jump operation</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Conditional branch 6th branch in internal TNT buffer</td>
<td>TNT</td>
</tr>
<tr>
<td>3</td>
<td>Conditional branch 1st, 5th branch in internal TNT buffer</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>Near indirect JMP or CALL</td>
<td>TIP(BLIP)</td>
</tr>
<tr>
<td>5</td>
<td>Direct near JMP or CALL</td>
<td>None</td>
</tr>
<tr>
<td>6</td>
<td>Near RET Uncompressed</td>
<td>TIP(BLIP)</td>
</tr>
<tr>
<td>7</td>
<td>Near RET Compressed, 6th branch in internal TNT buffer</td>
<td>TNT</td>
</tr>
<tr>
<td>8</td>
<td>Far Branch Assumes no update to CR3, CS.L, or CS.D</td>
<td>TIP(BLIP)</td>
</tr>
<tr>
<td>9</td>
<td>Far Branch Assumes update to CR3</td>
<td>PIP(NewCR3), TIP(BLIP)</td>
</tr>
<tr>
<td>10</td>
<td>Far Branch Assumes update to CR3 and CS.D/CS.L</td>
<td>PIP(NewCR3), MODE.Exec, TIP(BLIP)</td>
</tr>
<tr>
<td>11</td>
<td>External Interrupt or NMI Assumes no update to CR3, CS.D, or CS.L</td>
<td>FUP(NLIP), PIP(NewCR3), MODE.Exec, TIP(BLIP)</td>
</tr>
<tr>
<td>12</td>
<td>External Interrupt or NMI Assumes update to CR3 and CS.D/CS.L</td>
<td>FUP(NLIP), PIP(NewCR3), MODE.Exec, TIP(BLIP)</td>
</tr>
<tr>
<td>13</td>
<td>Exception/Fault or Software Interrupt Assumes no update to CR3, CS.D, or CS.L</td>
<td>FUP(CLIP), TIP(BLIP)</td>
</tr>
<tr>
<td>14</td>
<td>MOV to CR3</td>
<td>PIP(NewCR3, NR)</td>
</tr>
</tbody>
</table>
Table 33-56. Packet Generation under Different Example Operations

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>VM exit</td>
<td>Assumes system-wide tracing, see Section 33.5.2.1</td>
<td>See Table 33-53</td>
</tr>
<tr>
<td>16</td>
<td>VM entry</td>
<td>Assumes system-wide tracing, see Section 33.5.2.1</td>
<td>See Table 33-53</td>
</tr>
<tr>
<td>17</td>
<td>ENCLU[ENTER] / ENCLU[ERESUME] / ENCLU[EEEXIT] / AEX/EEE</td>
<td>Only debug enclaves allow PacketEn to be set during enclave execution. Assumes no change to CS.L or CS.D.</td>
<td>FUP(CLIP), TIP(BLIP)</td>
</tr>
<tr>
<td>18</td>
<td>XBEGIN/XACQUIRE/XEND/XRELEASE</td>
<td>Does not begin/end transactional execution</td>
<td>None</td>
</tr>
<tr>
<td>19</td>
<td>XBEGIN/XACQUIRE</td>
<td>Assumes beginning of transactional execution</td>
<td>MODE.TSX(InTX=1, TXAbort=0), FUP(CLIP)</td>
</tr>
<tr>
<td>20</td>
<td>XEND/XRELEASE</td>
<td>Completes transaction</td>
<td>MODE.TSX(InTX=0, TXAbort=0), FUP(CLIP)</td>
</tr>
<tr>
<td>21</td>
<td>XABORT or Asynchronous Abort</td>
<td>Aborts transactional execution</td>
<td>MODE.TSX(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)</td>
</tr>
<tr>
<td>22</td>
<td>INIT</td>
<td>On BSP. Assumes no CR3, CS.D, or CS.L update.</td>
<td>FUP(NLIP)</td>
</tr>
<tr>
<td>23</td>
<td>INIT</td>
<td>On AP, goes to wait-for-SIPI. Assumes no CR3 update.</td>
<td>FUP(NLIP)</td>
</tr>
<tr>
<td>24</td>
<td>SIPI</td>
<td>Assumes no CS.D or CS.L update</td>
<td>TIP.PGE(SIPI.LIP)</td>
</tr>
<tr>
<td>25</td>
<td>Wake from state deeper than C0.1, P-state change, or other scenario where timing packets (MTC, CYC) may have ceased.</td>
<td>TSC if TSCEn=1; TMA if TSCEn=MTCEn=1</td>
<td>TSC?, TMA?, CBR</td>
</tr>
<tr>
<td>26</td>
<td>UINTR</td>
<td>User interrupt handler entry.</td>
<td>FUP(NLIP)</td>
</tr>
<tr>
<td>27</td>
<td>UIRET</td>
<td>Exiting from user interrupt handler.</td>
<td>FUP(NLIP)</td>
</tr>
</tbody>
</table>

Table 33-56 illustrates the packets generated in example scenarios where the operation alters the value of PacketEn. Note that insertion of PSB+ is not included here, though it can be coincident with initial enabling of Intel PT. See Section 33.3.7 for details.

Table 33-56. Packet Generation with Operations That Alter the Value of PacketEn

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>PktEn Before</th>
<th>PktEn After</th>
<th>CnTxEn After</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WRMSR/XRSTORS that changes TraceEn 0 → 1</td>
<td>0 1 0</td>
<td>TSC if TSCEn=1; TMA if TSCEn=MTCEn=1</td>
<td>TSC?, TMA?, CBR, MODE.Exec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>WRMSR/XRSTORS that changes TraceEn 0 → 1</td>
<td>0 1 1</td>
<td>TSC if TSCEn=1; TMA if TSCEn=MTCEn=1</td>
<td>TSC?, TMA?, CBR, MODE.Exec, TIP.PGE(NLIP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>WRMSR that changes TraceEn 1 → 0</td>
<td>1 0 D.C.</td>
<td>FUP(CLIP), TIP.PGD()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Taken Branch</td>
<td>1 0 1</td>
<td>Source is in IP filter region. Target is outside IP filter region.</td>
<td>TIP.PGD(BLIP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Taken Branch, Interrupt, EEXIT, etc.</td>
<td>0 1 1</td>
<td>Source is outside IP filter region. Target is in IP filter region.</td>
<td>TIP.PGE(BLIP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Far Branch, Interrupt, EENTER, etc.</td>
<td>1 0 0</td>
<td>Requires change to CPL or CR3, or entry to opt-out enclave.</td>
<td>TIP.PGD()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 33-57. Examples of PTWRITE when TriggerEn && PTWEn is True

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>ContextEn</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Trap-like event (external interrupt, NMI, VM exit/entry, etc.)</td>
<td>D.C.</td>
<td>Requires change to CPL or CR3.</td>
<td>FUP(NLIP), TIP.PGD()</td>
</tr>
<tr>
<td>8</td>
<td>Fault-like event (exception/fault, software interrupt, VM exit/entry, etc.)</td>
<td>D.C.</td>
<td>Requires change to CPL or CR3.</td>
<td>FUP(CLIP), TIP.PGD()</td>
</tr>
<tr>
<td>9</td>
<td>SMI, VM exit/entry</td>
<td>0</td>
<td>TraceEn is cleared.</td>
<td>FUP(NLIP), TIP.PGD()</td>
</tr>
<tr>
<td>10</td>
<td>RSM, VM exit/entry</td>
<td>1</td>
<td>TraceEn is set.</td>
<td>See Case 2 for packets on enable. FUP/TIP.PGE IP is the BLIP.</td>
</tr>
<tr>
<td>11</td>
<td>VM Exit</td>
<td>1</td>
<td>Assumes guest-only tracing, see Section 33.5.2.2.</td>
<td>FUP(VMCSg.RIP), TIP.PGD()</td>
</tr>
<tr>
<td>12</td>
<td>VM entry</td>
<td>0</td>
<td>Assumes guest-only tracing, see Section 33.5.2.2.</td>
<td>TIP.PGE(VMCSg.RIP)</td>
</tr>
</tbody>
</table>

Table 33-58 illustrates examples of Power Event Trace, assuming TriggerEn & PwrEvtEn is true.

Table 33-58. Examples of Power Event Trace when TriggerEn && PwrEvtEn is True

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>ContextEn &amp; FilterEn</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PTWRITE rm32/64</td>
<td>0</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
Table 33-59 illustrates examples of Event Trace, assuming TriggerEn && ContextEn && EventEn is true. In all cases, other trace sources (e.g., BranchEn), if enabled, may generate additional packets. For details, see the other tables in this section.

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>ContextEn &amp; FilterEn</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>PTWRITE rm32</td>
<td>1</td>
<td>FUP, PTW.IP=1 if FUPonPTW=1</td>
<td>PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?</td>
</tr>
<tr>
<td>3</td>
<td>PTWRITE rm64</td>
<td>1</td>
<td>FUP, PTW.IP=1 if FUPonPTW=1</td>
<td>PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?</td>
</tr>
</tbody>
</table>

Table 33-58. Examples of Power Event Trace when (Contd.)TriggerEn && PwrEvtEn is True

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>ContextEn Before</th>
<th>ContextEn After</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IRET</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.IRET(IP=1), FUP(CLIP)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IRET</td>
<td>0</td>
<td>1</td>
<td>CFE(IRET)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>External interrupt, including NMI</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.INTR(IP=1, Vector), FUP(NLIP)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>External interrupt, including NMI</td>
<td>1</td>
<td>1</td>
<td>Assumes BranchEn=1, illustrates the shared FUP.</td>
<td>CFE.INTR(IP=0, Vector), FUP(NLIP), TIP(BLIP)</td>
</tr>
<tr>
<td>5</td>
<td>Sw Interrupt, Exception/Fault other than #PF</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.INTR(IP=1, Vector), FUP(CLIP)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Page Fault (#PF)</td>
<td>1</td>
<td>D.C.</td>
<td>EVD.PFA, CFE.INTR(IP=1,14), FUP(CLIP)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Page Fault (#PF)</td>
<td>0</td>
<td>D.C.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SMI</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.SMI(IP=1), FUP(NLIP)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RSM, TraceEn restored to 1</td>
<td>D.C.</td>
<td>1</td>
<td>CFE.RSM(IP=0)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Entry to Shutdown</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.SHUTDOWN(IP=1), FUP(CLIP)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>VM exit caused by interrupt, fault, or SMI</td>
<td>1</td>
<td>D.C.</td>
<td>Assumes &quot;Conceal VMX in PT&quot; exit control is 0.</td>
<td>EVD.VMQ, EVD.VMRX, CFE.VMEXIT_INTR(IP=1, Vector), FUP(VMCSg.LIP)</td>
</tr>
<tr>
<td>14</td>
<td>VM exit caused by other than interrupt, fault, or SMI</td>
<td>1</td>
<td>D.C.</td>
<td>Assumes &quot;Conceal VMX in PT&quot; exit control is 0.</td>
<td>EVD.VMQ, EVD.VMRX, CFE.VMEXIT(IP=1), FUP(VMCSg.LIP)</td>
</tr>
<tr>
<td>15</td>
<td>VM exit caused by other than interrupt, fault, or SMI</td>
<td>0</td>
<td>1</td>
<td>Assumes &quot;Conceal VMX in PT&quot; exit control is 0.</td>
<td>CFE.VMEXIT(IP=0)</td>
</tr>
<tr>
<td>16</td>
<td>VM entry</td>
<td>1</td>
<td>D.C.</td>
<td>Assumes &quot;Conceal VMX in PT&quot; entry control is 0.</td>
<td>CFE.VMENTRY(IP=1), FUP(VMCSh.LIP)</td>
</tr>
<tr>
<td>17</td>
<td>AEX/EEE, from opt-out (non-debug) enclave</td>
<td>0</td>
<td>0</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>AEX/EEE, from opt-out (non-debug) enclave</td>
<td>0</td>
<td>1</td>
<td>CFE.INTR(IP=0)</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>AEX, from opt-in (debug) enclave</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.INTR(IP=1, Vec), FUP(AEP LIP)</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>INIT</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.INIT(IP=1), FUP(NLIP)</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>SIPI</td>
<td>1</td>
<td>D.C.</td>
<td>CFE.SIPI(IP=0)</td>
<td></td>
</tr>
</tbody>
</table>
33.8 SOFTWARE CONSIDERATIONS

33.8.1 Tracing SMM Code

Nothing prevents an SMM handler from configuring and enabling packet generation for its own use. As described in Section 33.2.9.3, SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follow:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion.
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.
5. RSM
   - Software must ensure that TraceEn=0 at the time of RSM. Tracing RSM is not a supported usage model, and the packets generated by RSM are undefined.
   - For processors on which Intel PT and LBR use are mutually exclusive (see Section 33.3.1.2), any RSM during which TraceEn is restored to 1 will suspend any LBR or BTS logging.

33.8.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the "in-use" ownership of an agent who already configured the trace configuration MSRs, see architectural MSRs with the prefix "IA32_RTIT_ " in Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, where "in-use" can be determined by reading the "enable bits" in the configuration MSRs.
- Relinquish ownership of the trace configuration MSRs by clearing the "enabled bits" of those configuration MSRs.

33.8.3 Tracking Time

This section describes the relationships of several clock counters whose update frequencies reside in different domains that feed into the timing packets. To track time, the decoder also needs to know the regularity or irregularity of the occurrences of various timing packets that store those clock counters.

Intel PT provides time information for three different but related domains:

<table>
<thead>
<tr>
<th>Case</th>
<th>Operation</th>
<th>ContextEn Before</th>
<th>ContextEn After</th>
<th>Details</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>STI/CLI/POPF</td>
<td>1</td>
<td>1</td>
<td>Assumes a change to RFLAGS.IF.</td>
<td>MODE.Exec, FUP(CLIP)</td>
</tr>
</tbody>
</table>
• Processor timestamp counter

This counter increments at the max non-turbo or P1 frequency, and its value is returned on a RDTSC. Its frequency is fixed. The TSC packet holds the lower 7 bytes of the timestamp counter value. The TSC packet occurs occasionally and are much less frequent than the frequency of the time stamp counter. The timestamp counter will continue to increment when the processor is in deep C-States, with the exception of processors reporting CPUID.80000007H:EDX.InvariantTSC[bit 8] = 0.

• Core crystal clock

The ratio of the core crystal clock to timestamp counter frequency is known as P, and can be calculated as CPUID.15H:EBX[31:0] / CPUID.15H:EAX[31:0]. The frequency of the core crystal clock is fixed and lower than that of the timestamp counter. The periodic MTC packet is generated based on software-selected multiples of the crystal clock frequency. The MTC packet is expected to occur more frequently than the TSC packet.

• Processor core clock

The processor core clock frequency can vary due to P-state and thermal conditions. The CYC packet provides elapsed time as measured in processor core clock cycles relative to the last CYC packet. A decoder can use all or some combination of these packets to track time at different resolutions throughout the trace packets.

### 33.8.3.1 Time Domain Relationships

The three domains are related by the following formula:

\[
\text{TimeStampValue} = (\text{CoreCrystalClockValue} \times P) + \text{AdjustedProcessorCycles} + \text{Software_Offset};
\]

The CoreCrystalClockValue, also known as the Always Running Timer (ART) value, can provide the coarse-grained component of the TSC value. P, or the TSC/ART ratio, can be derived from CPUID leaf 15H, as described in Section 33.8.3.

The AdjustedProcessorCycles component provides the fine-grained distance from the rising edge of the last core crystal clock. Specifically, it is a cycle count in the same frequency as the timestamp counter from the last crystal clock rising edge. The value is adjusted based on the ratio of the processor core clock frequency to the Maximum Non-Turbo (or P1) frequency.

The Software_Offsets component includes software offsets that are factored into the timestamp value, such as IA32_TSC_ADJUST.

### 33.8.3.2 Estimating TSC within Intel PT

For many usages, it may be useful to have an estimated timestamp value for all points in the trace. The formula provided in Section 33.8.3.1 above provides the framework for how such an estimate can be calculated from the various timing packets present in the trace.

The TSC packet provides the precise timestamp value at the time it is generated; however, TSC packets are infrequent, and estimates of the current timestamp value based purely on TSC packets are likely to be very inaccurate for this reason. In order to get more precise timing information between TSC packets, CYC packets and/or MTC packets should be enabled.

MTC packets provide incremental updates of the CoreCrystalClockValue. On processors that support CPUID leaf 15H, the frequency of the timestamp counter and the core crystal clock is fixed, thus MTC packets provide a means to update the running timestamp estimate. Between two MTC packets A and B, the number of crystal clock cycles passed is calculated from the 8-bit payloads of respective MTC packets:

\[
(\text{CTC}_B - \text{CTC}_A), \text{ where } \text{CTC}_j = \text{MTC}_j[15:8] \ll \text{IA32_RTIT_CTL.MTCFreq} \text{ and } i = A, B.
\]

The time from a TSC packet to the subsequent MTC packet can be calculated using the TMA packet that follows the TSC packet. The TMA packet provides both the crystal clock value (lower 16 bits, in the CTC field) and the AdjustedProcessorCycles value (in the FastCounter field) that can be used in the calculation of the corresponding core crystal clock value of the TSC packet.

When the next MTC after a pair of TSC/TMA is seen, the number of crystal clocks passed since the TSC packet can be calculated by subtracting the TMA.CTC value from the time indicated by the MTCNext packet by

\[
\text{CTC}_{\text{Delta}}[15:0] = (\text{TCN}[15:0] - \text{TMA.CTC}[15:0]), \text{ where } \text{CTC}_{\text{Next}} = \text{MTC}_\text{Payload} \ll \text{IA32_RTIT_CTL.MTCFreq}.
\]
The TMA.FastCounter field provides the number of AdjustedProcessorCycles since the last crystal clock rising edge, from which it can be determined the percentage of the next crystal clock cycle that had passed at the time of the TSC packet.

CYC packets can provide further precision of an estimated timestamp value to many non-timing packets, by providing an indication of the time passed between other timing packets (MTCs or TSCs).

When enabled, CYC packets are sent preceding each CYC-eligible packet, and provide the number of processor core clock cycles that have passed since the last CYC packet. Thus between MTCs and TSCs, the accumulated CYC values can be used to estimate the AdjustedProcessorCycles component of the timestamp value. The accumulated CPU cycles will have to be adjusted to account for the difference in frequency between the processor core clock and the P1 frequency. The necessary adjustment can be estimated using the core:bus ratio value given in the CBR packet, by multiplying the accumulated cycle count value by $P1/CBR_{payload}$.

Note that stand-alone TSC packets (that is, TSC packets that are not a part of a PSB+) are typically generated only when generation of other timing packets (MTCs and CYCs) has ceased for a period of time. Example scenarios include when Intel PT is re-enabled, or on wake after a sleep state. Thus any calculated estimate of the timestamp value leading up to a TSC packet will likely result in a discrepancy, which the TSC packet serves to correct.

A greater level of precision may be achieved by calculating the CPU clock frequency, see Section 33.8.3.4 below for a method to do so using Intel PT packets.

CYCs can be used to estimate time between TSCs even without MTCs, though this will likely result in a reduction in estimated TSC precision.

### 33.8.3.3 VMX TSC Manipulation

When software executes in non-Root operation, additional offset and scaling factors may be applied to the TSC value. These are optional, but may be enabled via VMCS controls on a per-VM basis. See Chapter 26, “VMX Non-Root Operation,” for details on VMX TSC offsetting and TSC scaling.

Like the value returned by RDTSC, TSC packets will include these adjustments, but other timing packets (such as MTC, CYC, and CBR) are not impacted. In order to use the algorithm above to estimate the TSC value when TSC scaling is in use, it will be necessary for software to account for the scaling factor. See Section 33.5.2.4 for details.

### 33.8.3.4 Calculating Frequency with Intel PT

Because Intel PT can provide both wall-clock time and processor clock cycle time, it can be used to measure the processor core clock frequency. Either TSC or MTC packets can be used to track the wall-clock time. By using CYC packets to count the number of processor core cycles that pass in between a pair of wall-clock time packets, the ratio between processor core clock frequency and TSC frequency can be derived. If the P1 frequency is known, it can be applied to determine the CPU frequency. See Section 33.8.3.1 above for details on the relationship between TSC, MTC, and CYC.
34.1 OVERVIEW

Intel® Software Guard Extensions (Intel® SGX) is a set of instructions and mechanisms for memory accesses added to Intel® Architecture processors. Intel SGX can encompass two collections of instruction extensions, referred to as SGX1 and SGX2, see Table 34-1 and Table 34-2. The SGX1 extensions allow an application to instantiate a protected container, referred to as an enclave. The enclave is a trusted area of memory, where critical aspects of the application functionality have hardware-enhanced confidentiality and integrity protections. New access controls to restrict access to software not resident in the enclave are also introduced. The SGX2 extensions allow additional flexibility in runtime management of enclave resources and thread execution within an enclave.

Chapter 35 covers main concepts, objects and data structure formats that interact within the Intel SGX architecture. Chapter 36 covers operational aspects ranging from preparing an enclave, transferring control to enclave code, and programming considerations for the enclave code and system software providing support for enclave execution. Chapter 37 describes the behavior of Asynchronous Enclave Exit (AEX) caused by events while executing enclave code. Chapter 38 covers the syntax and operational details of the instruction and associated leaf functions available in Intel SGX. Chapter 39 describes interaction of various aspects of IA32 and Intel® 64 architectures with Intel SGX. Chapter 40 covers Intel SGX support for application debug, profiling, and performance monitoring.

34.2 ENCLAVE INTERACTION AND PROTECTION

Intel SGX allows the protected portion of an application to be distributed in the clear. Before the enclave is built, the enclave code and data are free for inspection and analysis. The protected portion is loaded into an enclave where its code and data is measured. Once the application’s protected portion of the code and data are loaded into an enclave, memory access controls are in place to restrict access by external software. An enclave can prove its identity to a remote party and provide the necessary building-blocks for secure provisioning of keys and credentials. The application can also request an enclave-specific and platform-specific key that it can use to protect keys and data that it wishes to store outside the enclave.1

Figure 34-1. An Enclave Within the Application’s Virtual Address Space

1. For additional information, see white papers on Intel SGX at https://www.intel.com/content/www/us/en/developer/tools/isa-extensions/overview.html.
Introduction to Intel® Software Guard Extensions

Intel SGX introduces two significant capabilities to the Intel Architecture. First is the change in enclave memory access semantics. The second is protection of the address mappings of the application.

34.3 Enclave Life Cycle

Enclave memory management is divided into two parts: address space allocation and memory commitment. Address space allocation is the specification of the range of linear addresses that the enclave may use. This range is called the ELRANGE. No actual resources are committed to this region. Memory commitment is the assignment of actual memory resources (as pages) within the allocated address space. This two-phase technique allows flexibility for enclaves to control their memory usage and to adjust dynamically without overusing memory resources when enclave needs are low. Commitment adds physical pages to the enclave. An operating system may support separate allocate and commit operations.

During enclave creation, code and data for an enclave are loaded from a clear-text source, i.e., from non-enclave memory.

Untrusted application code starts using an initialized enclave typically by using the EENTER leaf function provided by Intel SGX to transfer control to the enclave code residing in the protected Enclave Page Cache (EPC). The enclave code returns to the caller via the EEXIT leaf function. Upon enclave entry, control is transferred by hardware to software inside the enclave. The software inside the enclave switches the stack pointer to one inside the enclave. When returning back from the enclave, the software swaps back the stack pointer then executes the EEXIT leaf function.

On processors that support the SGX2 extensions, an enclave writer may add memory to an enclave using the SGX2 instruction set, after the enclave is built and running. These instructions allow adding additional memory resources to the enclave for use in such areas as the heap. In addition, SGX2 instructions allow the enclave to add new threads to the enclave. The SGX2 features provide additional capabilities to the software model without changing the security properties of the Intel SGX architecture.

Calling an external procedure from an enclave could be done using the EEXIT leaf function. Software would use EEXIT and a software convention between the trusted section and the untrusted section.

An active enclave consumes resources from the Enclave Page Cache (EPC, see Section 34.5). Intel SGX provides the EREMOVE instruction that an EPC manager can use to reclaim EPC pages committed to an enclave. The EPC manager uses EREMOVE on every enclave page when the enclave is torn down. After successful execution of EREMOVE the EPC page is available for allocation to another enclave.

34.4 Data Structures and Enclave Operation

There are 2 main data structures associated with operating an enclave, the SGX Enclave Control Structure (SECS, see Section 35.7) and the Thread Control Structure (TCS, see Section 35.8).

There is one SECS for each enclave. The SECS contains meta-data about the enclave which is used by the hardware and cannot be directly accessed by software. Included in the SECS is a field that stores the enclave build measurement value. This field, MRENCLAVE, is initialized by the ECREATE instruction and updated by every EADD and EEXTEND. It is locked by EINIT.

Every enclave contains one or more TCS structures. The TCS contains meta-data used by the hardware to save and restore thread specific information when entering/exiting the enclave. There is one field, FLAGS, that may be accessed by software. This field can only be accessed by debug enclaves. The flag bit, DBGOPTIN, allows to single step into the thread associated with the TCS. (see Section 35.8.1)

The SECS is created when ECREATE (see Table 34-1) is executed. The TCS can be created using the EADD instruction or the SGX2 instructions (see Table 34-2).

34.5 Enclave Page Cache

The Enclave Page Cache (EPC) is the secure storage used to store enclave pages when they are a part of an executing enclave. For an EPC page, hardware performs additional access control checks to restrict access to the page. After the current page access checks and translations are performed, the hardware checks that the EPC page
is accessible to the program currently executing. Generally an EPC page is only accessed by the owner of the executing enclave or an instruction which is setting up an EPC page.

The EPC is divided into EPC pages. An EPC page is 4KB in size and always aligned on a 4KB boundary.

Pages in the EPC can either be valid or invalid. Every valid page in the EPC belongs to one enclave instance. Each enclave instance has an EPC page that holds its SECS. The security metadata for each EPC page is held in an internal micro-architectural structure called Enclave Page Cache Map (EPCM, see Section 34.5.1).

The EPC is managed by privileged software. Intel SGX provides a set of instructions for adding and removing content to and from the EPC. The EPC may be configured by BIOS at boot time. On implementations in which EPC memory is part of system DRAM, the contents of the EPC are protected by an encryption engine.

### 34.5.1 Enclave Page Cache Map (EPCM)

The EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds one entry for each page in the EPC. The format of the EPCM is micro-architectural, and consequently is implementation dependent. However, the EPCM contains the following architectural information:

- The status of EPC page with respect to validity and accessibility.
- An SECS identifier (see Section 35.20) of the enclave to which the page belongs.
- The type of page: regular, SECS, TCS or VA.
- The linear address through which the enclave is allowed to access the page.
- The specified read/write/execute permissions on that page.

The EPCM structure is used by the CPU in the address-translation flow to enforce access-control on the EPC pages. The EPCM structure is described in Table 35-29, and the conceptual access-control flow is described in Section 35.5.

The EPCM entries are managed by the processor as part of various instruction flows.

### 34.6 Enclave Instructions and INTEL® SGX

The enclave instructions available with Intel SGX are organized as leaf functions under three instruction mnemonics: ENCLS (ring 0), ENCLU (ring 3), and ENCLV (VT root mode). Each leaf function uses EAX to specify the leaf function index, and may require additional implicit input registers as parameters. The use of EAX is implied implicitly by the ENCLS, ENCLU, and ENCLV instructions; ModR/M byte encoding is not used with ENCLS, ENCLU, and ENCLV. The use of additional registers does not use ModR/M encoding and is implied implicitly by the respective leaf function index.

Each leaf function index is also associated with a unique, leaf-specific mnemonic. A long-form expression of Intel SGX instruction takes the form of ENCLx[LEAF_MNEMONIC], where 'x' is either 'S', 'U', or 'V'. The long-form expression provides clear association of the privilege-level requirement of a given "leaf mnemonic". For simplicity, the unique "Leaf_Mnemonic" name is used (omitting the ENCLx for convenience) throughout in this document.

Details of individual SGX leaf functions are described in Chapter 38. Table 34-1 provides a summary of the instruction leaves that are available in the initial implementation of Intel SGX, which is introduced in the 6th generation Intel Core processors. Table 34-2 summarizes enhancement of Intel SGX for future Intel processors.

**Table 34-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1**

<table>
<thead>
<tr>
<th>Supervisor Instruction</th>
<th>Description</th>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLS[ECREATE]</td>
<td>Create an enclave.</td>
<td>ENCLU[E_GETKEY]</td>
<td>Create a cryptographic key.</td>
</tr>
<tr>
<td>ENCLS[EDBG]</td>
<td>Read data from a debug enclave by debugger.</td>
<td>ENCLU[E_REPORT]</td>
<td>Create a cryptographic report.</td>
</tr>
</tbody>
</table>
34.7 DISCOVERING SUPPORT FOR INTEL® SGX AND ENABLING ENCLAVE INSTRUCTIONS

Detection of support of Intel SGX and enumeration of available and enabled Intel SGX resources are queried using the CPUID instruction. The enumeration interface comprises the following:

- Processor support of Intel SGX is enumerated by a feature flag in CPUID leaf 07H: CPUID.(EAX=07H, ECX=0H):EBX.SGX[bit 2]. If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor has support for Intel SGX, and requires opt-in enabling by BIOS via IA32_FEATURE_CONTROL MSR.

If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, CPUID will report via the available sub-leaves of CPUID.(EAX=12H) on available and/or configured Intel SGX resources.

- The available and configured Intel SGX resources enumerated by the sub-leaves of CPUID.(EAX=12H) depend on the state of BIOS configuration.

### Table 34-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

<table>
<thead>
<tr>
<th>Supervisor Instruction</th>
<th>Description</th>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLS[EDBGWR]</td>
<td>Write data into a debug enclave by debugger.</td>
<td>ENCLU[ERELEASE]</td>
<td>Re-enter an enclave.</td>
</tr>
<tr>
<td>ENCLS[EEXTEND]</td>
<td>Extend EPC page measurement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EINIT]</td>
<td>Initialize an enclave.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ELDB]</td>
<td>Load an EPC page in blocked state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ELDU]</td>
<td>Load an EPC page in unblocked state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EPA]</td>
<td>Add an EPC page to create a version array.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EREMOVE]</td>
<td>Remove an EPC page from an enclave.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ETRACK]</td>
<td>Activate EBLOCK checks.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EWB]</td>
<td>Write back/invalidate an EPC page.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 34-2. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX2

<table>
<thead>
<tr>
<th>Supervisor Instruction</th>
<th>Description</th>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLS[EAMD]</td>
<td>Allocate EPC page to an existing enclave.</td>
<td>ENCLU[EACCEPT]</td>
<td>Accept EPC page into the enclave.</td>
</tr>
<tr>
<td>ENCLS[EMODT]</td>
<td>Modify EPC page type.</td>
<td>ENCLU[EACCEPTCOPY]</td>
<td>Copy contents to an augmented EPC page and accept the EPC page into the enclave.</td>
</tr>
</tbody>
</table>

### Table 34-3. VMX Operation and Supervisor Mode Enclave Instruction Leaf Functions in Long-Form of OVERSUB

<table>
<thead>
<tr>
<th>VMX Operation</th>
<th>Description</th>
<th>Supervisor Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLV[ESETCONTEXT]</td>
<td>Set virtualization context.</td>
<td>ENCLS[ELDBC/UC]</td>
<td>Load an EPC page with conflict reporting.</td>
</tr>
</tbody>
</table>
34.7.1 Intel® SGX Opt-In Configuration

On processors that support Intel SGX, IA32_FEATURE_CONTROL provides the SGX_ENABLE field (bit 18). Before system software can configure and enable Intel SGX resources, BIOS is required to set IA32_FEATURE_CONTROL.SGX_ENABLE = 1 to opt-in the use of Intel SGX by system software.

The semantics of setting SGX_ENABLE follows the rules of IA32_FEATURE_CONTROL.LOCK (bit 0). Software is considered to have opted into Intel SGX if and only if IA32_FEATURE_CONTROL.SGX_ENABLE and IA32_FEATURE_CONTROL.LOCK are set to 1. The setting of IA32_FEATURE_CONTROL.SGX_ENABLE (bit 18) is not reflected by CPUID.

34.7.2 Intel® SGX Resource Enumeration Leaves

If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor also supports querying CPUID with EAX=12H on Intel SGX resource capability and configuration. The number of available sub-leaves in leaf 12H depends on the Opt-in and system software configuration. Information returned by CPUID.12H is thread specific; software should not assume that if Intel SGX instructions are supported on one hardware thread, they are also supported elsewhere.

A properly configured processor exposes Intel SGX functionality with CPUID.EAX=12H reporting valid information (non-zero content) in three or more sub-leaves, see Table 34-5.

- CPUID.(EAX=12H, ECX=0H) enumerates Intel SGX capability, including enclave instruction opcode support.
- CPUID.(EAX=12H, ECX=1H) enumerates Intel SGX capability of processor state configuration and enclave configuration in the SECS structure (see Table 35-3).
- CPUID.(EAX=12H, ECX >1) enumerates available EPC resources.

### Table 34-4. Intel® SGX Opt-in and Enabling Behavior

<table>
<thead>
<tr>
<th>CPUID.(07H,0H):EBX.SGX</th>
<th>CPUID.(12H)</th>
<th>FEATURE_CONTROL.LOCK</th>
<th>FEATURE_CONTROL.SGX_ENABLE</th>
<th>Enclave Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invalid</td>
<td>X</td>
<td>X</td>
<td>#UD</td>
</tr>
<tr>
<td>1</td>
<td>Valid*</td>
<td>X</td>
<td>X</td>
<td>#UD**</td>
</tr>
<tr>
<td>1</td>
<td>Valid*</td>
<td>0</td>
<td>X</td>
<td>#GP</td>
</tr>
<tr>
<td>1</td>
<td>Valid*</td>
<td>1</td>
<td>0</td>
<td>#GP</td>
</tr>
<tr>
<td>1</td>
<td>Valid*</td>
<td>1</td>
<td>1</td>
<td>Available (see Table 34-5 for details of SGX1 and SGX2).</td>
</tr>
</tbody>
</table>

* Leaf 12H enumeration results are dependent on enablement.
** See list of conditions in the #UD section of the reference pages of ENCLS and ENCLU

### Table 34-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities

<table>
<thead>
<tr>
<th>CPUID.(EAX=12H,ECX=0)</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register</strong></td>
<td><strong>Bits</strong></td>
</tr>
<tr>
<td><strong>EAX</strong></td>
<td><strong>Bits</strong></td>
</tr>
<tr>
<td>0</td>
<td>SGX1: If 1, indicates leaf functions of SGX1 instruction listed in Table 34-1 are supported.</td>
</tr>
<tr>
<td>1</td>
<td>SGX2: If 1, indicates leaf functions of SGX2 instruction listed in Table 34-2 are supported.</td>
</tr>
<tr>
<td>4:2</td>
<td>Reserved (0)</td>
</tr>
<tr>
<td>5</td>
<td>OVERSUB: If 1, indicates Intel SGX supports instructions: EINCVIRTCHILD, EDECVIRTCHILD, and ESETCONTEXT.</td>
</tr>
<tr>
<td>6</td>
<td>OVERSUB: If 1, indicates Intel SGX supports instructions: ETRACKC, ERDINFO, ELDBC, and ELDUC.</td>
</tr>
<tr>
<td>31:7</td>
<td>Reserved (0)</td>
</tr>
<tr>
<td><strong>EBX</strong></td>
<td><strong>Bits</strong></td>
</tr>
<tr>
<td>31:0</td>
<td>MISCELECT: Reports the bit vector of supported extended features that can be written to the MISC region of the SSA.</td>
</tr>
<tr>
<td><strong>ECX</strong></td>
<td><strong>Bits</strong></td>
</tr>
<tr>
<td>31:0</td>
<td>Reserved (0).</td>
</tr>
</tbody>
</table>
INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

On processors that support Intel SGX1 and SGX2, CPUID leaf 12H sub-leaf 2 report physical memory resources available for use with Intel SGX. These physical memory sections are typically allocated by BIOS as Processor Reserved Memory, and available to the OS to manage as EPC.

To enumerate how many EPC sections are available to the EPC manager, software can enumerate CPUID leaf 12H with sub-leaf index starting from 2, and decode the sub-leaf-type encoding (returned in EAX[3:0]) until the sub-leaf type is invalid. All invalid sub-leaves of CPUID leaf 12H return EAX/EBX/ECX/EDX with 0.

### Table 34-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>7:0</td>
<td>MaxEnclaveSize_Not64: the maximum supported enclave size is (2^{(EDX[7:0])}) bytes when not in 64-bit mode.</td>
</tr>
<tr>
<td>EDX</td>
<td>15:8</td>
<td>MaxEnclaveSize_64: the maximum supported enclave size is (2^{(EDX[15:8])}) bytes when operating in 64-bit mode.</td>
</tr>
<tr>
<td>EDX</td>
<td>31:16</td>
<td>Reserved (0).</td>
</tr>
</tbody>
</table>

### Table 34-6. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>31:0</td>
<td>Report the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE. SECS.ATTRIBUTES[n] can be set to 1 using ECREATE only if EAX[n] is 1, where n &lt; 32.</td>
</tr>
<tr>
<td>EAX</td>
<td>19:0</td>
<td>If EAX[3:0] = 0001b, these are bits 51:32 of the physical address of the base of the EPC section.</td>
</tr>
<tr>
<td>EBS</td>
<td>31:0</td>
<td>Report the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE. SECS.ATTRIBUTES[n+64] can be set to 1 using ECREATE only if ECX[n] is 1, where n &lt; 32.</td>
</tr>
<tr>
<td>ECX</td>
<td>31:0</td>
<td>Report the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE. SECS.ATTRIBUTES[n+96] can be set to 1 using ECREATE only if EDX[n] is 1, where n &lt; 32.</td>
</tr>
</tbody>
</table>

On processors that support Intel SGX1 and SGX2, CPUID leaf 12H sub-leaf 2 report physical memory resources available for use with Intel SGX. These physical memory sections are typically allocated by BIOS as Processor Reserved Memory, and available to the OS to manage as EPC.

To enumerate how many EPC sections are available to the EPC manager, software can enumerate CPUID leaf 12H with sub-leaf index starting from 2, and decode the sub-leaf-type encoding (returned in EAX[3:0]) until the sub-leaf type is invalid. All invalid sub-leaves of CPUID leaf 12H return EAX/EBX/ECX/EDX with 0.

### Table 34-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>3:0</td>
<td>0000b: This sub-leaf is invalid; EDX:ECX:EBX:EAX return 0. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other encodings are reserved.</td>
</tr>
<tr>
<td>EAX</td>
<td>11:4</td>
<td>Reserved (enumerate 0).</td>
</tr>
<tr>
<td>EAX</td>
<td>31:12</td>
<td>If EAX[3:0] = 0001b, these are bits 31:12 of the physical address of the base of the EPC section.</td>
</tr>
<tr>
<td>EBX</td>
<td>19:0</td>
<td>If EAX[30] = 0001b, these are bits 51:32 of the physical address of the base of the EPC section.</td>
</tr>
<tr>
<td>EBX</td>
<td>31:20</td>
<td>Reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>3:0</td>
<td>If ECX[30] = 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If ECX[30] = 0001b, then this section has confidentiality and integrity protection. If ECX[30] = 0010b, then this section has confidentiality protection only. All other encodings are reserved.</td>
</tr>
<tr>
<td>ECX</td>
<td>11:4</td>
<td>Reserved (enumerate 0).</td>
</tr>
<tr>
<td>ECX</td>
<td>31:12</td>
<td>If EAX[30] = 0001b, these are bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.</td>
</tr>
</tbody>
</table>
34.8 INTEL® SGX INTERACTIONS WITH CONTROL-FLOW ENFORCEMENT TECHNOLOGY

This section discusses extensions to the Intel SGX architecture to support CET.

34.8.1 CET in Enclaves Model

Each enclave has its private configuration for CET that is not shared with the CET configurations of the enclosing application. On entry into the enclave, the CET state of the enclosing application is saved into scratchpad registers inside the processor and the CET state of the enclave is established. On an asynchronous exit, the enclave CET state is saved into the enclave state save area frame. On exit from the enclave, the CET state of the enclosing application is re-established from the scratchpad registers.

A new page type, PT_SS_FIRST, is used to denote pages in an enclave that can be used as a first page of a shadow stack.

A new page type, PT_SS_REST, is used to denote pages in an enclave that can be used as a non-first page of a shadow stack.

A page denoted as PT_SS_FIRST and PT_SS_REST will be a legal target for shadow_stack_load, shadow_stack_store, and regular load operations. Regular stores will be disallowed to such pages. A PT_SS_FIRST/PT_SS_REST page must be writeable in the IA page tables and in EPT.

When in enclave mode, shadow_stack_load and shadow_stack_store operations must be to addresses in the enclave ELRANGE.

The EAUG instruction is extended to allocate pages of type PT_SS_FIRST/PT_SS_REST; this page type requires specifying a SECINFO structure with page parameters. Shadow page permission must be R/W. Regular R/W pages may continue to be allocated by providing a SECINFO pointer value of 0. Regular R/W pages may also be allocated by providing a SECINFO structure that specifies the page parameters. The EAUG instruction creates a shadow-stack-restore token at offset 0xFF8 on a PT_SS_FIRST page. This allows a dynamically created shadow stack to be restored using the RSTORSSP instruction. The EADD and EAUG instructions disallow creation of a PT_SS_FIRST or PT_SS_REST page as the first or last page in ELRANGE.

The EADD instruction requires that the PT_SS_REST page be all zeroes. The EADD instruction requires that a PT_SS_FIRST page be all zeroes except the 8 bytes at offset 0xFF8 on that page that must have a shadow-stack-restore token. This shadow-stack-restore token must have a linear address which is the linear address of the PT_SS_FIRST page + 4096. As an enclave could be loaded at varying linear addresses, the enclave builder should not extend the measurement of the PT_SS_FIRST pages into the measurement registers. On first entry on to the enclave using a TCS, the enclave software can use the RSTORSSP instruction to restore its SSP. Subsequent to performing a RSTORSSP, the enclave software can use the INCSSP instruction to pop the previous-ssp token that is created by the RSTORSSP instruction at the top of the restored shadow stack.

On an enclave entry, the SSP will be initialized to the value in a new TCS field called PREVSSP. The PREVSSP field is written with the value of SSP on enclave exit and is loaded into SSP at enclave entry. When a TCS page is added using EADD or accepted using EACCEPT, the processor requires the PREVSSP field to be initialized to 0.
34.8.2 Operations Not Supported on Shadow Stack Pages

The following operations are not allowed on pages of type PT_SS_FIRST and PT_SS_REST:

- EACCEPTCOPY
- EMODPR
- EMODPE

34.8.3 Indirect Branch Tracking – Legacy Compatibility Treatment

The legacy code page bitmap is tested using the page offset within the ELRANGE instead of the absolute linear address of the address where ENDBRANCH was missed; see the detailed algorithm in Section 17.3.6, “Legacy Compatibility Treatment,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for additional details.
35.1 OVERVIEW OF ENCLAVE EXECUTION ENVIRONMENT

When an enclave is created, it has a range of linear addresses to which the processor applies enhanced access control. This range is called the ELRANGE (see Section 34.3). When an enclave generates a memory access, the existing IA32 segmentation and paging architecture are applied. Additionally, linear addresses inside the ELRANGE must map to an EPC page otherwise when an enclave attempts to access that linear address a fault is generated. The EPC pages need not be physically contiguous. System software allocates EPC pages to various enclaves. Enclaves must abide by OS/VMM imposed segmentation and paging policies. OS/VMM-managed page tables and extended page tables provide address translation for the enclave pages. Hardware requires that these pages are properly mapped to EPC (any failure generates an exception).

Enclave entry must happen through specific enclave instructions:
- ENCLU[EENTER], ENCLU[ERESUME].

Enclave exit must happen through specific enclave instructions or events:
- ENCLU[EEXIT], Asynchronous Enclave Exit (AEX).

Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations. As an example a read of an enclave page may result in the return of all one’s or return of cyphertext of the cache line. Writing to an enclave page may result in a dropped write or a machine check at a later time. The processor will provide the protections as described in Section 35.4 and Section 35.5 on such accesses.

35.2 TERMINOLOGY

A memory access to the ELRANGE and initiated by an instruction executed by an enclave is called a Direct Enclave Access (Direct EA).

Memory accesses initiated by certain Intel® SGX instruction leaf functions such as ECREATE, EADD, EDBGRD, EDBGWR, ELDU/ELDB, EWB, ERMOVE, EEENTER, and ERESUME to EPC pages are called Indirect Enclave Accesses (Indirect EA). Table 35-1 lists additional details of the indirect EA of SGX1 and SGX2 extensions.

Direct EAs and Indirect EAs together are called Enclave Accesses (EAs).

Any memory access that is not an Enclave Access is called a non-enclave access.

35.3 ACCESS-CONTROL REQUIREMENTS

Enclave accesses have the following access-control attributes:
- All memory accesses must conform to segmentation and paging protection mechanisms.
- Code fetches from inside an enclave to a linear address outside that enclave result in a #GP(0) exception.
- Shadow-stack-load or shadow-stack-store from inside an enclave to a linear address outside that enclave results in a #GP(0) exception.
- Non-enclave accesses to EPC memory result in undefined behavior. EPC memory is protected as described in Section 35.4 and Section 35.5 on such accesses.
- EPC pages of page types PT_REG, PT_TCS, and PT_TRIM must be mapped to ELRANGE at the linear address specified when the EPC page was allocated to the enclave using ENCLS[EADD] or ENCLS[EAUG] leaf functions. Enclave accesses through other linear address result in a #PF with the PFEC.SGX bit set.
• Direct EAs to any EPC pages must conform to the currently defined security attributes for that EPC page in the EPCM. These attributes may be defined at enclave creation time (EADD) or when the enclave sets them using SGX2 instructions. The failure of these checks results in a #PF with the PFEC.SGX bit set.
  — Target page must belong to the currently executing enclave.
  — Data may be written to an EPC page if the EPCM allow write access.
  — Data may be read from an EPC page if the EPCM allow read access.
  — Instruction fetches from an EPC page are allowed if the EPCM allows execute access.
  — Shadow-stack-load from an EPC page and shadow-stack-store to an EPC page are allowed only if the page type is PT_SS_FIRST or PT_SS_REST.
  — Data writes that are not shadow-stack-store are not allowed if the EPCM page type is PT_SS_FIRST or PT_SS_REST.
  — Target page must not have a restricted page type (PT_SECS, PT_TCS, PT_VA, or PT_TRIM).
  — The EPC page must not be BLOCKED.
  — The EPC page must not be PENDING.
  — The EPC page must not be MODIFIED.

35.4 SEGMENT-BASED ACCESS CONTROL

Intel SGX architecture does not modify the segment checks performed by a logical processor. All memory accesses arising from a logical processor in protected mode (including enclave access) are subject to segmentation checks with the applicable segment register.

To ensure that outside entities do not modify the enclave's logical-to-linear address translation in an unexpected fashion, ENCLU[EENTER] and ENCLU[ERESUME] check that CS, DS, ES, and SS, if usable (i.e., not null), have segment base value of zero. A non-zero segment base value for these registers results in a #GP(0).

On enclave entry either via EENTER or ERESUME, the processor saves the contents of the external FS and GS registers, and loads these registers with values stored in the TCS at build time to enable the enclave's use of these registers for accessing the thread-local storage inside the enclave. On EEXIT and AEX, the contents at time of entry are restored. On AEX, the values of FS and GS are saved in the SSA frame. On ERESUME, FS and GS are restored from the SSA frame. The details of these operations can be found in the descriptions of EENTER, ERESUME, EEXIT, and AEX flows.

35.5 PAGE-BASED ACCESS CONTROL

35.5.1 Access-control for Accesses that Originate from Non-SGX Instructions

Intel SGX builds on the processor’s paging mechanism to provide page-granular access-control for enclave pages. Enclave pages are designed to be accessible only from inside the currently executing enclave if they belong to that enclave. In addition, enclave accesses must conform to the access control requirements described in Section 35.3 or through certain Intel SGX instructions. Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations.

35.5.2 Memory Accesses that Split Across ELRANGE

Memory data accesses are allowed to split across ELRANGE (i.e., a part of the access is inside ELRANGE and a part of the access is outside ELRANGE) while the processor is inside an enclave. If an access splits across ELRANGE, the

---

1. EPCM may allow write, read or execute access only for pages with page type PT_REG.
35.5.3 Implicit vs. Explicit Accesses

Memory accesses originating from Intel SGX instruction leaf functions are categorized as either explicit accesses or implicit accesses. Table 35-1 lists the implicit and explicit memory accesses made by Intel SGX leaf functions.

35.5.3.1 Explicit Accesses

Accesses to memory locations provided as explicit operands to Intel SGX instruction leaf functions, or their linked data structures are called explicit accesses.

Explicit accesses are always made using logical addresses. These accesses are subject to segmentation, paging, extended paging, and APIC-virtualization checks, and trigger any faults/exit associated with these checks when the access is made.

The interaction of explicit memory accesses with data breakpoints is leaf-function-specific, and is documented in Section 40.3.4.

35.5.3.2 Implicit Accesses

Accesses to data structures whose physical addresses are cached by the processor are called implicit accesses. These addresses are not passed as operands of the instruction but are implied by use of the instruction.

These accesses do not trigger any access-control faults/ exits or data breakpoints. Table 35-1 lists memory objects that Intel SGX instruction leaf functions access either by explicit access or implicit access. The addresses of explicit access objects are passed via register operands with the second through fourth column of Table 35-1 matching implicitly encoded registers RBX, RCX, RDX.

Physical addresses used in different implicit accesses are cached via different instructions and for different durations. The physical address of SECS associated with each EPC page is cached at the time the page is added to the enclave via ENCLS[EADD] or ENCLS[EAUG], or when the page is loaded to EPC via ENCLS[ELDB] or ENCLS[ELDU]. This binding is severed when the corresponding page is removed from the EPC via ENCLS[EREMOVE] or ENCLS[EWB]. Physical addresses of TCS and SSA pages are cached at the time of most-recent enclave entry. Exit from an enclave (ENCLU[EEXIT] or AEX) flushes this caching. Details of Asynchronous Enclave Exit is described in Chapter 37.

The physical addresses that are cached for use by implicit accesses are derived from logical (or linear) addresses after checks such as segmentation, paging, EPT, and APIC virtualization checks. These checks may trigger exceptions or VM exits. Note, however, that such exception or VM exits may not occur after a physical address is cached and used for an implicit access.

<table>
<thead>
<tr>
<th>Instr. Leaf</th>
<th>Enum.</th>
<th>Explicit 1</th>
<th>Explicit 2</th>
<th>Explicit 3</th>
<th>Implicit</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCEPT</td>
<td>SGX2</td>
<td>SECINFO</td>
<td>EPCPAGE</td>
<td></td>
<td>SECS</td>
</tr>
<tr>
<td>EACCEPTCOPY</td>
<td>SGX2</td>
<td>SECINFO</td>
<td>EPCPAGE</td>
<td>EPCPAGE</td>
<td>SECS</td>
</tr>
<tr>
<td>EADD</td>
<td>SGX1</td>
<td>PAGEINFO and linked structures</td>
<td>EPCPAGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAUG</td>
<td>SGX2</td>
<td>PAGEINFO and linked structures</td>
<td>EPCPAGE</td>
<td></td>
<td>SECS</td>
</tr>
<tr>
<td>EBLOCK</td>
<td>SGX1</td>
<td>EPCPAGE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECREATE</td>
<td>SGX1</td>
<td>PAGEINFO and linked structures</td>
<td>EPCPAGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDBGWR</td>
<td>SGX1</td>
<td>EPCADDR</td>
<td>Destination</td>
<td>SECS</td>
<td></td>
</tr>
<tr>
<td>EDECVIRTCHILD</td>
<td>OVERSUB</td>
<td>EPCPAGE</td>
<td></td>
<td>SECS</td>
<td></td>
</tr>
<tr>
<td>EENTER</td>
<td>SGX1</td>
<td>TCS and linked SSA</td>
<td></td>
<td>SECS</td>
<td></td>
</tr>
</tbody>
</table>

Table 35-1. List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions
35.6 Intel® SGX Data Structures Overview

Enclave operation is managed via a collection of data structures. Many of the top-level data structures contain sub-structures. The top-level data structures relate to parameters that may be used in enclave setup/maintenance, by Intel SGX instructions, or AEX event. The top-level data structures are:

- SGX Enclave Control Structure (SECS)
- Thread Control Structure (TCS)
- State Save Area (SSA)
- Page Information (PAGEINFO)
- Security Information (SECINFO)
- Paging Crypto MetaData (PCMD)
- Enclave Signature Structure (SIGSTRUCT)
- EINIT Token Structure (EINITTOKEN)
- Report Structure (REPORT)
- Report Target Info (TARGETINFO)
- Key Request (KEYREQUEST)
- Version Array (VA)
- Enclave Page Cache Map (EPCM)
- Read Info (RDINFO)
Details of the top-level data structures and associated sub-structures are listed in Section 35.7 through Section 35.20.

### 35.7 SGX ENCLAVE CONTROL STRUCTURE (SECS)

The SECS data structure requires 4K-Bytes alignment.

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE</td>
<td>0</td>
<td>8</td>
<td>Size of enclave in bytes; must be power of 2.</td>
</tr>
<tr>
<td>BASEADDR</td>
<td>8</td>
<td>8</td>
<td>Enclave Base Linear Address must be naturally aligned to size.</td>
</tr>
<tr>
<td>SSAFRAMESIZE</td>
<td>16</td>
<td>4</td>
<td>Size of one SSA frame in pages, including XSAVE, pad, GPR, and MISC (if CPUID.(EAX=12H, ECX=0):EDX != 0).</td>
</tr>
<tr>
<td>MISCELECT</td>
<td>20</td>
<td>4</td>
<td>Bit vector specifying which extended features are saved to the MISC region (see Section 35.7.2) of the SSA frame when an AEX occurs.</td>
</tr>
<tr>
<td>CET_LEG_BITMAP _OFFSET</td>
<td>24</td>
<td>8</td>
<td>Page aligned offset of legacy code page bitmap from enclave base. Software is expected to program this offset such that the entire bitmap resides in the ELRANGE when legacy compatibility mode for indirect branch tracking is enabled. However this is not enforced by the hardware. This field exists when CPUID.(EAX=7, ECX=0):EDX.CET_IBT[bit 20] is enumerated as 1, else it is reserved.</td>
</tr>
<tr>
<td>CET_ATTRIBUTES</td>
<td>32</td>
<td>1</td>
<td>CET feature attributes of the enclave; see Table 35-5. This field exists when CPUID.(EAX=12,ECX=1):EAX[6] is enumerated as 1, else it is reserved.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>33</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>48</td>
<td>16</td>
<td>Attributes of the Enclave, see Table 35-3.</td>
</tr>
<tr>
<td>MRENCLAVE</td>
<td>64</td>
<td>32</td>
<td>Measurement Register of enclave build process. See SIGSTRUCT for format.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>96</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>MRSIGNER</td>
<td>128</td>
<td>32</td>
<td>Measurement Register extended with the public key that verified the enclave. See SIGSTRUCT for format.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>160</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>CONFIGID</td>
<td>192</td>
<td>64</td>
<td>Post EINIT configuration identity.</td>
</tr>
<tr>
<td>ISVPRODID</td>
<td>256</td>
<td>2</td>
<td>Product ID of enclave.</td>
</tr>
<tr>
<td>ISVSVN</td>
<td>258</td>
<td>2</td>
<td>Security version number (SVN) of the enclave.</td>
</tr>
<tr>
<td>CONFIGSVN</td>
<td>260</td>
<td>2</td>
<td>Post EINIT configuration security version number (SVN).</td>
</tr>
<tr>
<td>RESERVED</td>
<td>262</td>
<td>3834</td>
<td>The RESERVED field consists of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• EID: An 8 byte Enclave Identifier. Its location is implementation specific.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• PAD: A 352 bytes padding pattern from the Signature (used for key derivation strings). It’s location is implementation specific.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• VIRTUALCHILDCONT: An 8 byte Count of virtual children that have been paged out by a VMM. Its location is implementation specific.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ENCLAVECONTEXT: An 8 byte Enclave context pointer. Its location is implementation specific.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ISVFAMILYID: A 16 byte value assigned to identify the family of products the enclave belongs to.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ISVEXTPRODID: A 16 byte value assigned to identify the product identity of the enclave.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The remaining 3226 bytes are reserved area. The entire 3834 byte field must be cleared prior to executing ECREATE.</td>
</tr>
</tbody>
</table>
35.7.1 ATTRIBUTES

The ATTRIBUTES data structure is comprised of bit-granular fields that are used in the SECS, the REPORT and the KEYREQUEST structures. CPUID.(EAX=12H, ECX=1) enumerates a bitmap of permitted 1-setting of bits in ATTRIBUTES.

Table 35-3. Layout of ATTRIBUTES Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>0</td>
<td>This bit indicates if the enclave has been initialized by EINIT. It must be cleared when loaded as part of ECREATE. For EREPORT instruction, TARGET_INFO.ATTRIBUTES[ENIT] must always be 1 to match the state after EINIT has initialized the enclave.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>1</td>
<td>If 1, the enclave permit debugger to read and write enclave data using EDBGRD and EDBGWR.</td>
</tr>
<tr>
<td>MODE64BIT</td>
<td>2</td>
<td>Enclave runs in 64-bit mode.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>3</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>PROVISIONKEY</td>
<td>4</td>
<td>Provisioning Key is available from EGETKEY.</td>
</tr>
<tr>
<td>EINITTOKEN_KEY</td>
<td>5</td>
<td>EINIT token key is available from EGETKEY.</td>
</tr>
<tr>
<td>CET</td>
<td>6</td>
<td>Enable CET attributes. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0 this bit is reserved and must be 0.</td>
</tr>
<tr>
<td>KSS</td>
<td>7</td>
<td>Key Separation and Sharing Enabled.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>9:8</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>AEXNOTIFY</td>
<td>10</td>
<td>The bit indicates that threads within the enclave may receive AEX notifications.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>63:11</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>XFRM</td>
<td>127:64</td>
<td>XSAVE Feature Request Mask. See Section 39.7.</td>
</tr>
</tbody>
</table>

35.7.2 SECS.MISCSELECT Field

CPUID.(EAX=12H, ECX=0):EBX[31:0] enumerates which extended information that the processor can save into the MISC region of SSA when an AEX occurs. An enclave writer can specify via SIGSTRUCT how to set the SECS.MISCSELECT field. The bit vector of MISCSELECT selects which extended information is to be saved in the MISC region of the SSA frame when an AEX is generated. The bit vector definition of extended information is listed in Table 35-4.

If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, SECS.MISCSELECT field must be all zeros.

The SECS.MISCSELECT field determines the size of MISC region of the SSA frame, see Section 35.9.2.

Table 35-4. Bit Vector Layout of MISCSELECT Field of Extended Information

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXINFO</td>
<td>0</td>
<td>Report information about page fault and general protection exception that occurred inside an enclave.</td>
</tr>
<tr>
<td>CPINFO</td>
<td>1</td>
<td>Report information about control protection exception that occurred inside an enclave. When CPUID.(EAX=12H, ECX=0):EBX[1] is 0, this bit is reserved.</td>
</tr>
<tr>
<td>Reserved</td>
<td>31:2</td>
<td>Reserved (0).</td>
</tr>
</tbody>
</table>

35.7.3 SECS.CET_ATTRIBUTES Field

The SECS.CET_ATTRIBUTES field can be used by the enclave writer to enable various CET attributes in an enclave. This field exists when CPUID.(EAX=12, ECX=1):EAX[6] is enumerated as 1. Bits 1:0 are defined when CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1, and bits 5:2 are defined when CPUID.(EAX=7, ECX=0):EDX.CET_IBT is 1.
### 35.8 THREAD CONTROL STRUCTURE (TCS)

Each executing thread in the enclave is associated with a Thread Control Structure. It requires 4K-Bytes alignment.

#### Table 35-5. Bit Vector Layout of CET_ATTRIBUTES Field of Extended Information

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH_STK_EN</td>
<td>0</td>
<td>When set to 1, enable shadow stacks.</td>
</tr>
<tr>
<td>WR_SHSTK_EN</td>
<td>1</td>
<td>When set to 1, enables the WRSS[D,Q]W instructions.</td>
</tr>
<tr>
<td>ENDBR_EN</td>
<td>2</td>
<td>When set to 1, enables indirect branch tracking.</td>
</tr>
<tr>
<td>LEG_IW_EN</td>
<td>3</td>
<td>Enable legacy compatibility treatment for indirect branch tracking.</td>
</tr>
<tr>
<td>NO_TRACK_EN</td>
<td>4</td>
<td>When set to 1, enables use of no-track prefix for indirect branch tracking.</td>
</tr>
<tr>
<td>SUPPRESS_DIS</td>
<td>5</td>
<td>When set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.</td>
</tr>
<tr>
<td>Reserved</td>
<td>7:6</td>
<td>Reserved (0).</td>
</tr>
</tbody>
</table>

#### Table 35-6. Layout of Thread Control Structure (TCS)

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAGE</td>
<td>0</td>
<td>8</td>
<td>Enclave execution state of the thread controlled by this TCS. A value of 0 indicates that this TCS is available for enclave entry. A value of 1 indicates that a logical processor is currently executing an enclave in the context of this TCS.</td>
</tr>
<tr>
<td>FLAGS</td>
<td>8</td>
<td>8</td>
<td>The thread’s execution flags (see Section 35.8.1).</td>
</tr>
<tr>
<td>OSSA</td>
<td>16</td>
<td>8</td>
<td>Offset of the base of the State Save Area stack, relative to the enclave base. Must be page aligned.</td>
</tr>
<tr>
<td>CSSA</td>
<td>24</td>
<td>4</td>
<td>Current slot index of an SSA frame, cleared by EADD and EACCEPT.</td>
</tr>
<tr>
<td>NSSA</td>
<td>28</td>
<td>4</td>
<td>Number of available slots for SSA frames.</td>
</tr>
<tr>
<td>OENTRY</td>
<td>32</td>
<td>8</td>
<td>Offset in enclave to which control is transferred on EENTER relative to the base of the enclave.</td>
</tr>
<tr>
<td>AEP</td>
<td>40</td>
<td>8</td>
<td>The value of the Asynchronous Exit Pointer that was saved at EENTER time.</td>
</tr>
<tr>
<td>OFSBASE</td>
<td>48</td>
<td>8</td>
<td>Offset to add to the base address of the enclave for producing the base address of FS segment inside the enclave. Must be page aligned.</td>
</tr>
<tr>
<td>GOSBASE</td>
<td>56</td>
<td>8</td>
<td>Offset to add to the base address of the enclave for producing the base address of GS segment inside the enclave. Must be page aligned.</td>
</tr>
<tr>
<td>FSLIMIT</td>
<td>64</td>
<td>4</td>
<td>Size to become the new FS limit in 32-bit mode.</td>
</tr>
<tr>
<td>GSLIMIT</td>
<td>68</td>
<td>4</td>
<td>Size to become the new GS limit in 32-bit mode.</td>
</tr>
<tr>
<td>OCETSSA</td>
<td>72</td>
<td>8</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the offset of the CET state save area from enclave base. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.</td>
</tr>
<tr>
<td>PREVSSP</td>
<td>80</td>
<td>8</td>
<td>When CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] is 1, this field records the SSP at the time of AEX or EEXIT; used to setup SSP on entry. When CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] is 0, this field is reserved and must be 0.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>72</td>
<td>4024</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>
35.8.1 TCS.FLAGS

Table 35-7. Layout of TCS.FLAGS Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBGOPTIN</td>
<td>0</td>
<td>If set, allows debugging features (single-stepping, breakpoints, etc.) to be enabled and active while executing in the enclave on this TCS. Hardware clears this bit on EADD. A debugger may later modify it if the enclave’s ATTRIBUTES.DEBUG is set.</td>
</tr>
<tr>
<td>AEXNOTIFY</td>
<td>1</td>
<td>A thread that enters the enclave cannot receive AEX notifications unless this flag is set to 1.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>63:2</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>

35.8.2 State Save Area Offset (OSSA)

The OSSA points to a stack of State Save Area (SSA) frames (see Section 35.9) used to save the processor state when an interrupt or exception occurs while executing in the enclave.

35.8.3 Current State Save Area Frame (CSSA)

CSSA is the index of the current SSA frame that will be used by the processor to determine where to save the processor state on an interrupt or exception that occurs while executing in the enclave. It is an index into the array of frames addressed by OSSA. CSSA is incremented on an AEX and decremented on an ERESUME.

35.8.4 Number of State Save Area Frames (NSSA)

NSSA specifies the number of SSA frames available for this TCS. There must be at least one available SSA frame when EENTER-ing the enclave or the EENTER will fail.

35.9 STATE SAVE AREA (SSA) FRAME

When an AEX occurs while running in an enclave, the architectural state is saved in the thread’s current SSA frame, which is pointed to by TCS.CSSA. An SSA frame must be page aligned, and contains the following regions:

- The XSAVE region starts at the base of the SSA frame, this region contains extended feature register state in an XSAVE/FXSAVE-compatible non-compacted format.
- A Pad region: software may choose to maintain a pad region separating the XSAVE region and the MISC region. Software choose the size of the pad region according to the sizes of the MISC and GPRSGX regions.
- The GPRSGX region. The GPRSGX region is the last region of an SSA frame (see Table 35-8). This is used to hold the processor general purpose registers (RAX ... R15), the RIP, the outside RSP and RBP, RFLAGS, and the AEX information.
- The MISC region (If CPUIDEAX=12H, ECX=0):EBX[31:0] != 0). The MISC region is adjacent to the GRPSGX region, and may contain zero or more components of extended information that would be saved when an AEX occurs. If the MISC region is absent, the region between the GPRSGX and XSAVE regions is the pad region that software can use. If the MISC region is present, the region between the MISC and XSAVE regions is the pad region that software can use. See additional details in Section 35.9.2.
35.9.1 GPRSGX Region

The layout of the GPRSGX region is shown in Table 35-9.

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td>24</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td>32</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td>40</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RSI</td>
<td>48</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td>56</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>64</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>72</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>80</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>88</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>96</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>104</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>112</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td>120</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RFLAGS</td>
<td>128</td>
<td>8</td>
<td>Flag register.</td>
</tr>
<tr>
<td>RIP</td>
<td>136</td>
<td>8</td>
<td>Instruction pointer.</td>
</tr>
<tr>
<td>URSP</td>
<td>144</td>
<td>8</td>
<td>Non-Enclave (outside) stack pointer. Saved by EENTER, restored on AEX.</td>
</tr>
<tr>
<td>URBP</td>
<td>152</td>
<td>8</td>
<td>Non-Enclave (outside) RBP pointer. Saved by EENTER, restored on AEX.</td>
</tr>
<tr>
<td>EXITINFO</td>
<td>160</td>
<td>4</td>
<td>Contains information about exceptions that cause AEXs, which might be needed by enclave software (see Section 35.9.1.1).</td>
</tr>
<tr>
<td>RESERVED</td>
<td>164</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
35.9.1.1 EXITINFO

EXITINFO contains the information used to report exit reasons to software inside the enclave. It is a 4 byte field laid out as in Table 35-10. The VALID bit is set only for the exceptions conditions which are reported inside an enclave. See Table 35-11 for which exceptions are reported inside the enclave. If the exception condition is not one reported inside the enclave then VECTOR and EXIT_TYPE are cleared.

When a higher priority event, such as SMI, and a pending debug exception occur at the same time when executing inside an enclave, the higher priority event has precedence. As an example for an SMI, the SSA exit info is zero. The debug exception will be delivered upon return from the SMI. In such cases, the EXITINFO field will not contain the information of a debug exception.

### Table 35-10. Layout of EXITINFO Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR</td>
<td>7:0</td>
<td>Exception number of exceptions reported inside enclave.</td>
</tr>
<tr>
<td>EXIT_TYPE</td>
<td>10:8</td>
<td>011b: Hardware exceptions. 110b: Software exceptions. Other values: Reserved.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>30:11</td>
<td>Reserved as zero.</td>
</tr>
<tr>
<td>VALID</td>
<td>31</td>
<td>0: unsupported exceptions. 1: Supported exceptions. Includes two categories: • Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM. • Conditionally supported exception: — #PF, #GP if SECS.MISCSELECT.EXINFO = 1. — #CP if SECS.MISCSELECT.CPINFO=1.</td>
</tr>
</tbody>
</table>

35.9.1.2 VECTOR Field Definition

Table 35-11 contains the VECTOR field. This field contains information about some exceptions which occur inside the enclave. These vector values are the same as the values that would be used when vectoring into regular exception handlers. All values not shown are not reported inside an enclave.

### Table 35-11. Exception Vectors

<table>
<thead>
<tr>
<th>Name</th>
<th>Vector #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DE</td>
<td>0</td>
<td>Divider exception.</td>
</tr>
<tr>
<td>#DB</td>
<td>1</td>
<td>Debug exception.</td>
</tr>
<tr>
<td>#BP</td>
<td>3</td>
<td>Breakpoint exception.</td>
</tr>
<tr>
<td>#BR</td>
<td>5</td>
<td>Bound range exceeded exception.</td>
</tr>
<tr>
<td>#UD</td>
<td>6</td>
<td>Invalid opcode exception.</td>
</tr>
<tr>
<td>#GP</td>
<td>13</td>
<td>General protection exception. Only reported if SECS.MISCSELECT.EXINFO = 1.</td>
</tr>
<tr>
<td>#PF</td>
<td>14</td>
<td>Page fault exception. Only reported if SECS.MISCSELECT.EXINFO = 1.</td>
</tr>
</tbody>
</table>
35.9.2 MISC Region

The layout of the MISC region is shown in Table 35-12. The number of components that the processor supports in the MISC region corresponds to the bits of CPUID.(EAX=12H, ECX=0):EBX[31:0] set to 1. Each set bit in CPUID.(EAX=12H, ECX=0):EBX[31:0] has a defined size for the corresponding component, as shown in Table 35-12. Enclave writers need to do the following:

- Decide which MISC region components will be supported for the enclave.
- Allocate an SSA frame large enough to hold the components chosen above.
- Instruct each enclave builder software to set the appropriate bits in SECS.MISCSELECT.

The first component, EXINFO, starts next to the GPRSGX region. Additional components in the MISC region grow in ascending order within the MISC region towards the XSAVE region.

The size of the MISC region is calculated as follows:

- If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISC region is not supported.
- If CPUID.(EAX=12H, ECX=0):EBX[31:0] != 0, the size of MISC region is derived from the sum of the highest bit set in SECS.MISCSELECT and the size of the MISC component corresponding to that bit. Offset and size information of currently defined MISC components are listed in Table 35-12. For example, if the highest bit set in SECS.MISCSELECT is bit 0, the MISC region offset is OFFSET(GPRSGX)-16 and size is 16 bytes.
- The processor saves a MISC component i in the MISC region if and only if SECS.MISCSELECT[i] is 1.

### Table 35-12. Layout of MISC region of the State Save Area

<table>
<thead>
<tr>
<th>MISC Components</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXINFO</td>
<td>Offset(GPRSGX) -16</td>
<td>16</td>
<td>If CPUID.(EAX=12H, ECX=0):EBX[0] = 1, exception information on #GP or #PF that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[0] = 1. If CPUID.(EAX=12H, ECX=0):EBX[1] = 1, exception information on #CP that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[1] = 1.</td>
</tr>
</tbody>
</table>

35.9.2.1 EXINFO Structure

Table 35-13 contains the layout of the EXINFO structure that provides additional information.

### Table 35-13. Layout of EXINFO Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MADDR</td>
<td>0</td>
<td>8</td>
<td>If #PF: contains the page fault linear address that caused a page fault. If #GP: the field is cleared. If #CP: the field is cleared.</td>
</tr>
<tr>
<td>ERRCD</td>
<td>8</td>
<td>4</td>
<td>Exception error code for either #GP or #PF.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>12</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
35.9.2.2 Page Fault Error Code

Table 35-14 contains page fault error code that may be reported in EXINFO.ERRCD.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0</td>
<td>Same as non-SGX page fault exception P flag.</td>
</tr>
<tr>
<td>W/R</td>
<td>1</td>
<td>Same as non-SGX page fault exception W/R flag.</td>
</tr>
<tr>
<td>U/S</td>
<td>2</td>
<td>Always set to 1 (user mode reference).</td>
</tr>
<tr>
<td>RSVD</td>
<td>3</td>
<td>Same as non-SGX page fault exception RSVD flag.</td>
</tr>
<tr>
<td>I/D</td>
<td>4</td>
<td>Same as non-SGX page fault exception I/D flag.</td>
</tr>
<tr>
<td>PK</td>
<td>5</td>
<td>Protection Key induced fault.</td>
</tr>
<tr>
<td>RSVD</td>
<td>14:6</td>
<td>Reserved.</td>
</tr>
<tr>
<td>SGX</td>
<td>15</td>
<td>EPCM induced fault.</td>
</tr>
<tr>
<td>RSVD</td>
<td>31:5</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

NOTES:
1. Page faults incident to enclave mode that report U/S=0 are not reported in EXINFO.

35.10 CET STATE SAVE AREA FRAME

The CET state save area consists of an array of CET state save frames. The number of CET state save frames is equal to the TCS.NSSA. The current CET SSA frame is indicated by TCS.CSSA. The offset of the CET state save area is specified by TCS.OCETSSA.

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSP</td>
<td>0</td>
<td>8</td>
<td>Shadow Stack Pointer. This field is reserved when CPUID.(EAX=7, ECX=0):ECX[CET_SS] is 0.</td>
</tr>
<tr>
<td>IB_TRACK_STATE</td>
<td>8</td>
<td>8</td>
<td>Indirect branch tracker state: Bit 0: SUPPRESS - suppressed(1), tracking(0) Bit 1: TRACKER - IDLE (0), WAIT_FOR_ENDBRANCH (1) Bits 63:2 - Reserved This field is reserved when CPUID(EAX=7, ECX=0):EDX[CET_IBT] is 0.</td>
</tr>
</tbody>
</table>

35.11 PAGE INFORMATION (PAGEINFO)

PAGEINFO is an architectural data structure that is used as a parameter to the EPC-management instructions. It requires 32-Byte alignment.

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINADDR</td>
<td>0</td>
<td>8</td>
<td>Enclave linear address.</td>
</tr>
<tr>
<td>SRCPGE</td>
<td>8</td>
<td>8</td>
<td>Effective address of the page where contents are located.</td>
</tr>
<tr>
<td>SECINFO/PCMD</td>
<td>16</td>
<td>8</td>
<td>Effective address of the SECINFO or PCMD (for ELDU, ELDB, EwB) structure for the page.</td>
</tr>
<tr>
<td>SECS</td>
<td>24</td>
<td>8</td>
<td>Effective address of EPC slot that currently contains the SECS.</td>
</tr>
</tbody>
</table>
35.12 SECURITY INFORMATION (SECINFO)

The SECINFO data structure holds meta-data about an enclave page.

### Table 35-17. Layout of SECINFO Data Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAGS</td>
<td>0</td>
<td>8</td>
<td>Flags describing the state of the enclave page.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>8</td>
<td>56</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>

35.12.1 SECINFO.FLAGS

The SECINFO.FLAGS are a set of fields describing the properties of an enclave page.

### Table 35-18. Layout of SECINFO.FLAGS Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0</td>
<td>If 1 indicates that the page can be read from inside the enclave; otherwise the page cannot be read from inside the enclave.</td>
</tr>
<tr>
<td>W</td>
<td>1</td>
<td>If 1 indicates that the page can be written from inside the enclave; otherwise the page cannot be written from inside the enclave.</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>If 1 indicates that the page can be executed from inside the enclave; otherwise the page cannot be executed from inside the enclave.</td>
</tr>
<tr>
<td>PENDING</td>
<td>3</td>
<td>If 1 indicates that the page is in the PENDING state; otherwise the page is not in the PENDING state.</td>
</tr>
<tr>
<td>MODIFIED</td>
<td>4</td>
<td>If 1 indicates that the page is in the MODIFIED state; otherwise the page is not in the MODIFIED state.</td>
</tr>
<tr>
<td>PR</td>
<td>5</td>
<td>If 1 indicates that a permission restriction operation on the page is in progress, otherwise a permission restriction operation is not in progress.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>7:6</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>PAGE_TYPE</td>
<td>15:8</td>
<td>The type of page that the SECINFO is associated with.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>63:16</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>

35.12.2 PAGE_TYPE Field Definition

The SECINFO flags and EPC flags contain bits indicating the type of page.

### Table 35-19. Supported PAGE_TYPE

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT_SECS</td>
<td>0</td>
<td>Page is an SECS.</td>
</tr>
<tr>
<td>PT_TCS</td>
<td>1</td>
<td>Page is a TCS.</td>
</tr>
<tr>
<td>PT_REG</td>
<td>2</td>
<td>Page is a regular page.</td>
</tr>
<tr>
<td>PT_VA</td>
<td>3</td>
<td>Page is a Version Array.</td>
</tr>
<tr>
<td>PT_TRIM</td>
<td>4</td>
<td>Page is in trimmed state.</td>
</tr>
<tr>
<td>PT_SS_FIRST</td>
<td>5</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, Page is first page of a shadow stack. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this value is reserved.</td>
</tr>
<tr>
<td>PT_SS_REST</td>
<td>6</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, Page is not first page of a shadow stack. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this value is reserved.</td>
</tr>
<tr>
<td>All others</td>
<td></td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
35.13 **PAGING CRYPTO METADATA (PCMD)**

The PCMD structure is used to keep track of crypto meta-data associated with a paged-out page. Combined with PAGEINFO, it provides enough information for the processor to verify, decrypt, and reload a paged-out EPC page. The size of the PCMD structure (128 bytes) is architectural.

EWB calculates the Message Authentication Code (MAC) value and writes out the PCMD. ELDB/U reads the fields and checks the MAC.

The format of PCMD is as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECINFO</td>
<td>0</td>
<td>64</td>
<td>Flags describing the state of the enclave page; R/W by software.</td>
</tr>
<tr>
<td>ENCLAVEID</td>
<td>64</td>
<td>8</td>
<td>Enclave Identifier used to establish a cryptographic binding between paged-out page and enclave.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>72</td>
<td>40</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>MAC</td>
<td>112</td>
<td>16</td>
<td>Message Authentication Code for the page, page meta-data and reserved field.</td>
</tr>
</tbody>
</table>

**Table 35-20. Layout of PCMD Data Structure**

35.14 **ENCLAVE SIGNATURE STRUCTURE (SIGSTRUCT)**

SIGSTRUCT is a structure created and signed by the enclave developer that contains information about the enclave. SIGSTRUCT is processed by the EINIT leaf function to verify that the enclave was properly built.

SIGSTRUCT includes ENCLAVEHASH as SHA256 digest, as defined in FIPS PUB 180-4. The digests are byte strings of length 32. Each of the 8 HASH dwords is stored in little-endian order.

SIGSTRUCT includes four 3072-bit integers (MODULUS, SIGNATURE, Q1, Q2). Each such integer is represented as a byte strings of length 384, with the most significant byte at the position “offset + 383”, and the least significant byte at position “offset”.

The (3072-bit integer) SIGNATURE should be an RSA signature, where: a) the RSA modulus (MODULUS) is a 3072-bit integer; b) the public exponent is set to 3; c) the signing procedure uses the EMSA-PKCS1-v1.5 format with DER encoding of the “DigestInfo” value as specified in of PKCS#1 v2.1/RFC 3447.

The 3072-bit integers Q1 and Q2 are defined by:

\[ q_1 = \text{floor}(\text{Signature}^2 / \text{Modulus}); \]
\[ q_2 = \text{floor}((\text{Signature}^3 - q_1 \times \text{Signature} \times \text{Modulus}) / \text{Modulus}); \]

SIGSTRUCT must be page aligned

In column 5 of Table 35-21, ‘Y’ indicates that this field should be included in the signature generated by the developer.

**Table 35-21. Layout of Enclave Signature Structure (SIGSTRUCT)**

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER</td>
<td>0</td>
<td>16</td>
<td>Must be byte stream 060000000010000000000010000000000H</td>
<td>Y</td>
</tr>
<tr>
<td>VENDOR</td>
<td>16</td>
<td>4</td>
<td>Intel Enclave: 00008086H Non-Intel Enclave: 00000000H</td>
<td>Y</td>
</tr>
<tr>
<td>DATE</td>
<td>20</td>
<td>4</td>
<td>Build date is yyyyymmdd in hex: yyyy=4 digit year, mm=1-12, dd=1-31</td>
<td>Y</td>
</tr>
<tr>
<td>HEADER2</td>
<td>24</td>
<td>16</td>
<td>Must be byte stream 01010000060000000060000000010000000H</td>
<td>Y</td>
</tr>
<tr>
<td>SWDEFINED</td>
<td>40</td>
<td>4</td>
<td>Available for software use.</td>
<td>Y</td>
</tr>
</tbody>
</table>
35.15 **EINIT Token Structure (EINITTOKEN)**

The EINIT token is used by EINIT to verify that the enclave is permitted to launch. EINIT token is generated by an enclave in possession of the EINITTOKEN key (the Launch Enclave).

EINIT token must be 512-Byte aligned.

### Table 35-21. Layout of Enclave Signature Structure (SIGSTRUCT)

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESERVED</td>
<td>44</td>
<td>84</td>
<td>Must be zero.</td>
<td>Y</td>
</tr>
<tr>
<td>MODULUS</td>
<td>128</td>
<td>384</td>
<td>Module Public Key (keylength=3072 bits).</td>
<td>N</td>
</tr>
<tr>
<td>EXPONENT</td>
<td>512</td>
<td>4</td>
<td>RSA Exponent = 3.</td>
<td>N</td>
</tr>
<tr>
<td>SIGNATURE</td>
<td>516</td>
<td>384</td>
<td>Signature over Header and Body.</td>
<td>N</td>
</tr>
<tr>
<td>MISCELECT</td>
<td>900</td>
<td>4</td>
<td>Bit vector specifying Extended SSA frame feature set to be used.</td>
<td>Y</td>
</tr>
<tr>
<td>MISCMASK</td>
<td>904</td>
<td>4</td>
<td>Bit vector mask of MISCELECT to enforce.</td>
<td>Y</td>
</tr>
<tr>
<td>CET_ATTRIBUTES</td>
<td>908</td>
<td>1</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the Enclave CET attributes that must be set. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.</td>
<td>Y</td>
</tr>
<tr>
<td>CET_ATTRIBUTES_MASK</td>
<td>909</td>
<td>1</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the Mask of CET attributes to enforce. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.</td>
<td>Y</td>
</tr>
<tr>
<td>RESERVED</td>
<td>910</td>
<td>2</td>
<td>Must be zero.</td>
<td>Y</td>
</tr>
<tr>
<td>ISVFAMILYID</td>
<td>912</td>
<td>16</td>
<td>ISV assigned Product Family ID.</td>
<td>Y</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>928</td>
<td>16</td>
<td>Enclave Attributes that must be set.</td>
<td>Y</td>
</tr>
<tr>
<td>ATTRIBUTEMASK</td>
<td>944</td>
<td>16</td>
<td>Mask of Attributes to enforce.</td>
<td>Y</td>
</tr>
<tr>
<td>ENCLAVEHASH</td>
<td>960</td>
<td>32</td>
<td>MRENCLAVE of enclave this structure applies to.</td>
<td>Y</td>
</tr>
<tr>
<td>RESERVED</td>
<td>992</td>
<td>16</td>
<td>Must be zero.</td>
<td>Y</td>
</tr>
<tr>
<td>ISVEXTPRODID</td>
<td>1008</td>
<td>16</td>
<td>ISV assigned extended Product ID.</td>
<td>Y</td>
</tr>
<tr>
<td>ISVPRODID</td>
<td>1024</td>
<td>2</td>
<td>ISV assigned Product ID.</td>
<td>Y</td>
</tr>
<tr>
<td>ISVSVN</td>
<td>1026</td>
<td>2</td>
<td>ISV assigned SVN (security version number).</td>
<td>Y</td>
</tr>
<tr>
<td>RESERVED</td>
<td>1028</td>
<td>12</td>
<td>Must be zero.</td>
<td>N</td>
</tr>
<tr>
<td>Q1</td>
<td>1040</td>
<td>384</td>
<td>Q1 value for RSA Signature Verification.</td>
<td>N</td>
</tr>
<tr>
<td>Q2</td>
<td>1424</td>
<td>384</td>
<td>Q2 value for RSA Signature Verification.</td>
<td>N</td>
</tr>
</tbody>
</table>

**NOTES:**

1. If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISCELECT must be 0.
**Table 35-22. Layout of EINIT Token (EINITTOKEN)**

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>MACed</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>0</td>
<td>4</td>
<td>Y</td>
<td>Bit 0: 1: Valid; 0: Invalid. All other bits reserved.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>4</td>
<td>44</td>
<td>Y</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>48</td>
<td>16</td>
<td>Y</td>
<td>ATTRIBUTES of the Enclave.</td>
</tr>
<tr>
<td>MRENCLAVE</td>
<td>64</td>
<td>32</td>
<td>Y</td>
<td>MRENCLAVE of the Enclave.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>96</td>
<td>32</td>
<td>Y</td>
<td>Reserved.</td>
</tr>
<tr>
<td>MRSIGNER</td>
<td>128</td>
<td>32</td>
<td>Y</td>
<td>MRSIGNER of the Enclave.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>160</td>
<td>32</td>
<td>Y</td>
<td>Reserved.</td>
</tr>
<tr>
<td>CPUUSNLE</td>
<td>192</td>
<td>16</td>
<td>N</td>
<td>Launch Enclave’s CPUSVN.</td>
</tr>
<tr>
<td>ISVPRODIDLE</td>
<td>208</td>
<td>02</td>
<td>N</td>
<td>Launch Enclave’s ISVPRODID.</td>
</tr>
<tr>
<td>ISVSVNLE</td>
<td>210</td>
<td>02</td>
<td>N</td>
<td>Launch Enclave’s ISVSVN.</td>
</tr>
<tr>
<td>CET_MASKED_ATTRIBUTES_LE</td>
<td>212</td>
<td>1</td>
<td>N</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the Launch enclaves masked CET attributes. This should be set to LE’s CET_ATTRIBU-</td>
</tr>
<tr>
<td>RESERVED</td>
<td>213</td>
<td>23</td>
<td>N</td>
<td>Reserved.</td>
</tr>
<tr>
<td>MASKEDMISCSEL ECTLE</td>
<td>236</td>
<td>4</td>
<td></td>
<td>Launch Enclave’s MASKEDMISCSELECT: set by the LE to the resolved MISCESELECT value, used by EGETKEY (after applying KEYREQUEST’s masking).</td>
</tr>
<tr>
<td>MASKEDATTRIBUTES_LE</td>
<td>240</td>
<td>16</td>
<td>N</td>
<td>Launch Enclave’s MASKEDATTRIBUTES: This should be set to the LE’s ATTRIBUTES masked with ATTRIBUTEMASK of the LE’s KEYREQUEST.</td>
</tr>
<tr>
<td>KEYID</td>
<td>256</td>
<td>32</td>
<td>N</td>
<td>Value for key wear-out protection.</td>
</tr>
<tr>
<td>MAC</td>
<td>288</td>
<td>16</td>
<td>N</td>
<td>Message Authentication Code on EINITTOKEN using EINITTOKEN_KEY.</td>
</tr>
</tbody>
</table>

**35.16 REPORT (REPORT)**

The REPORT structure is the output of the EREPORT instruction, and must be 512-Byte aligned.

**Table 35-23. Layout of REPORT**

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUSVN</td>
<td>0</td>
<td>16</td>
<td>The security version number of the processor.</td>
</tr>
<tr>
<td>MISCESELECT</td>
<td>16</td>
<td>4</td>
<td>Bit vector specifying which extended features are saved to the MISC region of the SSA frame when an AEX occurs.</td>
</tr>
<tr>
<td>CET_ATTRIBUTES</td>
<td>20</td>
<td>1</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field reports the CET_ATTRIBU-</td>
</tr>
<tr>
<td>RESERVED</td>
<td>21</td>
<td>11</td>
<td>Zero.</td>
</tr>
<tr>
<td>ISVEXTNPRODID</td>
<td>32</td>
<td>16</td>
<td>The value of SECS.ISVEXTNPRODID.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>48</td>
<td>16</td>
<td>ATTRIBUTES of the Enclave. See Section 35.7.1.</td>
</tr>
<tr>
<td>MRENCLAVE</td>
<td>64</td>
<td>32</td>
<td>The value of SECS.MRENCLAVE.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>96</td>
<td>32</td>
<td>Zero.</td>
</tr>
<tr>
<td>MRSIGNER</td>
<td>128</td>
<td>32</td>
<td>The value of SECS.MRSIGNER.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>160</td>
<td>32</td>
<td>Zero.</td>
</tr>
</tbody>
</table>
35.16.1 REPORTDATA

REPORTDATA is a 64-Byte data structure that is provided by the enclave and included in the REPORT. It can be used to securely pass information from the enclave to the target enclave.

35.17 REPORT TARGET INFO (TARGETINFO)

This structure is an input parameter to the EREPORT leaf function. The address of TARGETINFO is specified as an effective address in RBX. It is used to identify the target enclave which will be able to cryptographically verify the REPORT structure returned by EREPORT. TARGETINFO must be 512-Byte aligned.

### Table 35-24. Layout of TARGETINFO Data Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEASUREMENT</td>
<td>0</td>
<td>32</td>
<td>The MRENCLAVE of the target enclave.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>32</td>
<td>16</td>
<td>The ATTRIBUTES field of the target enclave.</td>
</tr>
<tr>
<td>CET_ATTRIBUTES</td>
<td>48</td>
<td>1</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides the CET_ATTRIBUTES field of the target enclave. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>49</td>
<td>1</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>CONFIGSVN</td>
<td>50</td>
<td>2</td>
<td>CONFIGSVN of the target enclave.</td>
</tr>
<tr>
<td>MISCESELECT</td>
<td>52</td>
<td>4</td>
<td>The MISCESELECT of the target enclave.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>56</td>
<td>8</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>CONFIGID</td>
<td>64</td>
<td>64</td>
<td>CONFIGID of target enclave.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>128</td>
<td>384</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>

35.18 KEY REQUEST (KEYREQUEST)

This structure is an input parameter to the EGETKEY leaf function. It is passed in as an effective address in RBX and must be 512-Byte aligned. It is used for selecting the appropriate key and any additional parameters required in the derivation of that key.
### Table 35-25. Layout of KEYREQUEST Data Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYNAME</td>
<td>0</td>
<td>2</td>
<td>Identifies the Key Required.</td>
</tr>
<tr>
<td>KEYPOLICY</td>
<td>2</td>
<td>2</td>
<td>Identifies which inputs are required to be used in the key derivation.</td>
</tr>
<tr>
<td>ISVSVPN</td>
<td>4</td>
<td>2</td>
<td>The ISV security version number that will be used in the key derivation.</td>
</tr>
<tr>
<td>CET_ATTRIBUTES_MASK</td>
<td>6</td>
<td>1</td>
<td>When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides a mask that defines which CET_ATTRIBUTES bits will be included in key derivation. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, then this field is reserved and must be 0.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>7</td>
<td>1</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>CPUVSVPN</td>
<td>8</td>
<td>16</td>
<td>The security version number of the processor used in the key derivation.</td>
</tr>
<tr>
<td>ATTRIBUTEMASK</td>
<td>24</td>
<td>16</td>
<td>A mask defining which ATTRIBUTES bits will be included in key derivation.</td>
</tr>
<tr>
<td>KEYID</td>
<td>40</td>
<td>32</td>
<td>Value for key wear-out protection.</td>
</tr>
<tr>
<td>MISCMASK</td>
<td>72</td>
<td>4</td>
<td>A mask defining which MISSELECT bits will be included in key derivation.</td>
</tr>
<tr>
<td>CONFIGSVN</td>
<td>76</td>
<td>2</td>
<td>Identifies which enclave Configuration's Security Version should be used in key derivation.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>78</td>
<td>434</td>
<td></td>
</tr>
</tbody>
</table>

### 35.18.1 KEY REQUEST KeyNames

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINITTOKEN_KEY</td>
<td>0</td>
<td>EINIT_TOKEN key</td>
</tr>
<tr>
<td>PROVISION_KEY</td>
<td>1</td>
<td>Provisioning Key</td>
</tr>
<tr>
<td>PROVISION_SEAL_KEY</td>
<td>2</td>
<td>Provisioning Seal Key</td>
</tr>
<tr>
<td>REPORT_KEY</td>
<td>3</td>
<td>Report Key</td>
</tr>
<tr>
<td>SEAL_KEY</td>
<td>4</td>
<td>Seal Key</td>
</tr>
<tr>
<td>All others</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### Table 35-26. Supported KEYName Values

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRENCLAVE</td>
<td>0</td>
<td>If 1, derive key using the enclave's MRENCLAVE measurement register.</td>
</tr>
<tr>
<td>MRSIGNER</td>
<td>1</td>
<td>If 1, derive key using the enclave's MRSIGNER measurement register.</td>
</tr>
<tr>
<td>NOISVPROMID</td>
<td>2</td>
<td>If 1, derive key WITHOUT using the enclave' ISVPROMID value.</td>
</tr>
<tr>
<td>CONFIGID</td>
<td>3</td>
<td>If 1, derive key using the enclave's CONFIGID value.</td>
</tr>
<tr>
<td>ISVFAMILYID</td>
<td>4</td>
<td>If 1, derive key using the enclave ISVFAMILYID value.</td>
</tr>
<tr>
<td>ISVEXTPRODID</td>
<td>5</td>
<td>If 1, derive key using enclave's ISVEXTPRODID value.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>15:6</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>

### 35.18.2 Key Request Policy Structure
35.19 VERSION ARRAY (VA)

In order to securely store the versions of evicted EPC pages, Intel SGX defines a special EPC page type called a Version Array (VA). Each VA page contains 512 slots, each of which can contain an 8-byte version number for a page evicted from the EPC. When an EPC page is evicted, software chooses an empty slot in a VA page; this slot receives the unique version number of the page being evicted. When the EPC page is reloaded, there must be a VA slot that must hold the version of the page. If the page is successfully reloaded, the version in the VA slot is cleared.

VA pages can be evicted, just like any other EPC page. When evicting a VA page, a version slot in some other VA page must be used to hold the version for the VA being evicted. A Version Array Page must be 4K-Bytes aligned.

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot 0</td>
<td>0</td>
<td>8</td>
<td>Version Slot 0</td>
</tr>
<tr>
<td>Slot 1</td>
<td>8</td>
<td>8</td>
<td>Version Slot 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Slot 511</td>
<td>4088</td>
<td>8</td>
<td>Version Slot 511</td>
</tr>
</tbody>
</table>

35.20 ENCLAVE PAGE CACHE MAP (EPCM)

EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds exactly one entry for each page that is currently loaded into the EPC. EPCM is not accessible by software, and the layout of EPCM fields is implementation specific.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALID</td>
<td>Indicates whether the EPCM entry is valid.</td>
</tr>
<tr>
<td>R</td>
<td>Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.</td>
</tr>
<tr>
<td>W</td>
<td>Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.</td>
</tr>
<tr>
<td>X</td>
<td>Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.</td>
</tr>
<tr>
<td>PT</td>
<td>EPCM page type (PT_SECS, PT_TCS, PT_REG, PT_VA, PT_TRIM, PT_SS_FIRST, PT_SS_REST).</td>
</tr>
<tr>
<td>ENCLAVESECS</td>
<td>SECS identifier of the enclave to which the EPC page belongs.</td>
</tr>
<tr>
<td>ENCLAVEADDRESS</td>
<td>Linear enclave address of the EPC page.</td>
</tr>
<tr>
<td>BLOCKED</td>
<td>Indicates whether the EPC page is in the blocked state.</td>
</tr>
<tr>
<td>PENDING</td>
<td>Indicates whether the EPC page is in the pending state.</td>
</tr>
<tr>
<td>MODIFIED</td>
<td>Indicates whether the EPC page is in the modified state.</td>
</tr>
<tr>
<td>PR</td>
<td>Indicates whether the EPC page is in a permission restriction state.</td>
</tr>
</tbody>
</table>

35.21 READ INFO (RDINFO)

The RDINFO structure contains status information about an EPC page. It must be aligned to 32-Bytes.
Table 35-30. Layout of RDINFO Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>OFFSET (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>0</td>
<td>8</td>
<td>Page status information.</td>
</tr>
<tr>
<td>FLAGS</td>
<td>8</td>
<td>8</td>
<td>EPCM state of the page.</td>
</tr>
<tr>
<td>ENCLAVECONTEXT</td>
<td>16</td>
<td>8</td>
<td>Context pointer describing the page’s parent location.</td>
</tr>
</tbody>
</table>

35.21.1 RDINFO Status Structure

Table 35-31. Layout of RDINFO STATUS Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHILDPRESENT</td>
<td>0</td>
<td>Indicates that the page has one or more child pages present (always zero for non-SECS pages). In VMX non-root operation includes the presence of virtual children.</td>
</tr>
<tr>
<td>VIRTCILDPRESENT</td>
<td>1</td>
<td>Indicates that the page has one or more virtual child pages present (always zero for non-SECS pages). In VMX non-root operation this value is always zero.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>63:2</td>
<td></td>
</tr>
</tbody>
</table>

35.21.2 RDINFO Flags Structure

Table 35-32. Layout of RDINFO FLAGS Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0</td>
<td>Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.</td>
</tr>
<tr>
<td>W</td>
<td>1</td>
<td>Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.</td>
</tr>
<tr>
<td>PENDING</td>
<td>3</td>
<td>Indicates whether the EPC page is in the pending state.</td>
</tr>
<tr>
<td>MODIFIED</td>
<td>4</td>
<td>Indicates whether the EPC page is in the modified state.</td>
</tr>
<tr>
<td>PR</td>
<td>5</td>
<td>Indicates whether the EPC page is in a permission restriction state.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>7:6</td>
<td></td>
</tr>
<tr>
<td>PAGE_TYPE</td>
<td>15:8</td>
<td>Indicates the page type of the EPC page.</td>
</tr>
<tr>
<td>RESERVED</td>
<td>62:16</td>
<td></td>
</tr>
<tr>
<td>BLOCKED</td>
<td>63</td>
<td>Indicates whether the EPC page is in the blocked state.</td>
</tr>
</tbody>
</table>
The following aspects of enclave operation are described in this chapter:
- Enclave creation: Includes loading code and data from outside of enclave into the EPC and establishing the enclave entity.
- Adding pages and measuring the enclave.
- Initialization of an enclave: Finalizes the cryptographic log and establishes the enclave identity and sealing identity.
- Enclave entry and exiting including:
  - Controlled entry and exit.
  - Asynchronous Enclave Exit (AEX) and resuming execution after an AEX.

36.1 CONSTRUCTING AN ENCLAVE

Figure 36-1 illustrates a typical Enclave memory layout.

The enclave creation, commitment of memory resources, and finalizing the enclave’s identity with measurement comprises multiple phases. This process can be illustrated by the following exemplary steps:

1. The application hands over the enclave content along with additional information required by the enclave creation API to the enclave creation service running at privilege level 0.

2. The enclave creation service running at privilege level 0 uses the ECREATE leaf function to set up the initial environment, specifying base address and size of the enclave. This address range, the ELRANGE, is part of the application’s address space. This reserves the memory range. The enclave will now reside in this address space.
ENCLAVE OPERATION

region. ECREATE also allocates an Enclave Page Cache (EPC) page for the SGX Enclave Control Structure (SECS). Note that this page is not required to be a part of the enclave linear address space and is not required to be mapped into the process.

3. The enclave creation service uses the EADD leaf function to commit EPC pages to the enclave, and use EEXTEND to measure the committed memory content of the enclave. For each page to be added to the enclave:
   — Use EADD to add the new page to the enclave.
   — If the enclave developer requires measurement of the page as a proof for the content, use EEXTEND to add a measurement for 256 bytes of the page. Repeat this operation until the entire page is measured.

4. The enclave creation service uses the EINIT leaf function to complete the enclave creation process and finalize the enclave measurement to establish the enclave identity. Until an EINIT is executed, the enclave is not permitted to execute any enclave code (i.e., entering the enclave by executing EENTER would result in a fault).

36.1.1 ECREATE

The ECREATE leaf function sets up the initial environment for the enclave by reading an SGX Enclave Control Structure (SECS) that contains the enclave's address range (ELRANGE) as defined by BASEADDR and SIZE, the ATTRIBUTES and MISCELL bitmaps, and the SSAFRAMESIZE. It then securely stores this information in an Enclave Page Cache (EPC) page. ELRANGE is part of the application's address space. ECREATE also initializes a cryptographic log of the enclave's build process.

36.1.2 EADD and EEXTEND Interaction

Once the SECS has been created, enclave pages can be added to the enclave via EADD. This involves converting a free EPC page into either a PT_REG or a PT_TCS page.

When EADD is invoked, the processor will update the EPCM entry with the type of page (PT_REG or PT_TCS), the linear address used by the enclave to access the page, and the enclave access permissions for the page. It associates the page to the SECS provided as input. The EPCM entry information is used by hardware to manage access control to the page. EADD records EPCM information in the cryptographic log stored in the SECS and copies 4 KBytes of data from unprotected memory outside the EPC to the allocated EPC page.

System software is responsible for selecting a free EPC page. System software is also responsible for providing the type of page to be added, the attributes of the page, the contents of the page, and the SECS (enclave) to which the page is to be added as requested by the application. Incorrect data would lead to a failure of EADD or to an incorrect cryptographic log and a failure at EINIT time.

After a page has been added to an enclave, software can measure a 256 byte region as determined by the developer by invoking EEXTEND. Thus to measure an entire 4KB page, system software must execute EEXTEND 16 times. Each invocation of EEXTEND adds to the cryptographic log information about which region is being measured and the measurement of the section.

Entries in the cryptographic log define the measurement of the enclave and are critical in gaining assurance that the enclave was correctly constructed by the untrusted system software.

36.1.3 EINIT Interaction

Once system software has completed the process of adding and measuring pages, the enclave needs to be initialized by the EINIT leaf function. After an enclave is initialized, EADD and EEXTEND are disabled for that enclave (An attempt to execute EADD/EEXTEND to enclave after enclave initialization will result in a fault). The initialization process finalizes the cryptographic log and establishes the enclave identity and sealing identity used by EGETKEY and EREPORT.

A cryptographic hash of the log is stored as the enclave identity. Correct construction of the enclave results in the cryptographic hash matching the one built by the enclave owner and included as the ENCLAVEHASH field of SIGSTRUCT. The enclave identity provided by the EREPORT leaf function can be verified by a remote party.
The EINIT leaf function checks the EINIT token to validate that the enclave has been enabled on this platform. If the enclave is not correctly constructed, or the EINIT token is not valid for the platform, or SIGSTRUCT isn't properly signed, then EINIT will fail. See the EINIT leaf function for details on the error reporting.

The **enclave identity** is a cryptographic hash that reflects the enclave attributes and MISCESELECT value, content of the enclave, the order in which it was built, the addresses it occupies in memory, the security attributes, and access right permissions of each page. The **enclave identity** is established by the EINIT leaf function.

The **sealing identity** is managed by a sealing authority represented by the hash of the public key used to sign the SIGSTRUCT structure processed by EINIT. The sealing authority assigns a product ID (ISVPRODID) and security version number (ISVSVN) to a particular enclave identity.

EINIT establishes the sealing identity using the following steps:

1. Verifies that SIGSTRUCT is properly signed using the public key enclosed in the SIGSTRUCT.
2. Checks that the measurement of the enclave matches the measurement of the enclave specified in SIGSTRUCT.
3. Checks that the enclave’s attributes and MISCESELECT values are compatible with those specified in SIGSTRUCT.
4. Finalizes the measurement of the enclave and records the **sealing identity** (the sealing authority, product id and security version number) and **enclave identity** in the SECS.
5. Sets the ATTRIBUTES.INIT bit for the enclave.

### 36.1.4 Intel® SGX Launch Control Configuration

Intel® SGX Launch Control is a set of controls that govern the creation of enclaves. Before the EINIT leaf function will successfully initialize an enclave, a designated Launch Enclave must create an EINITTOKEN for that enclave. Launch Enclaves have SECS.ATTRIBUTES.EINITTOKEN_KEY = 1, granting them access to the EINITTOKEN_KEY from the EGETKEY leaf function. EINITTOKEN_KEY must be used by the Launch Enclave when computing EINITTOKEN.MAC, the Message Authentication Code of the EINITTOKEN.

The hash of the public key used to sign the SIGSTRUCT of the Launch Enclave must equal the value in the IA32_SGXLEPUBKEYHASH MSRs. Only Launch Enclaves are allowed to launch without a valid token.

The IA32_SGXLEPUBKEYHASH MSRs are provided to designate the platform’s Launch Enclave. IA32_SGXLEPUBKEYHASH defaults to digest of Intel’s launch enclave signing key after reset.

IA32FEATURE_CONTROL bit 17 controls the permissions on the IA32_SGXLEPUBKEYHASH MSRs when CPUID.(EAX=12H, ECX=00H):EAX[0] = 1. If IA32_FEATURE_CONTROL is locked with bit 17 set, IA32_SGXLEPUBKEYHASH MSRs are reconfigurable (writeable). If either IA32_FEATURE_CONTROL is not locked or bit 17 is clear, the MSRs are read only. By leaving these MSRs writable, system SW or a VMM can support a plurality of Launch Enclaves for hosting multiple execution environments. See Table 40.2.2 for more details.

### 36.2 ENCLAVE ENTRY AND EXITING

#### 36.2.1 Controlled Entry and Exit

The EENTER leaf function is the method to enter the enclave under program control. To execute EENTER, software must supply an address of a TCS that is part of the enclave to be entered. The TCS holds the location inside the enclave to transfer control to and a pointer to the SSA frame inside the enclave that an AEX should store the register state to.

When a logical processor enters an enclave, the TCS is considered busy until the logical processors exits the enclave. An attempt to enter an enclave through a busy TCS results in a fault. Intel® SGX allows an enclave builder to define multiple TCSs, thereby providing support for multithreaded enclaves.

Software must also supply to EENTER the Asynchronous Exit Pointer (AEP) parameter. AEP is an address external to the enclave which an exception handler will return to using IRET. Typically the location would contain the ERESUME instruction. ERESUME transfers control back to the enclave, to the address retrieved from the enclave thread’s saved state.

EENTER performs the following operations:
ENCLAVE OPERATION

1. Check that TCS is not busy and flush all cached linear-to-physical mappings.
2. Change the mode of operation to be in enclave mode.
3. Save the old RSP, RBP for later restore on AEX (Software is responsible for setting up the new RSP, RBP to be used inside enclave).
4. Save XCR0 and replace it with the XFRM value for the enclave.
5. Check if software wishes to debug (applicable to a debuggable enclave):
   — If not debugging, then configure hardware so the enclave appears as a single instruction.
   — If debugging, then configure hardware to allow traps, breakpoints, and single steps inside the enclave.
6. Set the TCS as busy.
7. Transfer control from outside enclave to predetermined location inside the enclave specified by the TCS.

The EEXIT leaf function is the method of leaving the enclave under program control. EEXIT receives the target address outside of the enclave that the enclave wishes to transfer control to. It is the responsibility of enclave software to erase any secret from the registers prior to invoking EEXIT. To allow enclave software to easily perform an external function call and re-enter the enclave (using EEXIT and EENTER leaf functions), EEXIT returns the value of the AEP that was used when the enclave was entered.

EEXIT performs the following operations:
1. Clear enclave mode and flush all cached linear-to-physical mappings.
2. Mark TCS as not busy.
3. Transfer control from inside the enclave to a location on the outside specified as parameter to the EEXIT leaf function.

36.2.2 Asynchronous Enclave Exit (AEX)

Asynchronous and synchronous events, such as exceptions, interrupts, traps, SMIs, and VM exits may occur while executing inside an enclave. These events are referred to as Enclave Exiting Events (EEE). Upon an EEE, the processor state is securely saved inside the enclave (in the thread’s current SSA frame) and then replaced by a synthetic state to prevent leakage of secrets. The process of securely saving state and establishing the synthetic state is called an Asynchronous Enclave Exit (AEX). Details of AEX is described in Chapter 37, “Enclave Exiting Events.”

As part of most EEEs, the AEP is pushed onto the stack as the location of the eventing address. This is the location where control will return to after executing the IRET. The ERESUME leaf function can be executed from that point to reenter the enclave and resume execution from the interrupted point.

After AEX has completed, the logical processor is no longer in enclave mode and the exiting event is processed normally. Any new events that occur after the AEX has completed are treated as having occurred outside the enclave (e.g., a #PF in dispatching to an interrupt handler).

36.2.3 Resuming Execution After AEX

After system software has serviced the event that caused the logical processor to exit an enclave, the logical processor can continue enclave execution using ERESUME. ERESUME restores processor state and returns control to where execution was interrupted.

If the cause of the exit was an exception or a fault and was not resolved, the event will be triggered again if the enclave is re-entered using ERESUME. For example, if an enclave performs a divide by 0 operation, executing ERESUME will cause the enclave to attempt to re-execute the faulting instruction and result in another divide by 0 exception. Intel® SGX provides the means for an enclave developer to handle enclave exceptions from within the enclave. Software can enter the enclave at a different location and invoke the exception handler within the enclave by executing the EENTER leaf function. The exception handler within the enclave can read the fault information from the SSA frame and attempt to resolve the faulting condition or simply return and indicate to software that the enclave should be terminated (e.g., using EEXIT).
36.2.3.1 ERESUME Interaction

ERESUME restores registers depending on the mode of the enclave (32 or 64 bit).

- In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32-bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP, and EFLAGS) are restored from the thread’s GPR area of the current SSA frame. Neither the upper 32 bits of the legacy registers nor the 64-bit registers (R8 ... R15) are loaded.
- In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 ... R15, RIP, and RFLAGS) are loaded.

Extended features specified by SECS.ATTRIBUTES.XFRM are restored from the XSAVE area of the current SSA frame. The layout of the x87 area depends on the current values of IA32_EFER.LMA and CS.L:

- IA32_EFER.LMA = 0 || CS.L = 0
  - 32-bit load in the same format that XSAVE/FXSAVE uses with these values.
- IA32_EFER.LMA = 1 && CS.L = 1
  - 64-bit load in the same format that XSAVE/FXSAVE uses with these values as if REX.W = 1.

36.2.3.2 Asynchronous Enclave Exit Notify and EDECCSSA

Asynchronous Enclave Exit Notify (AEX-Notify) is an extension to Intel SGX that allows Intel SGX enclaves to be notified after an asynchronous enclave exit (AEX) has occurred. EDECCSSA is a new Intel SGX user leaf function (ENCLU[EDECCSSA]) that can facilitate AEX notification handling, as well as software exception handling. This section provides information about changes to the Intel SGX architecture that support AEX-Notify and ENCLU[EDECCSSA].

NOTE

On some platforms, AEX-Notify and the EDECCSSA user leaf function may be enumerated by CPUID following a microcode update.

The following list summarizes the additions to existing Intel SGX data structures to support AEX-Notify:

- SECS.ATTRIBUTES.AEXNOTIFY: This enclave supports AEX-Notify.
- TCS.FLAGS.AEXNOTIFY: This enclave thread may receive AEX notifications.
- SSA.GPRSGX.AEXNOTIFY: Enclave-writable byte that allows enclave software to dynamically enable/disable AEX notifications.

An AEX notification is delivered by ENCLU[ERESUME] when the following conditions are met:

1. TCS.FLAGS.AEXNOTIFY is set.
2. TCS.CSSA (the current slot index of an SSA frame) is greater than zero.
3. TCS.SSA[TCS.CSSA-1].GPRSGX.AEXNOTIFY[0] is set.

Note that AEX increments TCS.CSSA, and ENCLU[ERESUME] decrements TCS.CSSA, except when an AEX notification is delivered. Instead of decrementing TCS.CSSA and restoring state from the SSA, ENCLU[ERESUME] delivers an AEX notification by behaving as ENCLU[EENTER]. Implications of this behavior include:

- The enclave thread is resumed at EnclaveBase + TCS.OENTRY.
- EAX contains the (non-decremented) value of TCS.CSSA.
- RCX contains the address of the IP following ENCLU[ERESUME].
- The architectural state saved by the most recent AEX is preserved in TCS.SSA[TCS.CSSA-1].

The enclave thread can return to the previous SSA context by invoking ENCLU[EDECCSSA], which decrements TCS.CSSA.
36.3 CALLING ENCLAVE PROCEDURES

36.3.1 Calling Convention
In standard call conventions subroutine parameters are generally pushed onto the stack. The called routine, being aware of its own stack layout, knows how to find parameters based on compile-time-computable offsets from the SP or BP register (depending on runtime conventions used by the compiler).

Because of the stack switch when calling an enclave, stack-located parameters cannot be found in this manner. Entering the enclave requires a modified parameter passing convention.

For example, the caller might push parameters onto the untrusted stack and then pass a pointer to those parameters in RAX to the enclave software. The exact choice of calling conventions is up to the writer of the edge routines; be those routines hand-coded or compiler generated.

36.3.2 Register Preservation
As with most systems, it is the responsibility of the callee to preserve all registers except that used for returning a value. This is consistent with conventional usage and tends to optimize the number of register save/restore operations that need be performed. It has the additional security result that it ensures that data is scrubbed from any registers that were used by enclave to temporarily contain secrets.

36.3.3 Returning to Caller
No registers are modified during EEXIT. It is the responsibility of software to remove secrets in registers before executing EEXIT.

36.4 INTEL® SGX KEY AND ATTESTATION

36.4.1 Enclave Measurement and Identification
During the enclave build process, two “measurements” are taken of each enclave and are stored in two 256-bit Measurement Registers (MR): MRENCLAVE and MRSIGNER. MRENCLAVE represents the enclave’s contents and build process. MRSIGNER represents the entity that signed the enclave’s SIGSTRUCT.

The values of the Measurement Registers are included in attestations to identify the enclave to remote parties. The MRs are also included in most keys, binding keys to enclaves with specific MRs.

36.4.1.1 MRENCLAVE
MRENCLAVE is a unique 256 bit value that identifies the code and data that was loaded into the enclave during the initial launch. It is computed as a SHA256 hash that is initialized by the ECREATE leaf function. EADD and EEXTEND leaf functions record information about each page and the content of those pages. The EINIT leaf function finalizes the hash, which is stored in SECS.MRENCLAVE. Any tampering with the build process, contents of a page, page permissions, etc will result in a different MRENCLAVE value.

Figure 36-2 illustrates a simplified flow of changes to the MRENCLAVE register when building an enclave:
- Enclave creation with ECREATE.
- Copying a non-enclave source page into the EPC of an un-initialized enclave with EADD.
- Updating twice of the MRENCLAVE after modifying the enclave’s page content, i.e., EEXTEND twice.
- Finalizing the enclave build with EINIT.

Details on specific values inserted in the hash are available in the individual instruction definitions.
36.4.1.2 MRSIGNER

Each enclave is signed using a 3072 bit RSA key. The signature is stored in the SIGSTRUCT. In the SIGSTRUCT, the enclave's signer also assigns a product ID (ISVPRODID) and a security version (ISVSVN) to the enclave.

MRSIGNER is the SHA-256 hash of the signer's public key. For platforms that support Key Separation and Sharing (CPUID.(EAX=12H, ECX=1).EAX.KSS[7]) the SIGSTRUCT can additionally specify an 16 byte extended product ID (ISVEXTPRODID), and a 16 byte family ID (ISVFAMILYID).

In attestation, MRSIGNER can be used to allow software to approve of an enclave based on the author rather than maintaining a list of MRENCLAVES. It is used in key derivation to allow software to create a lineage of an application. By signing multiple enclaves with the same key, the enclaves will share the same keys and data. Combined with security version numbering, the author can release multiple versions of an application which can access keys for previous versions, but not future versions of that application.

36.4.1.3 CONFIGID

For platforms that support enhancements for key separation and sharing (CPUID.(EAX=12H, ECX=1).EAX.KSS[7]) when the enclave is created the platform can additionally provide 32-byte configuration identifier (CONFIGID). How this value is used is dependent on the enclave but it is intended to allow enclave creators to indicate what additional content may be accepted by the enclave post-initialization.

36.4.2 Security Version Numbers (SVN)

Intel® SGX supports a versioning system that allows the signer to identify different versions of the same software released by an author. The security version is independent of the functional version an author uses and is intended to specify security equivalence. Multiple releases with functional enhancements may all share the same SVN if they all have the same security properties or posture. Each enclave has an SVN and the underlying hardware has an SVN.

The SVNs are attested to in EREPORT and are included in the derivation of most keys, thus providing separation between data for older/newer versions.
36.4.2.1 Enclave Security Version

In the SIGSTRUCT, the MRSIGNER is associated with a 16-bit Product ID (ISVPRODID) and a 16 bit integer SVN (ISVSVN). Together they define a specific group of versions of a specific product. Most keys, including the Seal Key, can be bound to this pair.

To support upgrading from one release to another, EGETKEY will return keys corresponding to any value less than or equal to the software's ISVSVN.

36.4.2.2 Hardware Security Version

CPUSVN is a 128 bit value that reflects the microcode update version and authenticated code modules supported by the processor. Unlike ISVSVN, CPUSVN is not an integer and cannot be compared mathematically. Not all values are valid CPUSVNs.

Software must ensure that the CPUSVN provided to EGETKEY is valid. EREPORT will return the CPUSVN of the current environment. Software can execute EREPORT with TARGETINFO set to zeros to retrieve a CPUSVN from REPORTDATA. Software can access keys for a CPUSVN recorded previously, provided that each of the elements reflected in CPUSVN are the same or have been upgraded.

36.4.2.3 CONFIGID Security Version

The CONFIGID field can be used to contain the hash of a signing key for verifying the additional content. In this case, similar to the relationship between MRSIGNER and ISVSVN, CONFIGID needs a CONFIGID Security Version Number. CONFIGIDSVN can be specified at the same time as CONFIGID.

36.4.3 Keys

Intel® SGX provides software with access to keys unique to each processor and rooted in HW keys inserted into the processor during manufacturing.

Each enclave requests keys using the EGETKEY leaf function. The key is based on enclave parameters such as measurement, the enclave signing key, security attributes of the enclave, and the Hardware Security version of the processor itself. A full list of parameter options is specified in the KEYREQUEST structure, see details in Section 35.18.

By deriving keys using enclave properties, SGX guarantees that if two enclaves call EGETKEY, they will receive a unique key only accessible by the respective enclave. It also guarantees that the enclave will receive the same key on every future execution of EGETKEY. Some parameters are optional or configurable by software. For example, a Seal key can be based on the signer of the enclave, resulting in a key available to multiple enclaves signed by the same party.

The EGETKEY leaf function provides several key types. Each key is specific to the processor, CPUSVN, and the enclave that executed EGETKEY. The EGETKEY instruction definition details how each of these keys is derived, see Table 38-66. Additionally,

- **SEAL Key**: The Seal key is a general purpose key for the enclave to use to protect secrets. Typical uses of the Seal key are encrypting and calculating MAC of secrets on disk. There are 2 types of Seal Key described in Section 36.4.3.1.

- **REPORT Key**: This key is used to compute the MAC on the REPORT structure. The EREPORT leaf function is used to compute this MAC, and destination enclave uses the Report key to verify the MAC. The software usage flow is detailed in Section 36.4.3.2.

- **EINITTOKEN_KEY**: This key is used by Launch Enclaves to compute the MAC on EINITTOKENs. These tokens are then verified in the EINIT leaf function. The key is only available to enclaves with ATTRIBUTE.EINITTOKEN_KEY set to 1.

- **PROVISIONING Key and PROVISIONING SEAL Key**: These keys are used by attestation key provisioning software to prove to remote parties that the processor is genuine and identify the currently executing TCB. These keys are only available to enclaves with ATTRIBUTE.PROVISIONKEY set to 1.
36.4.3.1 Sealing Enclave Data

Enclaves can protect persistent data using Seal keys to provide encryption and/or integrity protection. EGETKEY provides two types of Seal keys specified in KEYREQUEST.KEYPOLICY field: MRENCLAVE-based key and MRSIGNER-based key.

The MRENCLAVE-based keys are available only to enclave instances sharing the same MRENCLAVE. If a new version of the enclave is released, the Seal keys will be different. Retrieving previous data requires additional software support.

The MRSIGNER-based keys are bound to the 3 tuple (MRSIGNER, ISVPROMID, ISVSVN). These keys are available to any enclave with the same MRSIGNER and ISVPROMID and an ISVSVN equal to or greater than the key in question. This is valuable for allowing new versions of the same software to retrieve keys created before an upgrade.

For platforms that support enhancements for key separation and sharing (CPUID.(EAX=12H, ECX=1).EAX.KSS[7]) four additional key policies for seal key derivation are provided. These add the ISVEXTPRODID, ISVFAMILYID, and CONFIGID/CONFIGSVN to the key derivation. Additionally, there is a policy to remove ISVPROMID from a key derivation to create a shared between different products that share the same MRSIGNER.

36.4.3.2 Using REPORTs for Local Attestation

Intel SGX provides a means for enclaves to securely identify one another, this is referred to as “Local Attestation”. SGX provides a hardware assertion, REPORT that contains calling enclaves Attributes, Measurements and User supplied data (described in detail in Section 35.16). Figure 36-3 shows the basic flow of information.

1. The source enclave determines the identity of the target enclave to populate TARGETINFO.
2. The source enclave calls EREPORT instruction to generate a REPORT structure. The EREPORT instruction conducts the following:
   - Populates the REPORT with identify information about the calling enclave.
   - Derives the Report Key that is returned when the target enclave executes the EGETKEY. TARGETINFO provides information about the target.
   - Computes a MAC over the REPORT using derived target enclave Report Key.
3. Non-enclave software copies the REPORT from source to destination.
4. The target enclave executes the EGETKEY instruction to request its REPORT key, which is the same key used by EREPORT at the source.
5. The target enclave verifies the MAC and can then inspect the REPORT to identify the source.

![Figure 36-3. SGX Local Attestation](image-url)
36.5 EPC AND MANAGEMENT OF EPC PAGES

EPC layout is implementation specific, and is enumerated through CPUID (see Table 34-7 for EPC layout). EPC is typically configured by BIOS at system boot time.

36.5.1 EPC Implementation

EPC must be properly protected against attacks. One example of EPC implementation could use a Memory Encryption Engine (MEE). An MEE provides a cost-effective mechanism of creating cryptographically protected volatile storage using platform DRAM. These units provide integrity, replay, and confidentiality protection. Details are implementation specific.

36.5.2 OS Management of EPC Pages

The EPC is a finite resource. SGX1 (i.e., CPUID.(EAX=12H, ECX=0):EAX.SGX1 = 1 but CPUID.(EAX=12H, ECX=0):EAX.SGX2 = 0) provides the EPC manager with leaf functions to manage this resource and properly swap pages out of and into the EPC. For that, the EPC manager would need to keep track of all EPC entries, type and state, context affiliation, and SECS affiliation.

Enclave pages that are candidates for eviction should be moved to BLOCKED state using EBLOCK instruction that ensures no new cached virtual to physical address mappings can be created by attempts to reference a BLOCKED page.

Before evicting blocked pages, EPC manager should execute ETRACK leaf function on that enclave and ensure that there are no stale cached virtual to physical address mappings for the blocked pages remain on any thread on the platform.

After removing all stale translations from blocked pages, system software should use the EWB leaf function for securely evicting pages out of the EPC. EWB encrypts a page in the EPC, writes it to unprotected memory, and invalidates the copy in EPC. In addition, EWB also creates a cryptographic MAC (PCMD.MAC) of the page and stores it in unprotected memory. A page can be reloaded back to the processor only if the data and MAC match. To ensure that only the latest version of the evicted page can be loaded back, the version of the evicted page is stored securely in a Version Array (VA) in EPC.

SGX1 includes two instructions for reloading pages that have been evicted by system software: ELDU and ELDB. The difference between the two instructions is the value of the paging state at the end of the instruction. ELDU results in a page being reloaded and set to an UNBLOCKED state, while ELDB results in a page loaded to a BLOCKED state.

ELDB is intended for use by a Virtual Machine Monitor (VMM). When a VMM reloads an evicted page, it needs to restore it to the correct state of the page (BLOCKED vs. UNBLOCKED) as it existed at the time the page was evicted. Based on the state of the page at eviction, the VMM chooses either ELDB or ELDU.

36.5.2.1 Enhancement to Managing EPC Pages

On processors supporting SGX2 (i.e., CPUID.(EAX=12H, ECX=0):EAX.SGX2 = 1), the EPC manager can manage EPC resources (while enclave is running) with more flexibility provided by the SGX2 leaf functions. The additional flexibility is described in Section 36.5.7 through Section 36.5.11.

36.5.3 Eviction of Enclave Pages

Intel SGX paging is optimized to allow the Operating System (OS) to evict multiple pages out of the EPC under a single synchronization.

The suggested flow for evicting a list of pages from the EPC is:

1. For each page to be evicted from the EPC:
   a. Select an empty slot in a Version Array (VA) page.
      • If no empty VA page slots exist, create a new VA page using the EPA leaf function.
b. Remove linear-address to physical-address mapping from the enclave context’s mapping tables (page table and EPT tables).

c. Execute the EBLOCK leaf function for the target page. This sets the target page state to BLOCKED. At this point no new mappings of the page will be created. So any access which does not have the mapping cached in the TLB will generate a #PF.

2. For each enclave containing pages selected in step 1:
   — Execute an ETRACK leaf function pointing to that enclave’s SECS. This initiates the tracking process that ensures that all caching of linear-address to physical-address translations for the blocked pages is cleared.

3. For all logical processors executing in processes (OS) or guests (VMM) that contain the enclaves selected in step 1:
   — Issue an IPI (inter-processor interrupt) to those threads. This causes those logical processors to asynchronously exit any enclaves they might be in, and as a result flush cached linear-address to physical-address translations that might hold stale translations to blocked pages. There is no need for additional measures such as performing a “TLB shootdown”.

4. After enclaves exit, allow logical processors to resume normal operation, including enclave re-entry as the tracking logic keeps track of the activity.

5. For each page to be evicted:
   — Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents, and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

At this point, system software has the only copy of each page data encrypted with its page metadata in main memory.

36.5.4 Loading an Enclave Page

To reload a previously evicted page, system software needs four elements: the VA slot used when the page was evicted, a buffer containing the encrypted page contents, a buffer containing the page metadata, and the parent SECS to associate this page with. If the VA page or the parent SECS are not already in the EPC, they must be reloaded first.

1. Execute ELDB/ELDU (depending on the desired BLOCKED state for the page), passing as parameters: the EPC page linear address, the VA slot, the encrypted page, and the page metadata.

2. Create a mapping in the enclave context’s mapping tables (page tables and EPT tables) to allow the application to access that page (OS: system page table; VMM: EPT).

The ELDB/ELDU instruction marks the VA slot empty so that the page cannot be replayed at a later date.

36.5.5 Eviction of an SECS Page

The eviction of an SECS page is similar to the eviction of an enclave page. The only difference is that an SECS page cannot be evicted until all other pages belonging to the enclave have been evicted. Since all other pages have been evicted, there will be no threads executing inside the enclave and tracking with ETRACK isn’t necessary. When reloading an enclave, the SECS page must be reloaded before all other constituent pages.

1. Ensure all pages are evicted from enclave.

2. Select an empty slot in a Version Array page.
   — If no VA page exists with an empty slot, create a new one using the EPA function leaf.

3. Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.
36.5.6 Eviction of a Version Array Page

VA pages do not belong to any enclave and tracking with ETRACK isn’t necessary. When evicting the VA page, a slot in a different VA page must be specified in order to provide versioning of the evicted VA page.

1. Select a slot in a Version Array page other than the page being evicted.
   - If no VA page exists with an empty slot, create a new one using the EPA leaf function.
2. Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents, and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

36.5.7 Allocating a Regular Page

On processors that support SGX2, allocating a new page to an already initialized enclave is accomplished by invoking the EAUG leaf function. Typically, the enclave requests that the OS allocates a new page at a particular location within the enclave’s address space. Once allocated, the page remains in a pending state until the enclave executes the corresponding EACCEPT leaf function to accept the new page into the enclave. Page allocation operations may be batched to improve efficiency.

The typical process for allocating a regular page is as follows:

1. Enclave requests additional memory from OS when the current allocation becomes insufficient.
2. The OS invokes the EAUG leaf function to add a new memory page to the enclave.
   a. EAUG may only be called on a free EPC page.
   b. Successful completion of the EAUG instruction places the target page in the VALID and PENDING state.
   c. All dynamically created pages have the type PT_REG and content of all zeros.
3. The OS maps the page in the enclave context’s mapping tables.
4. The enclave issues an EACCEPT instruction, which verifies the page’s attributes and clears the PENDING state. At that point the page becomes accessible for normal enclave use.

36.5.8 Allocating a TCS Page

On processors that support SGX2, allocating a new TCS page to an already initialized enclave is a two-step process. First the OS allocates a regular page with a call to EAUG. This page must then be accepted and initialized by the enclave to which it belongs. Once the page has been initialized with appropriate values for a TCS page, the enclave requests the OS to change the page’s type to PT_TCS. This change must also be accepted. As with allocating a regular page, TCS allocation operations may be batched.

A typical process for allocating a TCS page is as follows:

1. Enclave requests an additional page from the OS.
2. The OS invokes EAUG to add a new regular memory page to the enclave.
   a. EAUG may only be called on a free EPC page.
   b. Successful completion of the EAUG instruction places the target page in the VALID and PENDING state.
3. The OS maps the page in the enclave context’s mapping tables.
4. The enclave issues an EACCEPT instruction, at which point the page becomes accessible for normal enclave use.
5. The enclave initializes the contents of the new page.
6. The enclave requests that the OS convert the page from type PT_REG to PT_TCS.
7. OS issues an EMODT instruction on the page.
   a. The parameters to EMODT indicate that the regular page should be converted into a TCS.
36.5.9 Trimming a Page

On processors that support SGX2, Intel SGX supports the trimming of an enclave page as a special case of EMODT. Trimming allows an enclave to actively participate in the process of removing a page from the enclave (deallocation) by splitting the process into first removing it from the enclave's access and then removing it from the EPC using the EREMOVE leaf function. The page type PT_TRIM indicates that a page has been trimmed from the enclave's address space and that the page is no longer accessible to enclave software. Modifications to a page in the PT_TRIM state are not permitted; the page must be removed and then reallocated by the OS before the enclave may use the page again. Page deallocation operations may be batched to improve efficiency.

The typical process for trimming a page from an enclave is as follows:

1. Enclave signals OS that a particular page is no longer in use.
2. OS invokes the EMODT leaf function on the page, requesting that the page’s type be changed to PT_TRIM.
   a. SECS and VA pages cannot be trimmed in this way, so the initial type of the page must be PT_REG or PT_TCS.
   b. EMODT may only be called on valid enclave pages.
3. OS invokes the ETRACK leaf function on the enclave containing the page to track removal the TLB addresses from all the processors.
4. Issue an IPI (inter-processor interrupt) to flush the stale linear-address to physical-address translations for all logical processors executing in processes that contain the enclave.
5. Enclave issues an EACCEPT leaf function.
6. The OS may now permanently remove the page from the EPC (by issuing EREMOVE).

36.5.10 Restricting the EPCM Permissions of a Page

On processors that support SGX2, restricting the EPCM permissions associated with an enclave page is accomplished using the EMODPR leaf function. This operation requires the cooperation of the OS to flush stale entries to the page and to update the page-table permissions of the page to match. Permissions restriction operations may be batched.

The typical process for restricting the permissions of an enclave page is as follows:

1. Enclave requests that the OS to restrict the permissions of an EPC page.
2. OS performs permission restriction, flushing cached linear-address to physical-address translations, and page-table modifications.
   a. Invokes the EMODPR leaf function to restrict permissions (EMODPR may only be called on VALID pages).
   b. Invokes the ETRACK leaf function on the enclave containing the page to track removal of the TLB addresses from all the processor.
   c. Issue an IPI (inter-processor interrupt) to flush the stale linear-address to physical-address translations for all logical processors executing in processes that contain the enclave.
   d. Sends IPIs to trigger enclave thread exit and TLB shootdown.
   e. OS informs the Enclave that all logical processors should now see the new restricted permissions.
3. Enclave invokes the EACCEPT leaf function.
   a. Enclave may access the page throughout the entire process.
   b. Successful call to EACCEPT guarantees that no stale cached linear-address to physical-address translations are present.
36.5.11 Extending the EPCM Permissions of a Page

On processors that support SGX2, extending the EPCM permissions associated with an enclave page is accomplished directly by the enclave using the EMODPE leaf function. After performing the EPCM permission extension, the enclave requests the OS to update the page table permissions to match the extended permission. Security wise, permission extension does not require enclave threads to leave the enclave as TLBs with stale references to the more restrictive permissions will be flushed on demand, but to allow forward progress, an OS needs to be aware that an application might signal a page fault.

The typical process for extending the permissions of an enclave page is as follows:
1. Enclave invokes EMODPE to extend the EPCM permissions associated with an EPC page (EMODPE may only be called on VALID pages).
2. Enclave requests that OS update the page tables to match the new EPCM permissions.
3. Enclave code resumes.
   a. If cached linear-address to physical-address translations are present to the more restrictive permissions, the enclave thread will page fault. The SGX2-aware OS will see that the page tables permit the access and resume the thread, which can now successfully access the page because exiting cleared the TLB.
   b. If cached linear-address to physical-address translations are not present, access to the page with the new permissions will succeed without an enclave exit.

36.5.12 VMM Oversubscription of EPC

On processors supporting oversubscription enhancements (i.e., CPUID.(EAX=12H, ECX=0):EAX[5]=1 & EAX[6] = 1) a Virtual Machine Monitor or other executive can more efficiently manage the EPC space available on the platform between virtualized entities. A typical process for using these instructions to support oversubscribing the physical EPC space on the platform is as follows:
1. VMM creates data structures for SECS tracking including a count of child pages.
2. VMM selects possible EPC victim pages.
3. VMM ages the victim pages. Some of the selected pages will be accessed by the guest. In this case the VMM will remove these pages from the victim pool and return them to the guest.
4. VMM makes remaining pages not present in EPT. It then issues IPI on each page to remove TLB mappings.
5. For every EPC victim page the VMM obtains the victim’s SECS page info using ERDINFO.
   a. ENCLAVECONTEXT field in RDINFO structure will indicate the location of SECS, and the PAGE_TYPE field will indicate the page type.
   b. Child pages of SECS can be evicted.
   c. SECS pages may be evicted if the child count is zero.
   d. Some pages may be returned to active state depending on such things as page type or child count.
6. VMM increments its evicted page count for the SECS of each page (stored in the data structure created in 1).
7. If this is the first evicted page of that SECS, set Marker on SECS of the victim page (EINCVIRTCHILD). This locks the SECS in the guest. The guest cannot page out the SECS.
8. EBLOCK, ETRACK, EWB eviction sequence is executed for page.
9. After loading an SECS page back in, the VMM will set the correct ENCLAVECONTEXT for the guest using ESETCONTEXT instruction.

36.6 CHANGES TO INSTRUCTION BEHAVIOR INSIDE AN ENCLAVE

This section covers instructions whose behavior changes when executed in enclave mode.
### 36.6.1 Illegal Instructions

The instructions listed in Table 36-1 are ring 3 instructions which become illegal when executed inside an enclave. Executing these instructions inside an enclave will generate an exception.

The first row of Table 36-1 enumerates instructions that may cause a VM exit for VMM emulation. Since a VMM cannot emulate enclave execution, execution of any of these instructions inside an enclave results in an invalid-opcode exception (#UD) and no VM exit.

The second row of Table 36-1 enumerates I/O instructions that may cause a fault or a VM exit for emulation. Again, enclave execution cannot be emulated, so execution of any of these instructions inside an enclave results in #UD.

The third row of Table 36-1 enumerates instructions that load descriptors from the GDT or the LDT or that change privilege level. The former class is disallowed because enclave software should not depend on the contents of the descriptor tables and the latter because enclave execution must be entirely with CPL = 3. Again, execution of any of these instructions inside an enclave results in #UD.

The fourth row of Table 36-1 enumerates instructions that provide access to kernel information from user mode and can be used to aid kernel exploits from within enclave. Execution of any of these instructions inside an enclave results in #UD.

RDTSC and RDTSCP are legal inside an enclave for processors that support SGX2 (subject to the value of CR4.TSD). For processors which support SGX1 but not SGX2, RDTSC and RDTSCP will cause #UD.

### Table 36-1. Illegal Instructions Inside an Enclave

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUID, GETSEC, RDPMC, SGDT, SIDT, SLDT, STR, VMCALL, VMFUNC</td>
<td>#UD</td>
<td>Might cause VM exit.</td>
</tr>
<tr>
<td>IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD</td>
<td>#UD</td>
<td>I/O fault may not safely recover. May require emulation.</td>
</tr>
<tr>
<td>SMSW</td>
<td>#UD</td>
<td>Might provide access to kernel information.</td>
</tr>
<tr>
<td>ENCLU[ENTER], ENCLU[ERESUME]</td>
<td>#GP</td>
<td>Cannot enter an enclave from within an enclave.</td>
</tr>
</tbody>
</table>

RDTSC and RDTSCP are legal inside an enclave for processors that support SGX2 (subject to the value of CR4.TSD). For processors which support SGX1 but not SGX2, RDTSC and RDTSCP will cause #UD.

### 36.6.2 RDRAND and RDSEED Instructions

These instructions may cause a VM exit if the “RDRAND exiting” VM-execution control is 1. Unlike other instructions that can cause VM exits, these instructions are legal inside an enclave. As noted in Section 28.1 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, any VM exit originating on an instruction boundary inside an enclave sets bit 27 of the exit-reason field of the VMCS. If a VMM receives a VM exit due to an attempt to execute either of these instructions determines (by that bit) that the execution was inside an enclave, it can do either of two things. It can clear the “RDRAND exiting” VM-execution control and execute VMRESUME; this will result in the enclave executing RDRAND or RDSEED again, and this time a VM exit will not occur. Alternatively, the VMM might choose to discontinue execution of this virtual machine.

**NOTE**

It is expected that VMMs that virtualize Intel SGX will not set “RDRAND exiting” to 1.

### 36.6.3 PAUSE Instruction

The PAUSE instruction may cause a VM exit from an enclave if the “PAUSE exiting” VM-execution control is 1. Unlike other instructions that can cause VM exits, the PAUSE instruction is legal inside an enclave. If a VMM receives a VM exit due to the 1-setting of “PAUSE exiting”, it can do either of two things. It can clear the “PAUSE exiting” VM-
execution control and execute VMRESUME; this will result in the enclave executing PAUSE again, but this time a VM exit will not occur. Alternatively, the VMM might choose to discontinue execution of this virtual machine.

The PAUSE instruction may also cause a VM exit outside of an enclave if the "PAUSE-loop exiting" VM-execution control is 1, but as the "PAUSE-loop exiting" control is ignored at CPL > 0 (see Section 26.1.3), VM exit from an enclave due to the 1-setting of "PAUSE-LOOP exiting" will never occur.

**NOTE**

It is expected that VMMs that virtualize Intel SGX will not set "PAUSE exiting" to 1.

### 36.6.4 Executions of INT1 and INT3 Inside an Enclave

The INT1 and INT3 instructions are legal inside an enclave, however, their behavior inside an enclave differs from that outside an enclave. See Section 40.4.1 for details.

### 36.6.5 INVD Handling when Enclaves Are Enabled

Once processor reserved memory protections are activated (see Section 36.5), any execution of INVD will result in a #GP(0).
Certain events, such as exceptions and interrupts, incident to (but asynchronous with) enclave execution may cause control to transition outside of enclave mode. (Most of these also cause a change of privilege level.) To protect the integrity and security of the enclave, the processor will exit the enclave (and enclave mode) before invoking the handler for such an event. For that reason, such events are called **enclave-exiting events** (EEE); EEEs include external interrupts, non-maskable interrupts, system-management interrupts, exceptions, and VM exits.

The process of leaving an enclave in response to an EEE is called an **asynchronous enclave exit** (AEX). To protect the secrecy of the enclave, an AEX saves the state of certain registers within enclave memory and then loads those registers with fixed values called **synthetic state**.

### 37.1 COMPATIBLE SWITCH TO THE EXITING STACK OF AEX

AEXs load registers with a pre-determined synthetic state. These registers may be later pushed onto the appropriate stack in a form as defined by the enclave-exiting event. To allow enclave execution to resume after the invoking handler has processed the enclave exiting event, the asynchronous enclave exit loads the address of trampoline code outside of the enclave into RIP. This trampoline code eventually returns to the enclave by means of an ENCLU(ERESUME) leaf function. Prior to exiting the enclave the RSP and RBP registers are restored to their values prior to enclave entry.

The stack to be used is chosen using the same rules as for non-SGX mode:

- If there is a privilege level change, the stack will be the one associated with the new ring.
- If there is no privilege level change, the current application stack is used.
- If the IA-32e IST mechanism is used, the exit stack is chosen using that method.

![Figure 37-1. Exit Stack Just After Interrupt with Stack Switch](image-url)
ENCLAVE EXITING EVENTS

In all cases, the choice of exit stack and the information pushed onto it is consistent with non-SGX operation. Figure 37-1 shows the Application and Exiting Stacks after an exit with a stack switch. An exit without a stack switch uses the Application Stack. The ERESUME leaf index value is placed into RAX, the TCS pointer is placed in RBX and the AEP (see below) is placed into RCX to facilitate resuming the enclave after the exit.

Upon an AEX, the AEP (Asynchronous Exit Pointer) is loaded into the RIP. The AEP points to a trampoline code sequence which includes the ERESUME instruction that is later used to reenter the enclave.

The following bits of RFLAGS are cleared before RFLAGS is pushed onto the exit stack: CF, PF, AF, ZF, SF, OF, RF. The remaining bits are left unchanged.

37.2 STATE SAVING BY AEX

The State Save Area holds the processor state at the time of an AEX. To allow handling events within the enclave and re-entering it after an AEX, the SSA can be a stack of multiple SSA frames as illustrated in Figure 37-2.

The location of the SSA frames to be used is controlled by the following variables in the TCS and the SECS:

- Size of a frame in the State Save Area (SECS.SSAFRAMESIZE): This defines the number of 4-KByte pages in a single frame in the State Save Area. The SSA frame size must be large enough to hold the GPR state, the XSAVE state, and the MISC state.
- Base address of the enclave (SECS.BASEADDR): This defines the enclave's base linear address from which the offset to the base of the SSA stack is calculated.
- Number of State Save Area Slots (TCS.NSSA): This defines the total number of slots (frames) in the State Save Area stack.
- Current State Save Area Slot (TCS.CSSA): This defines the slot to use on the next exit.
- State Save Area Offset (TCS.OSSA): This defines the offset of the base address of a set of State Save Area slots from the enclave’s base address.

When an AEX occurs, hardware selects the SSA frame to use by examining TCS.CSSA. Processor state is saved into the SSA frame (see Section 37.4) and loaded with a synthetic state (as described in Section 37.3.1) to avoid leaking secrets, RSP and RBP are restored to their values prior to enclave entry, and TCS.CSSA is incremented. As will be described later, if an exception takes the last slot, it will not be possible to reenter the enclave to handle the
exception from within the enclave. A subsequent ERESUME restores the processor state from the current SSA frame and frees the SSA frame.

The format of the XSAVE section of SSA is identical to the format used by the XSAVE/XRSTOR instructions. On EENTER, CSSA must be less than NSSA, ensuring that there is at least one State Save Area slot available for exits. If there is no free SSA frame when executing EENTER, the entry will fail.

## 37.3 SYNTHETIC STATE ON ASYNCHRONOUS ENCLAVE EXIT

### 37.3.1 Processor Synthetic State on Asynchronous Enclave Exit

Table 37-1 shows the synthetic state loaded on AEX. The values shown are the lower 32 bits when the processor is in 32 bit mode and 64 bits when the processor is in 64 bit mode.

### Table 37-1. GPR, x87 Synthetic States on Asynchronous Enclave Exit

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>3 (ENCLU[3] is ERESUME).</td>
</tr>
<tr>
<td>RBX</td>
<td>Pointer to TCS of interrupted enclave thread.</td>
</tr>
<tr>
<td>RCX</td>
<td>AEP of interrupted enclave thread.</td>
</tr>
<tr>
<td>RDX, RSI, RDI</td>
<td>0.</td>
</tr>
<tr>
<td>RSP</td>
<td>Restored from SSA.uRSP.</td>
</tr>
<tr>
<td>RBP</td>
<td>Restored from SSA.uRBP.</td>
</tr>
<tr>
<td>RB-R15</td>
<td>0 in 64-bit mode; unchanged in 32-bit mode.</td>
</tr>
<tr>
<td>RIP</td>
<td>AEP of interrupted enclave thread.</td>
</tr>
<tr>
<td>RFLAGS</td>
<td>CF, PF, AF, ZF, SF, OF, RF bits are cleared. All other bits are left unchanged.</td>
</tr>
<tr>
<td>x87/SSE State</td>
<td>Unless otherwise listed here, all x87 and SSE state are set to the INIT state. The INIT state is the state that would be loaded by the XRSTOR instruction with bits 1:0 both set in the requested feature bitmask (RFBM), and both clear in XSTATE_BV the XSAVE header.</td>
</tr>
<tr>
<td>FCW</td>
<td>On #MF exception: set to 037EH. On all other exits: set to 037FH.</td>
</tr>
<tr>
<td>FSW</td>
<td>On #MF exception: set to 8081H. On all other exits: set to 0H.</td>
</tr>
<tr>
<td>MXCSR</td>
<td>On #XM exception: set to 1F01H. On all other exits: set to 1FB0H.</td>
</tr>
</tbody>
</table>
| CR2      | If the event that caused the AEX is a #PF, and the #PF does not directly cause a VM exit, then the low 12 bits are cleared.  
  If the #PF leads directly to a VM exit, CR2 is not updated (usual IA behavior).  
  Note: The low 12 bits are not cleared if a #PF is encountered during the delivery of the EEE that caused the AEX. This is because the #PF was not the EEE. |
| FS, GS   | Restored to values as of most recent EENTER/ERESUME. |

### 37.3.2 Synthetic State for Extended Features

When CR4.OSXSAVE = 1, extended features (those controlled by XCR0[63:2]) are set to their respective INIT states when this corresponding bit of SECS.XFRM is set. The INIT state is the state that would be loaded by the XRSTOR instruction had the instruction mask and the XSTATE_BV field of the XSAVE header each contained the value XFRM. (When the AEX occurs in 32-bit mode, those features that do not exist in 32-bit mode are unchanged.)
ENCLAVE EXITING EVENTS

37.3.3 Synthetic State for MISC Features

State represented by SECS.MISCSELECT might also be overridden by synthetic state after it has been saved into the SSA. State represented by MISCSELECT[0] is not overridden but if the exiting event is a page fault then lower 12 bits of CR2 are cleared.

37.4 AEX FLOW

On Enclave Exiting Events (interrupts, exceptions, VM exits or SMIs), the processor state is securely saved inside the enclave, a synthetic state is loaded and the enclave is exited. The EEE then proceeds in the usual exit-defined fashion. The following sections describes the details of an AEX:

1. The exact processor state saved into the current SSA frame depends on whether the enclave is a 32-bit or a 64-bit enclave. In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32 bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP, and EFLAGS) are stored. The upper 32 bits of the legacy registers and the 64-bit registers (R8 ... R15) are not stored.

   In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 ... R15, RIP, and RFLAGS) are stored.

   The state of those extended features specified by SECS.ATTRIBUTES.XFRM are stored into the XSAVE area of the current SSA frame. The layout of the x87 and XMM portions (the 1st 512 bytes) depends on the current values of IA32_EFER.LMA and CS.L:

   If IA32_EFER.LMA = 0 || CS.L = 0, the same format (32-bit) that XSAVE/FXSAVE uses with these values.

   If IA32_EFER.LMA = 1 && CS.L = 1, the same format (64-bit) that XSAVE/FXSAVE uses with these values when REX.W = 1.

   The cause of the AEX is saved in the EXITINFO field. See Table 35-10 for details and values of the various fields.

   The state of those miscellaneous features (see Section 35.7.2) specified by SECS.MISCSELECT are stored into the MISC area of the current SSA frame.

   If CET was enabled in the enclave, then the CET state of the enclave is saved in the CET state save area. If shadow stacks were enabled in the enclave, then the SSP is also saved into the TCS.PREVSSP field.

2. Synthetic state is created for a number of processor registers to present an opaque view of the enclave state.

   Table 37-1 shows the values for GPRs, x87, SSE, FS, GS, Debug, and performance monitoring on AEX. The synthetic state for other extended features (those controlled by XCR0[62:2]) is set to their respective INIT states when their corresponding bit of SECS.ATTRIBUTES.XFRM is set. The INIT state is that state as defined by the behavior of the XRSTOR instruction when HEADER.XSTATE_BV[n] is 0. Synthetic state of those miscellaneous features specified by SECS.MISCSELECT depends on the miscellaneous feature. There is no synthetic state required for the miscellaneous state controlled by SECS.MISCSELECT[0].

3. Any code and data breakpoints that were suppressed at the time of enclave entry are unsuppressed when exiting the enclave.

4. RFLAGS.TF is set to the value that it had at the time of the most recent enclave entry (except for the situation that the entry was opt-in for debug; see Section 40.2). In the SSA, RFLAGS.TF is set to 0.

5. RFLAGS.RF is set to 0 in the synthetic state. In the SSA, the value saved is the same as what would have been saved on stack in the non-SGX case (architectural value of RF). Thus, AEXs due to interrupts, traps, and code breakpoints save RF unmodified into SSA, while AEXs due to other faults save RF as 1 in the SSA.

   If the event causing AEX happened on intermediate iteration of a REP-prefixed instruction, then RF=1 is saved on SSA, irrespective of its priority.

6. Any performance monitoring activity (including PEBS) or profiling activity (LBR, Tracing using Intel PT) on the exiting thread that was suppressed due to the enclave entry on that thread is unsuppressed. Any counting that had been demoted from AnyThread counting to MyThread counting (on one logical processor) is promoted back to AnyThread counting.

7. The CET state of the enclosing application is restored to the state at the time of the most recent enclave entry, and if CET indirect branch tracking was enabled then the indirect branch tracker is unsuppressed and moved to the WAIT_FOR_ENDBRANCH state.
37.4.1 AEX Operational Detail

The pseudo code in this section describes the internal operations that are executed when an AEX occurs in enclave mode. These operations occur just before the normal interrupt or exception processing occurs.

(* Save RIP for later use *)
TMP_RIP = Linear Address of Resume RIP
(* Is the processor in 64-bit mode? *)
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Save all registers, When saving EFLAGS, the TF bit is set to 0 and
the RF bit is set to what would have been saved on stack in the non-SGX case *)

IF (TMP_MODE64 = 0)
    THEN
        Save EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EFLAGS, EIP into the current SSA frame using
        CR_GPR_PA; (* see Table 38-5 for list of CREGs used to describe internal operation within Intel SGX *)
        SSA.RFLAGS.TF := 0;
    ELSE    (* TMP_MODE64 = 1 *)
        Save RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8-R15, RFLAGS, RIP into the current SSA frame using
        CR_GPR_PA;
        SSA.RFLAGS.TF := 0;
    FI;
Save FS and GS BASE into SSA using CR_GPR_PA;

(* store XSAVE state into the current SSA frame's XSAVE area using the physical addresses
that were determined and cached at enclave entry time with CR_XSAVE_PAGE_i. *)
For each XSAVE state i defined by (SECS.ATTRIBUTES.XFRM[i] = 1, destination address cached in
CR_XSAVE_PAGE_i)
    SSA.XSAVE.i := XSAVE_STATE_i;

(* Clear bytes 8 to 23 of XSAVE_HEADER, i.e., the next 16 bytes after XHEADER_BV *)
CR_XSAVE_PAGE_0.XHEADER_BV[191:64] := 0;

(* Clear bits in XHEADER_BV[63:0] that are not enabled in ATTRIBUTES.XFRM *)
CR_XSAVE_PAGE_0.XHEADER_BV[63:0] :=
    CR_XSAVE_PAGE_0.XHEADER_BV[63:0] & SECS(CR_ACTIVE_SECS).ATTRIBUTES.XFRM;
Apply synthetic state to GPRs, RFLAGS, extended features, etc.

(* Restore the RSP and RBP from the current SSA frame's GPR area using the physical address
that was determined and cached at enclave entry time with CR_GPR_PA. *)
RSP := CR_GPR_PA.URSP;
RBP := CR_GPR_PA.URBP;

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_RIP</td>
<td>Effective Address 32/64</td>
<td></td>
<td>Address of instruction at which to resume execution on ERESUME.</td>
</tr>
<tr>
<td>TMP_MODE64</td>
<td>binary</td>
<td>1</td>
<td>((IA32_EFER.LMA = 1) &amp;&amp; (CS.L = 1)).</td>
</tr>
<tr>
<td>TMP_BRANCH_RECORD</td>
<td>LBR Record</td>
<td>2x64</td>
<td>From/To address to be pushed onto LBR stack.</td>
</tr>
</tbody>
</table>
ENCLAVE EXITING EVENTS

(* Restore the FS and GS *)
FS.selector := CR_SAVE_FS.selector;
FS.base := CR_SAVE_FS.base;
FS.limit := CR_SAVE_FS.limit;
FS.access_rights := CR_SAVE_FS.access_rights;
GS.selector := CR_SAVE_GS.selector;
GS.base := CR_SAVE_GS.base;
GS.limit := CR_SAVE_GS.limit;
GS.access_rights := CR_SAVE_GS.access_rights;

(* Examine exception code and update enclave internal states*)
exception_code := Exception or interrupt vector;

(* Indicate the exit reason in SSA *)
IF (exception_code = (#DE OR #DB OR #BP OR #BR OR #UD OR #MF OR #AC OR #XM ))
THEN
    CR_GPR_PA.EXITINFO.VECTOR := exception_code;
    IF (exception code = #BP)
        THEN CR_GPR_PA.EXITINFO.EXIT_TYPE := 6;
        ELSE CR_GPR_PA.EXITINFO.EXIT_TYPE := 3;
    FI;
    CR_GPR_PA.EXITINFO.VALID := 1;
ELSE IF (exception code is #PF or #GP )
THEN
    (* Check SECS.MISCSELECT using CR_ACTIVE_SECS *)
    IF (SECS.MISCSELECT[0] is set)
        THEN
            CR_GPR_PA.EXITINFO.VECTOR := exception_code;
            CR_GPR_PA.EXITINFO.EXIT_TYPE := 3;
            IF (exception_code is #PF)
                THEN
                    SSA.MISC.EXINFO. MADDR := CR2;
                    SSA.MISC.EXINFO.ERRCD := PFEC;
                    SSA.MISC.EXINFO.RESERVED := 0;
                ELSE
                    SSA.MISC.EXINFO. MADDR := 0;
                    SSA.MISC.EXINFO.ERRCD := GPEC;
                    SSA.MISC.EXINFO.RESERVED := 0;
                FI;
            CR_GPR_PA.EXITINFO.VALID := 1;
        ELSE IF (exception code is #CP)
            THEN
                IF (SECS.MISCSELECT[1] is set)
                    THEN
                        CR_GPR_PA.EXITINFO.VECTOR := exception_code;
                        CR_GPR_PA.EXITINFO.EXIT_TYPE := 3;
                        CR_GPR_PA.EXITINFO.VALID := 1;
                        SSA.MISC.EXINFO. MADDR := 0;
                        SSA.MISC.EXINFO.ERRCD := CPEC;
                        SSA.MISC.EXINFO.RESERVED := 0;
                    FI;
                FI;
            ELSE
                FI;
            ELSE
                FI;
        ELSE
            FI;
    ELSE
        FI;
    FI;
ELSE
CR_GPR_PA.EXITINFO.VECTOR := 0;
CR_GPR_PA.EXITINFO.EXIT_TYPE := 0
CR_GPR_PA.REASON.VALID := 0;

FI;

(* Execution will resume at the AEP *)
RIP := CR_TCS_PA.AEP;

(* Set EAX to the ERESUME leaf index *)
EAX := 3;

(* Put the TCS LA into RBX for later use by ERESUME *)
RBX := CR_TCS_LA;

(* Put the AEP into RCX for later use by ERESUME *)
RCX := CR_TCS_PA.AEP;

(* Increment the SSA frame # *)
CR_TCS_PA.CSSA := CR_TCS_PA.CSSA + 1;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
   THEN XCR0 := CR_SAVE_XCR0; FI;

Un-suppress all code breakpoints that are outside ELRANGE

   THEN
      IF (CR4.CET == 1 AND IA32_U_CET.SH_STK_EN == 1)
         THEN
            CR_CET_SAVE_AREA_PA.SSP := SSP;
            CR_TCS_PA.PREVSSP := SSP;
         FI;
      IF (CR4.CET == 1 AND IA32_U_CET.ENDBR_EN == 1)
         THEN
            CR_CET_SAVE_AREA_PA.TRACKER := IA32_U_CET.TRACKER;
            CR_CET_SAVE_AREA_PA.SUPPRESS := IA32_U_CET.SUPPRESS;
         FI;
   FI;
IF (!CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (!CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1)
   THEN
      IF (CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1)
         THEN
            IA32_U_CET := CR_SAVE_IA32_U_CET;
            IF (CPUID.(EAX=7, ECX=0):ECX[CET_SS])
               THEN
                  SSP := CR_SAVE_SSP; FI;
         (* If indirect branch tracking enabled for enclosing application *)
         (* then move the tracker to wait_for_endbranch *)
         IF (CR4.CET == 1 AND IA32_U_CET.ENDBR_EN == 1)
            THEN
               IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
               IA32_U_CET.SUPPRESS := 0;
            FI;
ENCLAVE EXITING EVENTS

FI;

(* Update the thread context to show not in enclave mode *)
CR_ENCLAVE_MODE := 0;

(* Assure consistent translations. *)
Flush linear context including TLBs and paging-structure caches

IF (CR_DBGOPTIN = 0)
    THEN
        Un-suppress all breakpoints that overlap ELRANGE
        (* Clear suppressed breakpoint matches *)
        Restore suppressed breakpoint matches
        (* Restore TF *)
        RFLAGS.TF := CR_SAVE_TF;
        Un-suppress monitor trap flag;
        Un-suppress branch recording facilities;
        Un-suppress all suppressed performance monitoring activity;
        Promote any sibling-thread counters that were demoted from AnyThread to MyThread during enclave entry back to AnyThread;
    FI;

IF the “monitor trap flag” VM-execution control is 1
    THEN Pend MTF VM Exit at the end of exit; FI;

(* Clear low 12 bits of CR2 on #PF *)
IF (Exception code is #PF)
    THEN CR2 := CR2 & ~0xFFF; FI;

(* end_of_flow *)
(* Execution continues with normal event processing. *)
This chapter describes the supervisor and user level instructions provided by Intel® Software Guard Extensions (Intel® SGX). In general, various functionality is encoded as leaf functions within the ENCLS (supervisor), ENCLU (user), and the ENCLV (virtualization operation) instruction mnemonics. Different leaf functions are encoded by specifying an input value in the EAX register of the respective instruction mnemonic.

### 38.1 Intel® SGX Instruction Syntax and Operation

ENCLS, ENCLU, and ENCLV instruction mnemonics for all leaf functions are covered in this section.

For all instructions, the value of CS.D is ignored; addresses and operands are 64 bits in 64-bit mode and are otherwise 32 bits. Aside from EAX specifying the leaf number as input, each instruction leaf may require all or some subset of the RBX/RCX/RDX as input parameters. Some leaf functions may return data or status information in one or more of the general purpose registers.

#### 38.1.1 ENCLS Register Usage Summary

Table 38-1 summarizes the implicit register usage of supervisor mode enclave instructions.

<table>
<thead>
<tr>
<th>Instr. Leaf</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECREATE</td>
<td>00H (In)</td>
<td>PAGEINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EAADD</td>
<td>01H (In)</td>
<td>PAGEINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EINIT</td>
<td>02H (In)</td>
<td>SIGSTRUCT (In, EA)</td>
<td>SECS (In, EA)</td>
<td>EINITTOKEN (In, EA)</td>
</tr>
<tr>
<td>EREMOVE</td>
<td>03H (In)</td>
<td></td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EDBGRD</td>
<td>04H (In)</td>
<td>Result Data (Out)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EDBGWR</td>
<td>05H (In)</td>
<td>Source Data (In)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EEXTEND</td>
<td>06H (In)</td>
<td>SECS (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>ELDB</td>
<td>07H (In)</td>
<td>PAGEINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td>VERSION (In, EA)</td>
</tr>
<tr>
<td>ELDU</td>
<td>08H (In)</td>
<td>PAGEINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td>VERSION (In, EA)</td>
</tr>
<tr>
<td>EBLOCK</td>
<td>09H (In)</td>
<td></td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EPA</td>
<td>0AH (In)</td>
<td>PT_VA (In)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EwB</td>
<td>0BH (In)</td>
<td>PAGEINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td>VERSION (In, EA)</td>
</tr>
<tr>
<td>ETRACK</td>
<td>0CH (In)</td>
<td></td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EAUG</td>
<td>0DH (In)</td>
<td>PAGEINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EMODPR</td>
<td>0EH (In)</td>
<td>SECINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EMODT</td>
<td>0FH (In)</td>
<td>SECINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>ERDINFO</td>
<td>010H (In)</td>
<td>RDINFO (In, EA*)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>ETRACKC</td>
<td>011H (In)</td>
<td></td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>ELDBC</td>
<td>012H (In)</td>
<td>PAGEINFO (In, EA*)</td>
<td>EPCPAGE (In, EA)</td>
<td>VERSION (In, EA)</td>
</tr>
<tr>
<td>ELDUC</td>
<td>013H (In)</td>
<td>PAGEINFO (In, EA*)</td>
<td>EPCPAGE (In, EA)</td>
<td>VERSION (In, EA)</td>
</tr>
</tbody>
</table>

EA: Effective Address
### 38.1.2 ENCLU Register Usage Summary

Table 38-2 summarizes the implicit register usage of user mode enclave instructions.

**Table 38-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions**

<table>
<thead>
<tr>
<th>Instr. Leaf</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>EREPORT</td>
<td>00H (In)</td>
<td>TARGETINFO (In, EA)</td>
<td>REPORTDATA (In, EA)</td>
<td>OUTPUTDATA (In, EA)</td>
</tr>
<tr>
<td>EGETKEY</td>
<td>01H (In)</td>
<td>KEYREQUEST (In, EA)</td>
<td>KEY (In, EA)</td>
<td></td>
</tr>
<tr>
<td>ENTER</td>
<td>02H (In)</td>
<td>TCS (In, EA)</td>
<td>AEP (In, EA)</td>
<td>RBX.CSSA (Out)</td>
</tr>
<tr>
<td>ERESUME</td>
<td>03H (In)</td>
<td>TCS (In, EA)</td>
<td>AEP (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EEXIT</td>
<td>04H (In)</td>
<td>Target (In, EA)</td>
<td>Current AEP (Out)</td>
<td></td>
</tr>
<tr>
<td>EACCEPT</td>
<td>05H (In)</td>
<td>SECINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EMODPE</td>
<td>06H (In)</td>
<td>SECINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EACCEPTCOPY</td>
<td>07H (In)</td>
<td>SECINFO (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
<td>EPCPAGE (In, EA)</td>
</tr>
<tr>
<td>EDECCSSA</td>
<td>09H (In)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EA: Effective Address

### 38.1.3 ENCLV Register Usage Summary

Table 38-3 summarizes the implicit register usage of virtualization operation enclave instructions.

**Table 38-3. Register Usage of Virtualization Operation Enclave Instruction Leaf Functions**

<table>
<thead>
<tr>
<th>Instr. Leaf</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDECVIRTCHILD</td>
<td>00H (In)</td>
<td>EPCPAGE (In, EA)</td>
<td>SECS (In, EA)</td>
<td></td>
</tr>
<tr>
<td>EINCVIRTCHILD</td>
<td>01H (In)</td>
<td>EPCPAGE (In, EA)</td>
<td>SECS (In, EA)</td>
<td></td>
</tr>
<tr>
<td>ESETCONTEXT</td>
<td>02H (In)</td>
<td>EPCPAGE (In, EA)</td>
<td>Context Value (In, EA)</td>
<td></td>
</tr>
</tbody>
</table>

EA: Effective Address

### 38.1.4 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 38-4 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

**Table 38-4. Error or Information Codes for Intel® SGX Instructions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Returned By</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SGX_INVALID_SIG_STRUCT</td>
<td>1</td>
<td>EINIT</td>
</tr>
<tr>
<td>SGX_INVALID_ATTRIBUTE</td>
<td>2</td>
<td>EINIT, EGETKEY</td>
</tr>
<tr>
<td>SGX_BLKSTATE</td>
<td>3</td>
<td>EBLOCK</td>
</tr>
<tr>
<td>SGX_INVALID_MEASUREMENT</td>
<td>4</td>
<td>EINIT</td>
</tr>
<tr>
<td>SGX_NOTBLOCKABLE</td>
<td>5</td>
<td>EBLOCK</td>
</tr>
<tr>
<td>SGX_PG_INVLD</td>
<td>6</td>
<td>EBLOCK, ERDINFO, ETRACKC</td>
</tr>
</tbody>
</table>

38-2  Vol. 3D
### Table 38-4. Error or Information Codes for Intel® SGX Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Returned By</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>7</td>
<td>EBLOCK, EMODPR, EMODT, ERDINFO, EDECVIRTCHILD, EINCVIRTCHILD, ELDBC, ELDC, EACCEPT, EACCEPTCOPY, ETRACKC</td>
</tr>
<tr>
<td>SGX_INVALID_SIGNATURE</td>
<td>8</td>
<td>EINIT</td>
</tr>
<tr>
<td>SGX_MACCOMPARE_FAIL</td>
<td>9</td>
<td>ELDB, ELDU, ELDBC, ELDUC</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_BLOCKED</td>
<td>10</td>
<td>EwB</td>
</tr>
<tr>
<td>SGX_NOT_TRACKED</td>
<td>11</td>
<td>EwB, EACCEPT</td>
</tr>
<tr>
<td>SGX_VA_SLOT_OCCUPIED</td>
<td>12</td>
<td>EwB</td>
</tr>
<tr>
<td>SGX_CHILD_PRESENT</td>
<td>13</td>
<td>EwB, EREMOVE</td>
</tr>
<tr>
<td>SGX_ENCLAVE_ACT</td>
<td>14</td>
<td>EREMOVE</td>
</tr>
<tr>
<td>SGX_ENTRYEOPOCH_LOCKED</td>
<td>15</td>
<td>EBLOCK</td>
</tr>
<tr>
<td>SGX_INVALID_EINITTOKEN</td>
<td>16</td>
<td>EINIT</td>
</tr>
<tr>
<td>SGX_PREV_TRK_INCMPL</td>
<td>17</td>
<td>ETRACK, ETRACKC</td>
</tr>
<tr>
<td>SGX_PG_IS_SECS</td>
<td>18</td>
<td>EBLOCK</td>
</tr>
<tr>
<td>SGX_PAGE_ATTRIBUTES_MISMATCH</td>
<td>19</td>
<td>EACCEPT, EACCEPTCOPY</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_MODIFIABLE</td>
<td>20</td>
<td>EMODPR, EMODT</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_DEBUGGABLE</td>
<td>21</td>
<td>EDBGRD, EDBGWR</td>
</tr>
<tr>
<td>SGX_INVALID_COUNTER</td>
<td>25</td>
<td>EDECVIRTCHILD</td>
</tr>
<tr>
<td>SGX_PG_NONEPC</td>
<td>26</td>
<td>ERDINFO</td>
</tr>
<tr>
<td>SGX_TRACK_NOT_REQUIRED</td>
<td>27</td>
<td>ETRACKC</td>
</tr>
<tr>
<td>SGX_INVALID_CPUSVN</td>
<td>32</td>
<td>EINIT, EGETKEY</td>
</tr>
<tr>
<td>SGX_INVALID_ISVSVN</td>
<td>64</td>
<td>EGETKEY</td>
</tr>
<tr>
<td>SGX_UNMASKED_EVENT</td>
<td>128</td>
<td>EINIT</td>
</tr>
<tr>
<td>SGX_INVALID_KEYNAME</td>
<td>256</td>
<td>EGETKEY</td>
</tr>
</tbody>
</table>

### 38.1.5 Internal CREGs

The CREGs as shown in Table 5-4 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 38-5 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

### Table 38-5. List of Internal CREG

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (Bits)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR_ENCLAVE_MODE</td>
<td>1</td>
<td>LP</td>
</tr>
<tr>
<td>CR_DBGOPTIN</td>
<td>1</td>
<td>LP</td>
</tr>
<tr>
<td>CR_TCS_LA</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_TCS_PA</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_ACTIVE_SECS</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_ELRANGE</td>
<td>128</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_TF</td>
<td>1</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_FS</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_GPR_PA</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_XSAVE_PAGE_n</td>
<td>64</td>
<td>LP</td>
</tr>
</tbody>
</table>
38.1.6 Concurrent Operation Restrictions

Under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leafs to concurrently operate on the same EPC page.
  - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
  - EADD, EEXTEND, and EINIT leaves are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function may do one of the following:

- Return an SGX_EPC_PAGE_CONFLICT error code in RAX.
- Cause a #GP(0) exception.

To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same EPC page.

38.1.6.1 Concurrency Tables of Intel® SGX Instructions

The tables below detail the concurrent operation restrictions of all SGX leaf functions. For each leaf function, the table has a separate line for each of the EPC pages the leaf function accesses.

For each such EPC page, the base concurrency requirements are detailed as follows:

- **Exclusive Access** means that no other leaf function that requires either shared or exclusive access to the same EPC page may be executed concurrently. For example, EADD requires an exclusive access to the target page it accesses.

### Table 38-5. List of Internal CREG

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (Bits)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR_SAVE_DR7</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_PERF_GLOBAL_CTRL</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_DEBUGCTL</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_PEB_ENABLE</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_CPUSVN</td>
<td>128</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_SGXOWNEREPOCH</td>
<td>128</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_SAVE_XCR0</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SGX_ATTRIBUTES_MASK</td>
<td>128</td>
<td>LP</td>
</tr>
<tr>
<td>CR_PAGING_VERSION</td>
<td>64</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_VERSION_THRESHOLD</td>
<td>64</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_NEXT_EID</td>
<td>64</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_BASE_PK</td>
<td>128</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_SEAL_FUSES</td>
<td>128</td>
<td>PACKAGE</td>
</tr>
<tr>
<td>CR_CET_SAVE_AREA_PA</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_ENCLAVE_SS_TOKEN_PA</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_IA32_U_CET</td>
<td>64</td>
<td>LP</td>
</tr>
<tr>
<td>CR_SAVE_SSP</td>
<td>64</td>
<td>LP</td>
</tr>
</tbody>
</table>
• **Shared Access** means that no other leaf function that requires an exclusive access to the same EPC page may be executed concurrently. Other leaf functions that require shared access may run concurrently. For example, EADD requires a shared access to the SECS page it accesses.

• **Concurrent Access** means that any other leaf function that requires any access to the same EPC page may be executed concurrently. For example, EGETKEY has no concurrency requirements for the KEYREQUEST page.

In addition to the base concurrency requirements, additional concurrency requirements are listed, which apply only to specific sets of leaf functions. For example, there are additional requirements that apply for EADD, EXTEND, and EINIT. EADD and EEXTEND can’t execute concurrently on the same SECS page.

The tables also detail the leaf function’s behavior when a conflict happens, i.e., a concurrency requirement is not met. In this case, the leaf function may return an SGX_EPC_PAGE_CONFLICT error code in RAX, or it may cause an exception. In addition, the tables detail those conflicts where a VM Exit may be triggered, and list the Exit Qualification code that is provided in such cases.

### Table 38-6. Base Concurrency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCEPT</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SECINFO</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EACCEPTCOPY</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Source</td>
<td>[DS:RDX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SECINFO</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EADD</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
</tr>
<tr>
<td></td>
<td>SECS</td>
<td>[DS:RBX]PAGEINFO, SECS</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td>EAUG</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
</tr>
<tr>
<td></td>
<td>SECS</td>
<td>[DS:RBX]PAGEINFO, SECS</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td>EBLOCK</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
</tr>
<tr>
<td>ECREATE</td>
<td>SECS</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
</tr>
<tr>
<td>EDBGDRD</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td>EDBGWR</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td>EDENVIRCHTCHILD</td>
<td>Target</td>
<td>[DS:RBX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SECS</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EENTERTCS</td>
<td>SECS</td>
<td>[DS:RBX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td>EEEXIT</td>
<td></td>
<td></td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEXTEND</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SECS</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EGETKEY</td>
<td>KEYREQUEST</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUTDATA</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EINCVIRCHTCHILD</td>
<td>Target</td>
<td>[DS:RBX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SECS</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EINIT</td>
<td>SECS</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
</tbody>
</table>
### Table 38-6. Base Concurrency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELDB/ELDU</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>[DS:RDX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECS</td>
<td>[DS:RBX]PAGEINFO, SECS</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDLBC/ELDUC</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>EPC_PAGE_CONFLICT_ERROR</td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>[DS:RDX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECS</td>
<td>[DS:RBX]PAGEINFO, SECS</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMODPE</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECDINFO</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMODPR</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>EMODT</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>EPC_PAGE_CONFLICT_ERROR</td>
<td></td>
</tr>
<tr>
<td>EPA</td>
<td>[DS:RDX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>ERDINFO</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EREMOVE</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
</tr>
<tr>
<td>EREPORT</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPORTDATA</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUTDATA</td>
<td>[DS:RDX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERESUME</td>
<td>[DS:RBX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESETCONTEXT</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETRACK</td>
<td>SECS</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETRACKC</td>
<td>[DS:RCX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EwB</td>
<td>[DS:RDX]</td>
<td>Shared</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
</tr>
</tbody>
</table>

### Table 38-7. Additional Concurrency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
<td>On Conflict</td>
<td>Access</td>
</tr>
<tr>
<td>EACCEPT</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>Concurrent</td>
</tr>
<tr>
<td>SECINFO</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
### Table 38-7. Additional Concurrency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>vs. EACCEPTCOPY, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
<td>On Conflict</td>
<td>Access</td>
</tr>
<tr>
<td>EACCEPTCOPY</td>
<td>Source [DS:RDX]</td>
<td>Exclusive</td>
<td>#GP Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EACCEPTCOPY</td>
<td>EACCEPTCOPY [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EACCEPTCOPY</td>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EADD</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EADD</td>
<td>SEC [DS:RBX]</td>
<td>Concurrent</td>
<td>Exclusive #GP Concurrent</td>
<td></td>
</tr>
<tr>
<td>EAUG</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EAUG</td>
<td>SEC [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EBLOCK</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ECREATE</td>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EDBGWR</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EDENVIRCHILD</td>
<td>Target [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EDENVIRCHILD</td>
<td>SEC [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EEVENT</td>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EEVENT</td>
<td>SEC [DS:RBX]</td>
<td>Concurrent</td>
<td>Exclusive #GP Concurrent</td>
<td></td>
</tr>
<tr>
<td>EEVENT</td>
<td>EGETKEY KEYREQUEST [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EEVENT</td>
<td>OUTPUTDATA [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EINIT</td>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EINIT</td>
<td>SEC [DS:RBX]</td>
<td>Concurrent</td>
<td>Exclusive #GP Concurrent</td>
<td></td>
</tr>
<tr>
<td>ELDB/ELDUC</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ELDB/ELDUC</td>
<td>VA [DS:RDX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ELDB/ELDUC</td>
<td>SEC [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EDLBC/ELDUC</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EDLBC/ELDUC</td>
<td>VA [DS:RDX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EDLBC/ELDUC</td>
<td>SEC [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EMODPE</td>
<td>Target [DS:RCX]</td>
<td>Exclusive</td>
<td>#GP Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EMODPE</td>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EMODPR</td>
<td>Target [DS:RCX]</td>
<td>Exclusive</td>
<td>SGX_EPC_PAGE_CONFLICT Concurrent</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
### Table 38-7. Additional Concurrency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access On Conflict</td>
<td>Access On Conflict</td>
<td>Access On Conflict</td>
</tr>
<tr>
<td>EMODT</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Exclusive</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
</tr>
<tr>
<td>EPA</td>
<td>VA</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ERDINFO</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EREMOVE</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EREPORT</td>
<td>TARGETINFO</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>REPORTDATA</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>OUTPUTDATA</td>
<td>[DS:RDX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ERESUME</td>
<td>TCS</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ESETCONTEXT</td>
<td>SECS</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ETRACK</td>
<td>SECS</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>ETRACKC</td>
<td>Target</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>SECS</td>
<td>Implicit</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td>EwB</td>
<td>Source</td>
<td>[DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>VA</td>
<td>[DS:RDX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

**NOTES:**
1. SGX_CONFLICT VM Exit Qualification = TRACKINGRESOURCE_CONFLICT.

### 38.2 INTEL® SGX INSTRUCTION REFERENCE
### ENCLS—Execute an Enclave System Function of Specified Leaf Number

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 CF ENCLS</td>
<td>ZO</td>
<td>V/V</td>
<td>NA</td>
<td>This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Implicit Register Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>See Section 38.3</td>
</tr>
</tbody>
</table>

### Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the “enable ENCLS exiting” VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the “enable ENCLS exiting” VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

### Operation

**IF TSX_ACTIVE**

THEN GOTO TSX_ABORT_PROCESSING; FI;

**IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0**

THEN #UD; FI;

**IF (CPL > 0)**

THEN #UD; FI;

**IF in VMX non-root operation and the “enable ENCLS exiting” VM-execution control is 1**

THEN

**IF EAX < 63 and ENCLS_exiting_bitmap[EAX] = 1 or EAX > 62 and ENCLS_exiting_bitmap[63] = 1**

THEN VM exit;

FI;

FI;

**IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0**

THEN #GP(0); FI;

**IF (EAX is an invalid leaf number)**

THEN #GP(0); FI;
IF CR0.PG = 0
  THEN #GP(0); Fl;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
  THEN #GP(0); Fl;

Jump to leaf specific flow

**Flags Affected**
See individual leaf functions

**Protected Mode Exceptions**

* #UD If any of the LOCK/66H/REP/VEX prefixes are used.
* If current privilege level is not 0.
* If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
  * If logical processor is in SMM.
* #GP(0) If IA32_FEATURE_CONTROL.LOCK = 0.
  * If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
  * If input value in EAX encodes an unsupported leaf.
  * If data segment expand down.
  * If CR0.PG=0.

**Real-Address Mode Exceptions**

* #UD ENCLS is not recognized in real mode.

**Virtual-8086 Mode Exceptions**

* #UD ENCLS is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

* #UD If any of the LOCK/66H/REP/VEX prefixes are used.
  * If current privilege level is not 0.
  * If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
  * If logical processor is in SMM.
* #GP(0) If IA32_FEATURE_CONTROL.LOCK = 0.
  * If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
  * If input value in EAX encodes an unsupported leaf.
ENCLU—Execute an Enclave User Function of Specified Leaf Number

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 D7 ENCLU</td>
<td>ZO</td>
<td>V/V</td>
<td>NA</td>
<td>This instruction is used to execute non-privileged Intel SGX leaf functions.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Implicit Register Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>See Section 38.4</td>
</tr>
</tbody>
</table>

### Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

### Operation

```plaintext
IN_64BIT_MODE := 0;
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

(* If enclosing app has CET indirect branch tracking enabled then if it is not ERESUME leaf cause a #CP fault *)
(* If the ERESUME is not successful it will leave tracker in WAIT_FOR_ENDBRANCH *)
TRACKER = (CPL == 3) ? IA32_U_CET.TRACKER : IA32_S_CET.TRACKER
IF EndbranchEnabledAndNotSuppressed(CPL) and TRACKER = WAIT_FOR_ENDBRANCH and (EAX != ERESUME or CR0.TS or (in SMM) or (CPUID.SGX_LEAF.0:EAX.SE1 = 0) or (CPL < 3))
    THEN
        Handle CET State machine violation (* see Section 17.3.6, “Legacy Compatibility Treatment,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. *)
FI;

IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF CR0.TS = 1
    THEN #NM; FI;

IF CPL < 3
    THEN #UD; FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;
```

Vol. 3D 38-11
IF EAX is invalid leaf number
  THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0
  THEN #GP(0); FI;

IN_64BIT_MODE := IA32_EFER.LMA AND CS.L ? 1 : 0;
(* Check not in 16-bit mode and DS is not a 16-bit segment *)
IF not in 64-bit mode and CS.D = 0
  THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 1 and (EAX = 2 or EAX = 3) (* EENTER or ERESUME *)
  THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or EAX = 6 or EAX = 7 or EAX = 9)
(* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, EACCEPTCOPY, or EDECCSSA *)
  THEN #GP(0); FI;

Jump to leaf specific flow

Flags Affected
See individual leaf functions

Protected Mode Exceptions

#UD If any of the LOCK/66H/REP/VEX prefixes are used.
If current privilege level is not 3.
If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
If logical processor is in SMM.

#GP(0) If IA32_FEATURE_CONTROL.LOCK = 0.
If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
If input value in EAX encodes an unsupported leaf.
If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE
and ENCLAVE_MODE = 0.
If operating in 16-bit mode.
If data segment is in 16-bit mode.
If CR0.PG = 0 or CR0.NE= 0.

#NM If CR0.TS = 1.

Real-Address Mode Exceptions

#UD ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-Bit Mode Exceptions

#UD  If any of the LOCK/66H/REP/VEX prefixes are used.
     If current privilege level is not 3.
     If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
     If logical processor is in SMM.

#GP(0) If IA32_FEATURE_CONTROL.LOCK = 0.
       If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
       If input value in EAX encodes an unsupported leaf.
       If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
       If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE
          and ENCLAVE_MODE = 0.
          If CR0.NE= 0.

#NM If CR0.TS = 1.
**INTEL® SGX INSTRUCTION REFERENCES**

**ENCLV—Execute an Enclave VMM Function of Specified Leaf Number**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP 0F 01 C0 ENCLV</td>
<td>ZO</td>
<td>V/V</td>
<td>NA</td>
<td>This instruction is used to execute privileged SGX leaf functions that are reserved for VMM use. They are used for managing the enclaves.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Implicit Register Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>See Section 38.3</td>
</tr>
</tbody>
</table>

**Description**

The ENCLV instruction invokes the virtualization SGX leaf functions for managing enclaves in a virtualized environment. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In non-64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLV instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, if it is executed in system-management mode (SMM), or not in VMX operation. Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

Software in VMX root mode of operation can enable execution of the ENCLV instruction in VMX non-root mode by setting enable ENCLV execution control in the VMCS. If enable ENCLV execution control in the VMCS is clear, execution of the ENCLV instruction in VMX non-root mode results in #UD.

When execution of ENCLV instruction in VMX non-root mode is enabled, software in VMX root operation can intercept the invocation of various ENCLV leaf functions in VMX non-root operation by setting the corresponding bits in the ENCLV-exiting bitmap.

Addresses and operands are 32 bits in 32-bit mode (IA32_EFER.LMA == 0 || CS.L == 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA == 1 && CS.L == 1). CS.D value has no impact on address calculation.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

**Operation**

IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.OSS = 0
    THEN #UD; FI;

IF not in VMX Operation or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation
    IF "enable ENCLV exiting“ VM-execution control is 1
        THEN
            IF EAX < 63 and ENCLV_exiting_bitmap[EAX] = 1 or EAX > 62 and ENCLV_exiting_bitmap[63] = 1
                THEN VM exit;
            FI;
            ELSE
                #UD; FI;

38-14 Vol. 3D
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
THEN #GP(0); FI;

IF (EAX is an invalid leaf number)
THEN #GP(0); FI;

IF CR0.PG = 0
THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
THEN #GP(0); FI;

Jump to leaf specific flow

Flags Affected
See individual leaf functions.

Protected Mode Exceptions
#UD If any of the LOCK/66H/REP/VEX prefixes are used.
If current privilege level is not 0.
If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0.
If logical processor is in SMM.
#GP(0) If IA32_FEATURE_CONTROL.LOCK = 0.
If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
If input value in EAX encodes an unsupported leaf.
If data segment expand down.
If CR0.PG=0.

Real-Address Mode Exceptions
#UD ENCLV is not recognized in real mode.

Virtual-8086 Mode Exceptions
#UD ENCLV is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#UD If any of the LOCK/66H/REP/VEX prefixes are used.
If current privilege level is not 0.
If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0.
If logical processor is in SMM.
#GP(0) If IA32_FEATURE_CONTROL.LOCK = 0.
If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
If input value in EAX encodes an unsupported leaf.
38.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLS instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.
EADD—Add a Page to an Uninitialized Enclave

**Opcode/Instruction** | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EAX = 01H ENCLS[EADD] | IR | V/V | SGX1 | This leaf function adds a page to an uninitialized enclave.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EADD (In)</td>
<td>Address of a PAGEINFO (In)</td>
<td>Address of the destination EPC page (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function copies a source page from non-enclave memory into the EPC, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in EPCM. As part of the association, the enclave offset and the security attributes are measured and extended into the SECS.MRENCLAVE. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of EADD leaf function.

**EADD Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>PAGEINFO</th>
<th>PAGEINFO.SECS</th>
<th>PAGEINFO.SRCPAGE</th>
<th>PAGEINFO.SECINFO</th>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read/Write access permitted by Enclave</td>
<td>Read access permitted by Non Enclave</td>
<td>Read access permitted by Non Enclave</td>
<td>Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EADD Faulting Conditions**

- The operands are not properly aligned.
- Unsupported security attributes are set.
- Refers to an invalid SECS.
- Reference is made to an SECS that is locked by another thread.
- The EPC page is locked by another thread.
- RCX does not contain an effective address of an EPC page.
- The EPC page is already valid.
- If security attributes specifies a TCS and the source page specifies unsupported TCS values or fields.
- The SECS has been initialized.
- The specified enclave offset is outside of the enclave address space.

**Concurrency Restrictions**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EADD</td>
<td>Target [DS:RCX]</td>
<td>Access: Exclusive; On Conflict: #GP; SGX_CONFLICT VM Exit Qualification: EPC_PAGE_CONFLICT_EXCEPTION</td>
</tr>
<tr>
<td>SECS [DS:RBX] PAGEINFO.SECS</td>
<td>Shared: #GP</td>
<td></td>
</tr>
</tbody>
</table>

Vol. 3D 38-17
Operation

**Temp Variables in EADD Operational Flow**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SRCPGE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of the source page.</td>
</tr>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of the SECS destination page.</td>
</tr>
<tr>
<td>TMP_SECINFO</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of an SECINFO structure which contains security attributes of the page to be added.</td>
</tr>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:TMP_SECINFO.</td>
</tr>
<tr>
<td>TMP_LINADDR</td>
<td>Unsigned Integer</td>
<td>64</td>
<td>Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.</td>
</tr>
<tr>
<td>TMP_ENCLAVEOFFSET</td>
<td>Enclave Offset</td>
<td>64</td>
<td>The page displacement from the enclave base address.</td>
</tr>
<tr>
<td>TMPUPDATEFIELD</td>
<td>SHA256 Buffer</td>
<td>512</td>
<td>Buffer used to hold data being added to TMP_SECS.MRENCLAVE.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); Fl;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); Fl;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fl;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_SECINFO := DS:RBX.SECINFO;
TMP_LINADDR := DS:RBX.LINADDR;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECS is not 4KByte aligned or DS:TMP_SECINFO is not 64Byte aligned or TMP_LINADDR is not 4KByte aligned)
    THEN #GP(0); Fl;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); Fl;

SCRATCH_SECINFO := DS:TMP_SECINFO;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero or
IF (SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS or
(SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
(SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
THEN #GP(0); Fi;

(* If PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST OR
      SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST) AND CR4.CET == 0)
THEN #GP(0); Fi;

(* Check the EPC page for concurrency *)
IF (EPC page is not available for EADD)
THEN
  IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
    THEN
      VMCS.Exit_reason := SGX_CONFLICT;
      VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
      VMCS.Exit_qualification.error := 0;
      VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
      VMCS.Guest-linear_address := DS:RCX;
      Deliver VMEXIT;
    ELSE
      #GP(0);
    FI;
  ELSE
    #GP(0);
  FI;
ELSE
  #GP(0);
FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
THEN #PF(DS:RCX); Fi;

(* Check the SECS for concurrency *)
IF (SECS is not available for EADD)
THEN #GP(0); Fi;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
THEN #PF(DS:TMP_SECS); Fi;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPGE[32767:0];
CASE (SCRATCH_SECINFO.FLAGS.PT)

  PT_TCS:
    IF (DS:RCX.RESERVED ≠ 0) #GP(0); Fi;
    IF ((DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and
        ((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH))) #GP(0); Fi;
    (* Ensure TCS.PREVSSP is zero *)
    IF (CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1) and (DS:RCX.PREVSSP != 0) #GP(0); Fi;
    BREAK;
  PT_REG:
    IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #GP(0); Fi;
    BREAK;
  PT_SS_FIRST:
  PT_SS_REST:
  (* SS pages cannot be created on first or last page of ELRANGE *)
IF (TMP_LINADDR = DS:TMP_SECS.BASEADDR or
    TMP_LINADDR = (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000))
    THEN #GP(0); Fl;
IF (DS:RCX[4087:0] != 0) #GP(0); Fl;
IF (SCRATCH_SECINFO.FLAGS.PT == PT_SS_FIRST)
    THEN
        (* Check that valid RSTORSSP token exists *)
        IF (DS:RCX[4095:4088] != ((TMP_LINADDR + 0x1000) | DS:TMP_SECS.ATTRIBUTES.MODE64BIT)) #GP(0); Fl;
    ELSE
        (* Check the 8 bytes are zero *)
        IF (DS:RCX[4095:4088] != 0) #GP(0); Fl;
    Fl;
IF (SCRATCH_SECINFO.FLAGS.W = 0 OR SCRATCH_SECINFO.FLAGS.R = 0 OR
    SCRATCH_SECINFO.FLAGS.X = 1) #GP(0); Fl;
BREAK;
ESAC;

(* Check the enclave offset is within the enclave linear address space *)
IF (TMP_LINADDR < DS:TMP_SECS.BASEADDR or
    TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE)
    THEN #GP(0); Fl;

(* Check concurrency of measurement resource*)
IF (Measurement being updated)
    THEN #GP(0); Fl;

(* Check if the enclave to which the page will be added is already in Initialized state *)
IF (DS:TMP_SECS already initialized)
    THEN #GP(0); Fl;

(* For TCS pages, force EPCM.rwx bits to 0 and no debug access *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        SCRATCH_SECINFO.FLAGS.R := 0;
        SCRATCH_SECINFO.FLAGS.W := 0;
        SCRATCH_SECINFO.FLAGS.X := 0;
        (DS:RCX).FLAGS.DBGOPTIN := 0; // force TCS.FLAGS.DBGOPTIN off
        DS:RCX.CSSA := 0;
        DS:RCX.AEP := 0;
        DS:RCX.STATE := 0;
    Fl;

(* Add enclave offset and security attributes to MRENCLAVE *)
TMP_ENCLAVEOFFSET := TMP_LINADDR - DS:TMP_SECS.BASEADDR;
TMPUPDATEFIELD[63:0] := 0000000044444145H; // “EADD”
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] := SCRATCH_SECINFO[375:0]; // 48 bytes
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave’s MRENCLAVE update counter;

(* Add enclave offset and security attributes to MRENCLAVE *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;
(*) associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(*) Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).VALID := 1;

Flags Affected
None

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If an enclave memory operand is outside of the EPC.
If an enclave memory operand is the wrong type.
If a memory operand is locked.
If the enclave is initialized.
If the enclave's MRENCLAVE is locked.
If the TCS page reserved bits are set.
If the TCS page PREVSSP field is not zero.
If the PT_SS_REST or PT_SS_REST page is the first or last page in the enclave.
If the PT_SS_FIRST or PT_SS_REST page is not initialized correctly.

#PF(error code) If a page fault occurs in accessing memory operands.
If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0) If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If an enclave memory operand is outside of the EPC.
If an enclave memory operand is the wrong type.
If a memory operand is locked.
If the enclave is initialized.
If the enclave's MRENCLAVE is locked.
If the TCS page reserved bits are set.
If the TCS page PREVSSP field is not zero.
If the PT_SS_REST or PT_SS_REST page is the first or last page in the enclave.
If the PT_SS_FIRST or PT_SS_REST page is not initialized correctly.

#PF(error code) If a page fault occurs in accessing memory operands.
If the EPC page is valid.
**EAUG—Add a Page to an Initialized Enclave**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = ODH ENCLS[EAUG]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX2</td>
<td>This leaf function adds a page to an initialized enclave.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EAUG (In)</td>
<td>Address of a PAGEINFO (In)</td>
<td>Address of the destination EPC page (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPT-COPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

**EAUG Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>PAGEINFO</th>
<th>PAGEINFO.SECS</th>
<th>PAGEINFO.SRCPGE</th>
<th>PAGEINFO.SECINFO</th>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read/Write access permitted by Enclave</td>
<td>Must be zero</td>
<td>Read access permitted by Non Enclave</td>
<td>Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EAUG Faulting Conditions**

| The operands are not properly aligned. | Unsupported security attributes are set. |
| Refers to an invalid SECS. | Reference is made to an SECS that is locked by another thread. |
| The EPC page is locked by another thread. | RCX does not contain an effective address of an EPC page. |
| The EPC page is already valid. | The specified enclave offset is outside of the enclave address space. |
| The SECS has been initialized. |

**Concurrency Restrictions**

**Table 38-10. Base Concurrency Restrictions of EAUG**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAUG</td>
<td>Target [DS:RCX]</td>
<td>Exclusive #GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECS</td>
<td>[DS:RBX]PAGEINFO.SECS</td>
<td>Shared #GP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EAUG Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of the SECS destination page.</td>
</tr>
<tr>
<td>TMP_SECINFO</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of an SECINFO structure which contains security attributes of the page to be added.</td>
</tr>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:TMP_SECINFO.</td>
</tr>
<tr>
<td>TMP_LINADDR</td>
<td>Unsigned Integer</td>
<td>64</td>
<td>Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 32Byte Aligned)
THEN #GP(O); Fl;

IF (DS:RCX is not 4KByte Aligned)
THEN #GP(O); Fl;

IF (DS:RCX does not resolve within an EPC)
THEN #PF(DS:RCX); Fl;

TMP_SECS := DS:RBX.SECONDS;
TMP_SECINFO := DS:RBX.SECINFO;
IF (DS:RBX.SECINFO is not 0)
THEN
  IF (DS:TMP_SECINFO is not 64B aligned)
    THEN #GP(O); Fl;

Fl;

TMP_LINADDR := DS:RBX.LINADDR;

IF (DS:TMP_SECS is not 4Kbyte aligned or TMP_LINADDR is not 4Kbyte aligned )
THEN #GP(O); Fl;

IF DS:RBX.SRCPAGE is not 0
THEN #GP(O); Fl;

IF (DS:TMP_SECS does not resolve within an EPC)
THEN #PF(DS:TMP_SECS); Fl;

(* Check the EPC page for concurrency *)

Table 38-11. Additional Concurrency Restrictions of EAUG

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>Additional Concurrency Restrictions vs. EADD, EEXTEND, EINIT</th>
<th>Additional Concurrency Restrictions vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
<td>On Conflict</td>
<td>Access</td>
</tr>
<tr>
<td>EAUG</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RBX]PAGEINFO:SECS</td>
<td>Concurrent</td>
<td></td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
IF (EPC page in use) THEN IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>) THEN
VMCS.Exit_reason := SGX_CONFLICT;
VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
VMCS.Exit_qualification.error := 0;
VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
VMCS.Guest-linear_address := DS:RCX;
Deliver VMEXIT;
ELSE
#GP(0);
FI:
FI:

IF (EPCM(DS:RCX).VALID ≠ 0) THEN #PF(DS:RCX); FI;

(* copy SECINFO contents into a scratch SECINFO *)
IF (DS:RBX.SECINFO is 0) THEN
(* allocate and initialize a new scratch SECINFO structure *)
SCRATCH_SECINFO.PT := PT_REG;
SCRATCH_SECINFO.R := 1;
SCRATCH_SECINFO.W := 1;
SCRATCH_SECINFO.X := 0;
<< zero out remaining fields of SCRATCH_SECINFO >>
ELSE
(* copy SECINFO contents into scratch SECINFO *)
SCRATCH_SECINFO := DS:TMP_SECINFO;
(* check SECINFO flags for misconfiguration *)
(* reserved fields must be zero *)
(* SECINFO.FLAGS.PT must either be PT_SS_FIRST, or PT_SS_REST *)
IF ( (SCRATCH_SECINFO reserved fields are not 0) or
CPUID.(EAX=12H, ECX=1):EAX[6] is 0) OR
(SCRATCH_SECINFO.PT is not PT_SS_FIRST, or PT_SS_REST) OR
( (SCRATCH_SECINFO.FLAGS.R is 0) OR (SCRATCH_SECINFO.FLAGS.W is 0) OR (SCRATCH_SECINFO.FLAGS.X is 1) ))
THEN #GP(0); FI;
FI;

(* Check if PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) AND CR4.CET == 0 )
THEN #GP(0); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EAUG)
THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
THEN #PF(DS:TMP_SECS); FI;

(* Check if the enclave to which the page will be added is in the Initialized state *)
IF (DS:TMP_SECS is not initialized)
THEN #GP(0); FI;
(* Check the enclave offset is within the enclave linear address space *)
IF ( (TMP_LINADDR < DS:TMP_SECS.BASEADDR) or (TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE) )
    THEN #GP(0); FI;

IF ( (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) )
THEN
    (* SS pages cannot created on first or last page of ELRANGE *)
    IF ( TMP_LINADDR == DS:TMP_SECS.BASEADDR OR
         TMP_LINADDR == (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
        THEN
            #GP(0); FI;
    FI;

(* Clear the content of EPC page*)
DS:RCX[32767:0] := 0;

IF (CPUID.(EAX=07H, ECX=0H):ECX[CET_SS] = 1)
    THEN
        (* set up shadow stack RSTORSSP token *)
        IF (SCRATCH_SECINFO.PT is PT_SS_FIRST)
            THEN
                DS:RCX[0xFF8] := (TMP_LINADDR + 0x1000) | TMP_SECS.ATTRIBUTES.MODE64BIT; FI;
        FI;
    FI;

(* Set EPCM security attributes *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_LINADDR;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 1;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM valid fields *)
EPCM(DS:RCX).VALID := 1;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
    If a memory operand is not properly aligned.
    If a memory operand is locked.
    If the enclave is not initialized.

#PF(error code) If a page fault occurs in accessing memory operands.
64-Bit Mode Exceptions

#GP(0)      If a memory operand is non-canonical form.
            If a memory operand is not properly aligned.
            If a memory operand is locked.
            If the enclave is not initialized.

#PF(error code)  If a page fault occurs in accessing memory operands.
EBLOCK—Mark a page in EPC as Blocked

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 09H ENCLS[EBLOCK]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function marks a page in the EPC as blocked.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EBLOCK (In)</td>
<td>Return error code (Out)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function causes an EPC page to be marked as BLOCKED. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

An error code is returned in RAX.

The table below provides additional information on the memory parameter of EBLOCK leaf function.

**EBLOCK Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The error codes are:

**Table 38-12. EBLOCK Return Value in RAX**

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EBLOCK successful.</td>
</tr>
<tr>
<td>SGX_BLKSTATE</td>
<td>Page already blocked. This value is used to indicate to a VMM that the page was already in BLOCKED state as a result of EBLOCK and thus will need to be restored to this state when it is eventually reloaded (using ELDB).</td>
</tr>
<tr>
<td>SGX_ENTRYEPOCH_LOCKED</td>
<td>SECS locked for Entry Epoch update. This value indicates that an ETRACK is currently executing on the SECS. The EBLOCK should be reattempted.</td>
</tr>
<tr>
<td>SGX_NOTBLOCKABLE</td>
<td>Page type is not one which can be blocked.</td>
</tr>
<tr>
<td>SGX_PG_INVLD</td>
<td>Page is not valid and cannot be blocked.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.</td>
</tr>
</tbody>
</table>

**Concurrency Restrictions**

**Table 38-13. Base Concurrency Restrictions of EBLOCK**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBLOCK</td>
<td>Target [DS:RCX]</td>
<td>Access On Conflict SGX_CONFLICT_vm Exit Qualification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shared</td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EBLOCK Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_BLKSTATE</td>
<td>Integer</td>
<td>64</td>
<td>Page is already blocked.</td>
</tr>
</tbody>
</table>

IF (DS:RCX is not 4KByte Aligned)
   THEN #GP(0); Fi;

IF (DS:RCX does not resolve within an EPC)
   THEN #PF(DS:RCX); Fi;

RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;

(* Check the EPC page for concurrency*)
IF (EPC page in use)
   THEN
      RFLAGS.ZF := 1;
      RAX := SGX_EPC_PAGE_CONFLICT;
      GOTO DONE;
   Fi;

IF (EPCM(DS:RCX).VALID = 0)
   THEN
      RFLAGS.ZF := 1;
      RAX := SGX_PG_INVLD;
      GOTO DONE;
   Fi;

IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_TRIM) and EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
   THEN
      RFLAGS.CF := 1;
      IF (EPCM(DS:RCX).PT = PT_SECS)
         THEN RAX := SGX_PG_IS_SECS;
         ELSE RAX := SGX_NOTBLOCKABLE;
      Fi;
      GOTO DONE;
   Fi;

(* Check if the page is already blocked and report blocked state *)
TMP_BLKSTATE := EPCM(DS:RCX).BLOCKED;
(* at this point, the page must be valid and PT_TCS or PT_REG or PT_TRIM*)

IF (TMP_BLKSTATE = 1)
    THEN
        RFLAGS.CF := 1;
        RAX := SGX_BLKSTATE;
    ELSE
        EPCM(DS:RCX).BLOCKED := 1
    FI;

DONE:

Flags Affected
Sets ZF if SECS is in use or invalid, otherwise cleared. Sets CF if page is BLOCKED or not blockable, otherwise cleared. Clears PF, AF, OF, SF.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If the specified EPC resource is in use.
#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.

64-Bit Mode Exceptions
#GP(0) If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If the specified EPC resource is in use.
#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
ECREATE—Create an SECS page in the Enclave Page Cache

**Op Code/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ECREATE (In)</td>
<td>Address of a PAGEINFO (In)</td>
<td>Address of the destination SECS page (In)</td>
</tr>
</tbody>
</table>

**Description**

ENCLS[ECREATE] is the first instruction executed in the enclave build process. ECREATE copies an SECS structure outside the EPC into an SECS page inside the EPC. The internal structure of SECS is not accessible to software.

ECREATE will set up fields in the protected SECS and mark the page as valid inside the EPC. ECREATE initializes or checks unused fields.

Software sets the following fields in the source structure: SECS:BASEADDR, SECS:SIZE in bytes, ATTRIBUTES, CONFIGID, and CONFIGSVN. SECS:BASEADDR must be naturally aligned on an SECS:SIZE boundary. SECS:SIZE must be at least 2 pages (8192).

The source operand RBX contains an effective address of a PAGEINFO structure. PAGEINFO contains an effective address of a source SECS and an effective address of an SECINFO. The SECS field in PAGEINFO is not used.

The RCX register is the effective address of the destination SECS. It is an address of an empty slot in the EPC. The SECS structure must be page aligned. SECINFO flags must specify the page as an SECS page.

**ECREATE Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>PAGEINFO</th>
<th>PAGEINFO.SRCPAGE</th>
<th>PAGEINFO.SECINFO</th>
<th>ECPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read access permitted by Non Enclave</td>
<td>Read access permitted by Non Enclave</td>
<td>Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

ECREATE will fault if the SECS target page is in use; already valid; outside the EPC. It will also fault if addresses are not aligned; unused PAGEINFO fields are not zero.

If the amount of space needed to store the SSA frame is greater than the amount specified in SECS.SSAFRAME-SIZE, a #GP(0) results. The amount of space needed for an SSA frame is computed based on DS:TMP_SECS.ATTRIBUTES.XFRM size. Details of computing the size can be found Section 39.7.

**Concurrency Restrictions**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECREATE</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>
Temp Variables in ECREATE Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SRCPGE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of the SECS source page.</td>
</tr>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of the SECS destination page.</td>
</tr>
<tr>
<td>TMP_SECINFO</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of an SECINFO structure which contains security attributes of the SECS page to be added.</td>
</tr>
<tr>
<td>TMP_XSIZE</td>
<td>SSA Size</td>
<td>64</td>
<td>The size calculation of SSA frame.</td>
</tr>
<tr>
<td>TMP_MISC_SIZE</td>
<td>MISC Field Size</td>
<td>64</td>
<td>Size of the selected MISC field components.</td>
</tr>
<tr>
<td>TMPUPDATEFIELD</td>
<td>SHA256 Buffer</td>
<td>512</td>
<td>Buffer used to hold data being added to TMP_SECS.MRENCLAVE.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); Fi;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); Fi;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fi;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECINFO := DS:RBX.SECINFO;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECINFO is not 64Byte aligned)
    THEN #GP(0); Fi;

IF (DS:RBX.LINADDR ≠ 0 or DS:RBX.SECS ≠ 0)
    THEN #GP(0); Fi;

(* Check for misconfigured SECINFO flags*)
IF (DS:TMP_SECINFO reserved fields are not zero or DS:TMP_SECINFO.FLAGS.PT ≠ PT_SECS)
    THEN #GP(0); Fi;

TMP_SECS := RCX;

IF (EPC entry in use)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
VMCSEXit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
VMCSEXit_qualification.error := 0;
VMCS.Guest-physical_address :=
   << translation of DS:TMP_SECS produced by paging >>;
VMCS.Guest-linear_address := DS:TMP_SECS;
Deliver VMEXIT;
ELSE
   #GP(0);
FI;

IF (EPC entry in use)
   THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
   THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] := DS:TMP_SRCPE[32767:0];

(* Check lower 2 bits of XFRM are set *)
IF ( (DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
   THEN #GP(0); FI;

IF (XFRM is illegal)
   THEN #GP(0); FI;

(* Check legality of CET_ATTRIBUTES *)
IF ((DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_ATTRIBUTES ≠ 0) ||
   (DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
   (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
   (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[5:2] ≠ 0) ||
   (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[1:0] ≠ 0) ||
   (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1 and
    DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) not canonical) ||
   (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0 and
    (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) & 0xFFFFFFFF00000000) ||
   ((DS:TMP_SECS.CET_ATTRIBUTES.reserved fields not 0) or
    (DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) is not page aligned))
   THEN
      #GP(0);
   FI;

(* Make sure that the SECS does not have any unsupported MISCSELECT options*)
IF ( ((CPUID.(EAX=12H, ECX=0):EBX[31:0] & DS:TMP_SECS.MISCSELECT[31:0]) )
   THEN
      #GP(0);
   FI;

(* Compute size of MISC area *)
TMP_MISC_SIZE := compute_misc_region_size();

(* Compute the size required to save state of the enclave on async exit, see Section 39.7.2.2*)
TMP_XSIZE := compute_xsave_size(DS:TMP_SECS.ATTRIBUTES.XFRM) + GPR_SIZE + TMP_MISC_SIZE;

(* Ensure that the declared area is large enough to hold XSAVE and GPR stat *)
IF ( DS:TMP_SECS.SSAFRAMESIZE*4096 < TMP_XSIZE)
    THEN #GP(0); FI;

IF ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.BASEADDR is not canonical))
    THEN #GP(0); FI;

IF ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.BASEADDR and 0FFFFFFFF00000000H) )
    THEN #GP(0); FI;

IF ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) )
    THEN #GP(0); FI;

IF ( DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) )
    THEN #GP(0); FI;

(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
    THEN #GP(0); FI;

(* Ensure base address of an enclave is aligned on size*)
IF ( (( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) )
    THEN #GP(0); FI;

(* Ensure the SECS does not have any unsupported attributes*)
IF ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
    THEN #GP(0); FI;

IF ( DS:TMP_SECS reserved fields are not zero)
    THEN #GP(0); FI;

(* Verify that CONFIGID/CONFIGSVN are not set with attribute *)
IF ( ((DS:TMP_SECS.CONFIGID ≠ 0) or (DS:TMP_SECS.CONFIGSVN ≠ 0)) AND (DS:TMP_SECS.ATTRIBUTES.KSS == 0))
    THEN #GP(0); FI;

Clear DS:TMP_SECS to Uninitialized;
DS:TMP_SECS.MRENCLAVE := SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
DS:TMP_SECS.ISVSVN := 0;
DS:TMP_SECS.ISVPROID := 0;

(* Initialize hash updates etc*)
Initialize enclave's MRENCLAVE update counter;

(* Add “ECREATE” string and SECS fields to MRENCLAVE *)
TMPUPDATEFIELD[63:0] := 0045544145524345H; // “ECREATE”
TMPUPDATEFIELD[95:64] := DS:TMP_SECS.SSAFRAMESIZE;
TMPUPDATEFIELD[159:96] := DS:TMP_SECS.SIZE;
IF (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 1)
    THEN
        ELSE
        TMPUPDATEFIELD[223:160] := 0;

Vol. 3D  38-33
INTEL® SGX INSTRUCTION REFERENCES

Fl:
TMPUPDATEFIELD[511:160] := 0;
DS:TMP_SECS.MRENCLAVE := SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Set EID *)
DS:TMP_SECS.EID := LockedXAdd(CR_NEXT_EID, 1);

(* Initialize the virtual child count to zero *)
DS:TMP_SECS.VIRTCILDNT := 0;

(* Load ENCLAVECONTEXT with Address out of paging of SECS *)
<< store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>

(* Set the EPCM entry, first create SECS identifier and store the identifier in EPCM *)
EPCM(DS:TMP_SECS).PT := PT_SECS;
EPCM(DS:TMP_SECS).ENCLAVEADDRESS := 0;
EPCM(DS:TMP_SECS).R := 0;
EPCM(DS:TMP_SECS).W := 0;
EPCM(DS:TMP_SECS).X := 0;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).VALID := 1;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If the reserved fields are not zero.
If PAGEINFO.SECS is not zero.
If PAGEINFO.LINADDR is not zero.
If the SECS destination is locked.
If SECS.SSAFRAMESIZE is insufficient.

#PF(error code) If a page fault occurs in accessing memory operands.
If the SECS destination is outside the EPC.

64-Bit Mode Exceptions
#GP(0) If a memory address is non-canonical form.
If a memory operand is not properly aligned.
If the reserved fields are not zero.
If PAGEINFO.SECS is not zero.
If PAGEINFO.LINADDR is not zero.
If the SECS destination is locked.
If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)  If a page fault occurs in accessing memory operands.
If the SECS destination is outside the EPC.
EDBGRD—Read From a Debug Enclave

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 04H ENCLS[EDBGRD]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function reads a dword/quadword from a debug enclave.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EDBGRD (In)</td>
<td>Return error code (Out)</td>
<td>Data read from a debug enclave (Out)</td>
</tr>
</tbody>
</table>

### Description

This leaf function copies a quadword/doubleword from an EPC page belonging to a debug enclave into the RBX register. Eight bytes are read in 64-bit mode, four bytes are read in non-64-bit modes. The size of data read cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

### EDBGRD Memory Parameter Semantics

<table>
<thead>
<tr>
<th>EPCQw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The error codes are:

### Table 38-17. EDBGRD Return Value in RAX

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EDBGRD successful.</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_DEBUGGABLE</td>
<td>The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

### EDBGRD Faulting Conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCX points into a page that is an SECS.</td>
<td>RCX does not resolve to a naturally aligned linear address.</td>
</tr>
<tr>
<td>RCX points to a page that does not belong to an enclave that is in debug mode.</td>
<td>RCX points to a location inside a TCS that is beyond the architectural size of the TCS (SGX_TCS_LIMIT).</td>
</tr>
<tr>
<td>An operand causing any segment violation.</td>
<td>May page fault.</td>
</tr>
<tr>
<td>CPL &gt; 0.</td>
<td></td>
</tr>
</tbody>
</table>

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.
Concurrency Restrictions

Table 38-18. Base Concurrency Restrictions of EDBGRD

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>EDBGRD</td>
<td>Target [DS:RCX]</td>
<td>Shared</td>
</tr>
</tbody>
</table>

Table 38-19. Additional Concurrency Restrictions of EDBGRD

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>EDBGRD</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Operation

Temp Variables in EDBGRD Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_MODE64</td>
<td>Binary</td>
<td>1</td>
<td>((IA32_EFER.LMA = 1) &amp;&amp; (CS.L = 1))</td>
</tr>
<tr>
<td>TMP_SECS</td>
<td>64</td>
<td></td>
<td>Physical address of SECS of the enclave to which source operand belongs.</td>
</tr>
</tbody>
</table>

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); Fl;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); Fl;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fl;

(* make sure no other Intel SGX instruction is accessing the same EPCM entry *)
IF (Another instruction modifying the same EPCM entry is executing)
    THEN #GP(0); Fl;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); Fl;

(* make sure that DS:RCX (SOURCE) is pointing to a PT_REG or PT_TCS or PT_VA or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT # PT_REG) and (EPCM(DS:RCX).PT # PT_TCS) and (EPCM(DS:RCX).PT # PT_VA) and (EPCM(DS:RCX).PT # PT_SS_FIRST) and (EPCM(DS:RCX).PT # PT_SS_REST))
    THEN #PF(DS:RCX); Fl;

(* make sure that DS:RCX points to an accessible EPC page *)
IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
INTEL® SGX INSTRUCTION REFERENCES

```
RAX := SGX_PAGE_NOT_DEBUGGABLE;
GOTO DONE;
FI;

(* If source is a TCS, then make sure that the offset into the page is not beyond the TCS size*)
IF ( ( EPCM(DS:RCX).PT = PT_TCS) and ((DS:RCX) & FFFH ≥ SGX_TCS_LIMIT) )
    THEN #GP(0); FI;

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF ( (EPCM(DS:RCX).PT = PT_REG) or (EPCM(DS:RCX).PT = PT_TCS) )
    THEN
        TMP_SECS := GET_SECS_ADDRESS;
        IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
            THEN #GP(0); FI;
        IF ( (TMP_MODE64 = 1) )
            THEN RBX[63:0] := (DS:RCX)[63:0];
            ELSE EBX[31:0] := (DS:RCX)[31:0];
            FI;
    ELSE
        TMP_64BIT_VAL[63:0] := (DS:RCX)[63:0] & (~07H); // Read contents from VA slot
        IF (TMP_MODE64 = 1)
            THEN
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN RBX[63:0] := 0FFFFFFFFFFFFFFFFH;
                    ELSE RBX[63:0] := 0H;
                FI;
                ELSE
                    IF (TMP_64BIT_VAL ≠ 0H)
                        THEN EBX[31:0] := 0FFFFFFFFH;
                        ELSE EBX[31:0] := 0H;
                    FI;
            FI;
    FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected
ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set to 0. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions
#GP(0) If the address in RCS violates DS limit or access rights.
If DS segment is unusable.
If RCX points to a memory location not 4Byte-aligned.
If the address in RCX points to a page belonging to a non-debug enclave.
If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
```
#PF(error code) If a page fault occurs in accessing memory operands.
If the address in RCX points to a non-EPC page.
If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0) If RCX is non-canonical form.
If RCX points to a memory location not 8Byte-aligned.
If the address in RCX points to a page belonging to a non-debug enclave.
If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.

#PF(error code) If a page fault occurs in accessing memory operands.
If the address in RCX points to a non-EPC page.
If the address in RCX points to an invalid EPC page.
EDBGWR—Write to a Debug Enclave

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 05H ENCLS[EDBGWR]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function writes a dword/quadword to a debug enclave.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EDBGWR (In)</td>
<td>Return error code (Out)</td>
<td>Data to be written to a debug enclave (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden. The effective address of the target location inside the EPC is provided in the register RCX.

**EDBGWR Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>EPCQW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EDBGWR Faulting Conditions**

- RCX points into a page that is an SECS. RCX does not resolve to a naturally aligned linear address.
- RCX points to a page that does not belong to an enclave that is in debug mode. RCX points to a location inside a TCS that is not the FLAGS word.
- An operand causing any segment violation. May page fault.
- CPL > 0.

The error codes are:

**Table 38-20. EDBGWR Return Value in RAX**

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EDBGWR successful.</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_DEBUGGABLE</td>
<td>The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.</td>
</tr>
</tbody>
</table>

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.
Concurrent Restrictions

**Table 38-21. Base Concurrency Restrictions of EDBGWR**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
</table>

**Table 38-22. Additional Concurrency Restrictions of EDBGWR**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDBGWR</td>
<td>Target [DS:RCX]</td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPR, EMODT: Concurrent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vs. EADD, EEXTEND, EINIT: Access, On Conflict</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vs. ETRACK, ETRACKC: Access, On Conflict</td>
</tr>
</tbody>
</table>

Operation

**Temp Variables in EDBGWR Operational Flow**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_MODE64</td>
<td>Binary</td>
<td>1</td>
<td>((\text{IA32}_{-}\text{EFER.LMA} = 1) &amp;&amp; (\text{CS.L} = 1))).</td>
</tr>
<tr>
<td>TMP_SECS</td>
<td></td>
<td>64</td>
<td>Physical address of SECS of the enclave to which source operand belongs.</td>
</tr>
</tbody>
</table>

\[
\text{TMP\_MODE64} := (\text{IA32\_EFER.LMA} = 1) \&\& (\text{CS.L} = 1); \]

\[
\text{IF} \ (\text{TMP\_MODE64} = 1) \text{ and } (\text{DS\_RCX} \text{ is not 8Byte Aligned}) \]
\[
\quad \text{THEN} \ #\GP(0); \ \text{Fi}; \]

\[
\text{IF} \ (\text{TMP\_MODE64} = 0) \text{ and } (\text{DS\_RCX} \text{ is not 4Byte Aligned}) \]
\[
\quad \text{THEN} \ #\GP(0); \ \text{Fi}; \]

\[
\text{IF} \ (\text{DS\_RCX} \text{ does not resolve within an EPC}) \]
\[
\quad \text{THEN} \ #\PF(\text{DS\_RCX}); \ \text{Fi}; \]

\[
\left(\ast \text{ make sure no other Intel SGX instruction is accessing the same EPCM entry } \ast\right) \]
\[
\text{IF} \ (\text{Another instruction modifying the same EPCM entry is executing}) \]
\[
\quad \text{THEN} \ #\GP(0); \ \text{Fi}; \]

\[
\text{IF} \ (\text{EPCM(\text{DS\_RCX}).VALID = 0}) \]
\[
\quad \text{THEN} \ #\PF(\text{DS\_RCX}); \ \text{Fi}; \]

\[
\left(\ast \text{ make sure that DS\_RCX (DST) is pointing to a PT\_REG or PT\_TCS or PT\_SS\_FIRST or PT\_SS\_REST } \ast\right) \]
\[
\text{IF} \ ((\text{EPCM(\text{DS\_RCX}).PT \ # PT\_REG}) \text{ and } (\text{EPCM(\text{DS\_RCX}).PT \ # PT\_TCS}) \text{ and } (\text{EPCM(\text{DS\_RCX}).PT \ # PT\_SS\_FIRST}) \text{ and } (\text{EPCM(\text{DS\_RCX}).PT \ # PT\_SS\_REST})) \]
\[
\quad \text{THEN} \ #\PF(\text{DS\_RCX}); \ \text{Fi}; \]

\[
\left(\ast \text{ make sure that DS\_RCX points to an accessible EPC page } \ast\right) \]
\[
\text{IF} \ ((\text{EPCM(\text{DS\_RCX}).PENDING is not 0}) \text{ or } (\text{EPCM(\text{DS\_RCX}).MODIFIED is not 0})) \]
\[
\quad \text{THEN} \quad \text{RFLAGS.ZF} := 1; \]
RAX := SGX_PAGE_NOT_DEBUGGABLE;
GOTO DONE;
FI;

(* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FF8H ≠ offset_of_FLAGS & 0FF8H) )
    THEN #GP(0); FI;

(* Locate the SECS for the enclave to which the DS:RCX page belongs *)
TMP_SECS := GET_SECS_PHYS_ADDRESS(EPCM(DS:RCX).ENCLAVESECS);

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 1) )
    THEN (DS:RCX)[63:0] := RBX[63:0];
    ELSE (DS:RCX)[31:0] := EBX[31:0];
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0

Flags Affected
ZF is set if the page is MODIFIED or PENDING; RAX contains the error code. Otherwise ZF is cleared and RAX is set
to 0. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions
#GP(0) If the address in RCS violates DS limit or access rights.
          If DS segment is unusable.
          If RCX points to a memory location not 4Byte-aligned.
          If the address in RCX points to a page belonging to a non-debug enclave.
          If the address in RCX points to a page which is not PT_TCS or PT_REG.
          If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code) If a page fault occurs in accessing memory operands.
          If the address in RCX points to a non-EPC page.
          If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions
#GP(0) If RCX is non-canonical form.
          If RCX points to a memory location not 8Byte-aligned.
          If the address in RCX points to a page belonging to a non-debug enclave.
          If the address in RCX points to a page which is not PT_TCS or PT_REG.
          If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)  If a page fault occurs in accessing memory operands.
If the address in RCX points to a non-EPC page.
If the address in RCX points to an invalid EPC page.
**EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
</tr>
</thead>
</table>
| EAX = 06H
ENCLS[EEXTEND] |

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>EBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EEXTEND (In)</td>
<td>Effective address of the SECS of the data chunk (In)</td>
<td>Effective address of a 256-byte chunk in the EPC (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EXTEND string compromising of “EEXTEND” || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

**EEXTEND Memory Parameter Semantics**

| EPC[RCX] |

Read access by Enclave

The instruction faults if any of the following:

**EEXTEND Faulting Conditions**

<table>
<thead>
<tr>
<th>RBX points to an address not 4KBytes aligned.</th>
<th>RBX does not resolve to an SECS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBX does not point to an SECS page.</td>
<td>RBX does not point to the SECS page of the data chunk.</td>
</tr>
<tr>
<td>RCX points to an address not 256B aligned.</td>
<td>RCX points to an unused page or a SECS.</td>
</tr>
<tr>
<td>RCX does not resolve in an EPC page.</td>
<td>If SECS is locked.</td>
</tr>
<tr>
<td>If the SECS is already initialized.</td>
<td>May page fault.</td>
</tr>
<tr>
<td>CPL &gt; 0.</td>
<td></td>
</tr>
</tbody>
</table>

**Concurrency Restrictions**

<table>
<thead>
<tr>
<th>Table 38-23. Base Concurrency Restrictions of EEXTEND</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEXTEND</td>
<td>Target [DS:RCX]</td>
<td>Shared #GP</td>
</tr>
<tr>
<td>SECS</td>
<td>[DS:RBX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

38-44 Vol. 3D
INTEL® SGX INSTRUCTION REFERENCES

Operation

Temp Variables in EEXTEND Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>64</td>
<td>Physical address of SECS of the enclave to which source operand belongs.</td>
<td></td>
</tr>
<tr>
<td>TMP_ENCLAVEOFFSET</td>
<td>Enclave Offset</td>
<td>64</td>
<td>The page displacement from the enclave base address.</td>
</tr>
<tr>
<td>TMPUPDATEFIELD</td>
<td>SHA256 Buffer</td>
<td>512</td>
<td>Buffer used to hold data being added to TMP_SECS.MRENCLAVE.</td>
</tr>
</tbody>
</table>

Table 38-24. Additional Concurrency Restrictions of EEXTEND

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEXTEND</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RBX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Temp Variables in EEXTEND Operational Flow

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve to an EPC page)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other instructions accessing EPCM)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)  
    and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST) )
    THEN #PF(DS:RCX); FI;

TMP_SECS := Get_SECS_ADDRESS();

IF (DS:RBX does not resolve to TMP_SECS)
    THEN #GP(0); FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS))
THEN #GP(0); FI;

(* Calculate enclave offset *)
TMP_ENCLAVEOFFSET := EPCM(DS:RCX).ENCLAVEADDRESS - TMP_SECS.BASEADDR;
TMP_ENCLAVEOFFSET := TMP_ENCLAVEOFFSET + (DS:RCX & 0FFFH)

(* Add EEXTEND message and offset to MRENCLAVE *)
TMPUPDATEFIELD[63:0] := 00444E4554454548; // "EEXTEND"
TMPUPDATEFIELD[127:64] := TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] := 0; // 48 bytes
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(*Add 256 bytes to MRENCLAVE, 64 byte at a time *)
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[511:0] );
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1023: 512] );
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1535: 1024] );
TMP_SECS.MRENCLAVE := SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[2047: 1536] );
INC enclave's MRENCLAVE update counter by 4;

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0) If the address in RBX is outside the DS segment limit.
If RBX points to an SECS page which is not the SECS of the data chunk.
If the address in RCX is outside the DS segment limit.
If RCX points to a memory location not 256Byte-aligned.
If another instruction is accessing MRENCLAVE.
If another instruction is checking or updating the SECS.
If the enclave is already initialized.

#PF(error code) If a page fault occurs in accessing memory operands.
If the address in RBX points to a non-EPC page.
If the address in RCX points to a page which is not PT_TCS or PT_REG.
If the address in RCX points to a non-EPC page.
If the address in RCX points to an invalid EPC page.

**64-Bit Mode Exceptions**

#GP(0) If RBX is non-canonical form.
If RBX points to an SECS page which is not the SECS of the data chunk.
If RCX is non-canonical form.
If RCX points to a memory location not 256 Byte-aligned.
If another instruction is accessing MRENCLAVE.
If another instruction is checking or updating the SECS.
If the enclave is already initialized.

#PF(error code) If a page fault occurs in accessing memory operands.
If the address in RBX points to a non-EPC page.
If the address in RCX points to a page which is not PT_TCS or PT_REG.
If the address in RCX points to a non-EPC page.
If the address in RCX points to an invalid EPC page.
**EINIT—Initialize an Enclave for Execution**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 02H ENCLS[EINIT]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function initializes the enclave and makes it ready to execute enclave code.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EINIT (In)</td>
<td>Error code (Out)</td>
<td>Address of SIGSTRUCT (In)</td>
<td>Address of SECS (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function is the final instruction executed in the enclave build process. After EINIT, the MRENCLAVE measurement is complete, and the enclave is ready to start user code execution using the EENTER instruction.

EINIT takes the effective address of a SIGSTRUCT and EINITTOKEN. The SIGSTRUCT describes the enclave including MRENCLAVE, ATTRIBUTES, ISVSVN, a 3072 bit RSA key, and a signature using the included key. SIGSTRUCT must be populated with two values, q1 and q2. These are calculated using the formulas shown below:

\[
q_1 = \text{floor}(\text{Signature}^2 / \text{Modulus});
\]

\[
q_2 = \text{floor}((\text{Signature}^3 - q_1 * \text{Signature} * \text{Modulus}) / \text{Modulus});
\]

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

---

![Figure 38-1. Relationships Between SECS, SIGSTRUCT, and EINITTOKEN](image-url)
EINIT performs the following steps, which can be seen in Figure 38-1:

1. Validates that SIGSTRUCT is signed using the enclosed public key.
2. Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.
3. Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.
4. Checks that the result of bitwise and-ing SIGSTRUCT.ATTRIBUTEMASK with SIGSTRUCT.ATTRIBUTES equals the result of bitwise and-ing SIGSTRUCT.ATTRIBUTEMASK with SECS.ATTRIBUTES.
5. If EINITTOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.
6. If EINITTOKEN.VALID is 1, checks the validity of EINITTOKEN.
7. If EINITTOKEN.VALID is 1, checks that EINITTOKEN.MRENCLAVE equals SECS.MRENCLAVE.
8. If EINITTOKEN.VALID is 1 and EINITTOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.
9. Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.
10. Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX_UNMASKED_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EINIT successful.</td>
</tr>
<tr>
<td>SGX_INVALID_SIG_STRUCT</td>
<td>If SIGSTRUCT contained an invalid value.</td>
</tr>
<tr>
<td>SGX_INVALID_ATTRIBUTE</td>
<td>If SIGSTRUCT contains an unauthorized attributes mask.</td>
</tr>
<tr>
<td>SGX_INVALID_MEASUREMENT</td>
<td>If SIGSTRUCT contains an incorrect measurement.</td>
</tr>
<tr>
<td></td>
<td>If EINITTOKEN contains an incorrect measurement.</td>
</tr>
<tr>
<td>SGX_INVALID_SIGNATURE</td>
<td>If signature does not validate with enclosed public key.</td>
</tr>
<tr>
<td>SGX_INVALID_LICENSE</td>
<td>If license is invalid.</td>
</tr>
<tr>
<td>SGX_INVALID_CPUSVN</td>
<td>If license SVN is unsupported.</td>
</tr>
<tr>
<td>SGX_UNMASKED_EVENT</td>
<td>If an unmasked event is received before the instruction completes its operation.</td>
</tr>
</tbody>
</table>
Concurrent Restrictions

Table 38-26. Base Concurrency Restrictions of EINIT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINIT</td>
<td>SECS [DS:RCX]</td>
<td></td>
<td></td>
<td></td>
<td>#GP</td>
</tr>
</tbody>
</table>

Table 38-27. Additional Concurrency Restrictions of EINIT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
<th>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
</table>

Operation

Temp Variables in EINIT Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SIG</td>
<td>SIGSTRUCT</td>
<td>1808Bytes</td>
<td>Temp space for SIGSTRUCT.</td>
</tr>
<tr>
<td>TMP_TOKEN</td>
<td>EINITTOKEN</td>
<td>304Bytes</td>
<td>Temp space for EINITTOKEN.</td>
</tr>
<tr>
<td>TMP_MRENCLAVE</td>
<td></td>
<td>32Bytes</td>
<td>Temp space for calculating MRENCLAVE.</td>
</tr>
<tr>
<td>TMP_MRSIGNER</td>
<td></td>
<td>32Bytes</td>
<td>Temp space for calculating MRSIGNER.</td>
</tr>
<tr>
<td>CONTROLLED_ATTRIBU</td>
<td>ESTATE ATTRIBU</td>
<td>16Bytes</td>
<td>Constant mask of all ATTRIBUTE bits that can only be set for authorized</td>
</tr>
<tr>
<td>ESTATE ATTRIBUTES</td>
<td>ESTATE</td>
<td></td>
<td>enclaves.</td>
</tr>
<tr>
<td>TMP_KEYDEPENDENCI</td>
<td>Buffer</td>
<td>224Bytes</td>
<td>Temp space for key derivation.</td>
</tr>
<tr>
<td>TMP_EINITTOKENKEY</td>
<td></td>
<td>16Bytes</td>
<td>Temp space for the derived EINITTOKEN Key.</td>
</tr>
<tr>
<td>TMP_SIG_PADDING</td>
<td>PKCS Padding</td>
<td>352Bytes</td>
<td>The value of the top 352 bytes from the computation of Signature^3</td>
</tr>
<tr>
<td></td>
<td>Buffer</td>
<td></td>
<td>modulo MRSIGNER.</td>
</tr>
</tbody>
</table>

(* make sure SIGSTRUCT and SECS are aligned *)

IF ((DS:RBX is not 4KByte Aligned) or (DS:RCX is not 4KByte Aligned))
    THEN #GP(0); Fl;

(* make sure the EINITTOKEN is aligned *)

IF (DS:RDX is not 512Byte Aligned)
    THEN #GP(0); Fl;

(* make sure the SECS is inside the EPC *)

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fl;

TMP_SIG[14463:0] := DS:RBX[14463:0]; // 1808 bytes
TMP_TOKEN[2423:0] := DS:RDX[2423:0]; // 304 bytes
(* Verify SIGSTRUCT Header. *)
IF ((TMP_SIG.HEADER ≠ 06000000E10000000000010000000000h) or 
 ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h)) or 
 (TMP_SIG.EXPONENT ≠ 010100010000000000000000001000000h) or 
 (TMP_SIG.EXPONENT ≠ 00000003h) or (Reserved space is not 0's))
   THEN
     RFLAGS.ZF := 1;
     RAX := SGX_INVALID_SIG_STRUCT;
     GOTO EXIT;
   FI;

(* Open “Event Window” Check for Interrupts. Verify signature using embedded public key, q1, and q2. Save upper 352 bytes of the 
PKCS1.5 encoded message into the TMP_SIG_PADDING*)
IF (interrupt was pending) THEN
  RFLAGS.ZF := 1;
  RAX := SGX_UNMASKED_EVENT;
  GOTO EXIT;
FI
IF (signature failed to verify) THEN
  RFLAGS.ZF := 1;
  RAX := SGX_INVALID_SIGNATURE;
  GOTO EXIT;
FI;

(* Close “Event Window” *)

(* make sure no other Intel SGX instruction is modifying SECS*)
IF (Other instructions modifying SECS) THEN
  THEN GP(0); FI;

IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT ≠ PT_SECS))
  THEN PF(DS:RCX); FI;

(* Verify ISVFAMILYID is not used on an enclave with KSS disabled *)
IF ((TMP_SIG.ISVFAMILYID != 0) AND (DS:RCX.ATTRIBUTES.KSS == 0))
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_SIG_STRUCT;
    GOTO EXIT;
  FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS’s Initialized state))
  THEN #GP(0); FI;

(* Calculate finalized version of MRENCLAVE *)
(* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest *)
TMP_ENCLAVE := SHA256FINAL( (DS:RCX).MRENCLAVE, enclave’s MRENCLAVE update count *512);

(* Verify MRENCLAVE from SIGSTRUCT *)
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
  RFLAGS.ZF := 1;
  RAX := SGX_INVALID_MEASUREMENT;
  GOTO EXIT;
FI;
TMP_MRSIGNER := SHA256(TMP_SIG.MODULUS)

(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
CONTROLLED_ATTRIBUTES := 0000000000000020H;
IF ( ((DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES) ≠ 0) and (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH) )
  RFLAGS.ZF := 1;
  RAX := SGX_INVALID_ATTRIBUTE;
  GOTO EXIT;
FI;

(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
IF ( (DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK) ≠ (TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK) )
  RFLAGS.ZF := 1;
  RAX := SGX_INVALID_ATTRIBUTE;
  GOTO EXIT;
FI;

(* Verify SIGSTRUCT.MISCSELECT requirements are met *)
IF ( (DS:RCX.MISCSELECT & TMP_SIG.MISCMASK) ≠ (TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK) )
  THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_ATTRIBUTE;
    GOTO EXIT
  FI;

  IF ( (DS:RCX.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIBUTES_MASK) ≠ (TMP_SIG.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIB-UTES_MASK) )
    THEN
      RFLAGS.ZF := 1;
      RAX := SGX_INVALID_ATTRIBUTE;
      GOTO EXIT
    FI;
  FI;

(* If EINITTOKEN.VALID[0] is 0, verify the enclave is signed by an authorized key *)
IF (TMP_TOKEN.VALID[0] = 0)
  IF (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
  FI;
  GOTO COMMIT;
FI;

(* Debug Launch Enclave cannot launch Production Enclaves *)
IF ( (DS:RDX.MASKEDATTRIBUTESLE.DEBUG = 1) and (DS:RCX.ATTRIBUTES.DEBUG = 0) )
  RFLAGS.ZF := 1;
  RAX := SGX_INVALID_EINITTOKEN;
  GOTO EXIT;
FI;
(* Check reserve space in EINIT token includes reserved regions and upper bits in valid field *)
IF (TMP_TOKEN reserved space is not clear)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* EINIT token must not have been created by a configuration beyond the current CPU configuration *)
IF (TMP_TOKEN.CPUSVN must not be a configuration beyond CR_CPUSVN)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_CPUSVN;
    GOTO EXIT;
FI;

(* Derive Launch key used to calculate EINITTOKEN.MAC *)
HARDCODED_PKCS1_5_PADDING[15:0] := 0100H;
HARDCODED_PKCS1_5_PADDING[2655:16] := SignExtend330Byte(-1); // 330 bytes of 0FFH
HARDCODED_PKCS1_5_PADDING[2815:2656] := 2004000501020403060D30313000H;

TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_TOKEN.ISVPRODIDLE;
TMP_KEYDEPENDENCIES.ISVSVN := TMP_TOKEN.ISVSVNLE;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_TOKEN.MASKEDATTRIBUTESLE;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := IA32_SGXLEPUBKEYHASH;
TMP_KEYDEPENDENCIES.KEYID := TMP_TOKEN.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := TMP_TOKEN.CPUSVNLE;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_TOKEN.MASKEDMISCSELECTLE;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_TOKEN.CET_MASKED_ATTRIBUTES_LE;
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;

(* Calculate the derived key*)
TMP_EINITTOKENKEY := derivekey(TMP_KEYDEPENDENCIES);

(* Verify EINITTOKEN was generated using this CPU's Launch key and that it has not been modified since issuing by the Launch Enclave. Only 192 bytes of EINITTOKEN are CMACed *)
IF (TMP_TOKEN.MAC ≠ CMAC(TMP_EINITTOKENKEY, TMP_TOKEN[1535:0]))
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;
(* Verify EINITTOKEN (RDX) is for this enclave *)
IF ((TMP_TOKEN.MRENCLAVE ≠ TMP_MRENCLAVE) or (TMP_TOKEN.MRSIGNER ≠ TMP_MRSIGNER))
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

(* Verify ATTRIBUTES in EINITTOKEN are the same as the enclave's *)
IF (TMP_TOKEN.ATTRIBUTES ≠ DS:RCX.ATTRIBUTES)
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_EINIT_ATTRIBUTE;
    GOTO EXIT;
FI;

COMMIT:
(* Commit changes to the SECS; Set ISVPRODID, ISVSIGN, MRSIGNER, INIT ATTRIBUTE fields in SECS (RCX) *)
DS:RCX.MRENCLAVE := TMP_MRENCLAVE;
(* MRSIGNER stores a SHA256 in little endian implemented natively on x86 *)
DS:RCX.MRSIGNER := TMP_MRSIGNER;
DS:RCX.ISVEXTPRODID := TMP_SIG.ISVEXTPRODID;
DS:RCX.ISVPRODID := TMP_SIG.ISVPRODID;
DS:RCX.ISVSIGN := TMP_SIG.ISVSIGN;
DS:RCX.ISVFAMILYID := TMP_SIG.ISVFAMILYID;
DS:RCX.PADDING := TMP_SIG.PADDING;

(* Mark the SECS as initialized *)
Update DS:RCX to initialized;

(* Set RAX and ZF for success*)
    RFLAGS.ZF := 0;
    RAX := 0;
EXIT:
    RFLAGS.CF,PF,AF,OF, SF := 0;

Flags Affected
ZF is cleared if successful, otherwise ZF is set and RAX contains the error code. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions
#GP(0) If a memory operand is not properly aligned.
If another instruction is modifying the SECS.
If the enclave is already initialized.
If the SECS.MRENCLAVE is in use.

#PF(error code) If a page fault occurs in accessing memory operands.
If RCX does not resolve in an EPC page.
If the memory address is not a valid, uninitialized SECS.

64-Bit Mode Exceptions
#GP(0) If a memory operand is not properly aligned.
If another instruction is modifying the SECS.
If the enclave is already initialized.
If the SECS.MRENCLAVE is in use.
#PF(error code) If a page fault occurs in accessing memory operands.
If RCX does not resolve in an EPC page.
If the memory address is not a valid, uninitialized SECS.
ELDB/ELDU/ELDBC/ELDUC—Load an EPC Page and Mark its State

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 07H ENCLS[ELDB]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function loads, verifies an EPC page and marks the page as blocked.</td>
</tr>
<tr>
<td>EAX = 08H ENCLS[ELDU]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function loads, verifies an EPC page and marks the page as unblocked.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th></th>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ELDB/ELDU</td>
<td>Return error code (Out)</td>
<td>Address of the PAGEINFO</td>
<td>Address of the EPC page</td>
<td>Address of the version-array slot (In)</td>
</tr>
</tbody>
</table>

Description

This leaf function copies a page from regular main memory to the EPC. As part of the copying process, the page is cryptographically authenticated and decrypted. This instruction can only be executed when current privilege level is 0.

The ELDB leaf function sets the BLOCK bit in the EPCM entry for the destination page in the EPC after copying. The ELDU leaf function clears the BLOCK bit in the EPCM entry for the destination page in the EPC after copying.

RBX contains the effective address of a PAGEINFO structure; RCX contains the effective address of the destination EPC page; RDX holds the effective address of the version array slot that holds the version of the page.

The ELDBC/ELDUC leafs are very similar to ELDB and ELDU. They provide an error code on the concurrency conflict for any of the pages which need to acquire a lock. These include the destination, SECS, and VA slot.

The table below provides additional information on the memory parameter of ELDB/ELDU leaf functions.

ELDB/ELDU/ELDBC/ELBUC Memory Parameter Semantics

<table>
<thead>
<tr>
<th>PAGEINFO</th>
<th>PAGEINFO.SRCPGE</th>
<th>PAGEINFO.PCMD</th>
<th>PAGEINFO.SECS</th>
<th>ECPAGE</th>
<th>Version-Array Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-enclave read access</td>
<td>Non-enclave read access</td>
<td>Non-enclave read access</td>
<td>Enclave read/write access</td>
<td>Read/Write access permitted by Enclave</td>
<td>Read/Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The error codes are:

Table 38-28. ELDB/ELDU/ELDBC/ELBUC Return Value in RAX

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>ELDB/ELDU successful.</td>
</tr>
<tr>
<td>SGX_MACCOMPARE_FAIL</td>
<td>If the MAC check fails.</td>
</tr>
</tbody>
</table>
Concurrency Restrictions

Table 38-29. Base Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELDB/ELDU</td>
<td>Target [DS:RCX]</td>
<td>Exclusive #GP EPC_PAGE_CONFLICT_EXCEPTION</td>
</tr>
<tr>
<td></td>
<td>VA [DS:RDX]</td>
<td>Shared #GP</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RBX]PAGEINFO.SECS</td>
<td>Shared #GP</td>
</tr>
<tr>
<td>ELDBC/ELBUC</td>
<td>Target [DS:RCX]</td>
<td>Exclusive SGX_EPC_PAGE_CONFLICT_ERROR</td>
</tr>
<tr>
<td></td>
<td>VA [DS:RDX]</td>
<td>Shared SGX_EPC_PAGE_CONFLICT_ERROR</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RBX]PAGEINFO.SECS</td>
<td>Shared SGX_EPC_PAGE_CONFLICT_ERROR</td>
</tr>
</tbody>
</table>

Table 38-30. Additional Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELDB/ELDU</td>
<td>Target [DS:RCX]</td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td></td>
<td>VA [DS:RDX]</td>
<td>vs. EADD, EEXTEND, EINIT</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RBX]PAGEINFO.SECS</td>
<td>vs. ETRACK, ETRACKC</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RBX]PAGEINFO.SECS</td>
<td>Access On Conflict</td>
</tr>
</tbody>
</table>

Operation

Temp Variables in ELDB/ELDU/ELDBC/ELBUC Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SRCPGE</td>
<td>Memory page</td>
<td>4KBytes</td>
<td></td>
</tr>
<tr>
<td>TMP_SECS</td>
<td>Memory page</td>
<td>4KBytes</td>
<td></td>
</tr>
<tr>
<td>TMP_PCMD</td>
<td>PCMD</td>
<td>128 Bytes</td>
<td></td>
</tr>
<tr>
<td>TMP_HEADER</td>
<td>MACHEADER</td>
<td>128 Bytes</td>
<td></td>
</tr>
<tr>
<td>TMP_VER</td>
<td>UINT64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>TMP_MAC</td>
<td>UINT128</td>
<td>128</td>
<td>Page encryption/MAC key.</td>
</tr>
<tr>
<td>TMP_PK</td>
<td>UINT128</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>SCRATCH_PCMD</td>
<td>PCMD</td>
<td>128 Bytes</td>
<td></td>
</tr>
</tbody>
</table>

(* Check PAGEINFO and EPCPAGE alignment *)
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
  THEN #GP(0); FI;
IF (DS:RCX does not resolve within an EPC)
   THEN #PF(DS:RCX); FI;

(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
   THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
   THEN #PF(DS:RDX); FI;

TMP_SRCPGE := DS:RBX.SRCPGE;
TMP_SECS := DS:RBX.SECS;
TMP_PCMD := DS:RBX.PCMD;

(* Check alignment of PAGEINFO (RBX) linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
   THEN #GP(0); FI;

(* Check concurrency of EPC by other Intel SGX instructions *)
IF (other instructions accessing EPC)
   THEN
   IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
      THEN
      IF (<<VMX non-root operation>> AND
          <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>) 
         THEN
         VMCS.Exit_reason := SGX_CONFLICT;
         VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
         VMCS.Exit_qualification.error := 0;
         VMCS.Guest-physical_address :=
         << translation of DS:RCX produced by paging >>;
         VMCS.Guest-linear_address := DS:RCX;
         Deliver VMEXIT;
         ELSE
         #GP(0);
         FI;
      ELSE (* ELDBC/ELDUC *)
      IF (<<VMX non-root operation>> AND
          <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>) 
         THEN
         VMCS.Exit_reason := SGX_CONFLICT;
         VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_ERROR;
         VMCS.Exit_qualification.error := SGX_EPC_PAGE_CONFLICT;
         VMCS.Guest-physical_address :=
         << translation of DS:RCX produced by paging >>;
         VMCS.Guest-linear_address := DS:RCX;
         Deliver VMEXIT;
         ELSE
         RFLAGS.ZF := 1;
         RFLAGS.CF := 0;
         RAX := SGX_EPC_PAGE_CONFLICT;
         GOTO ERROR_EXIT;
         FI;
   ELSE
   #GP(0);
   FI;
INTEL® SGX INSTRUCTION REFERENCES

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF (Other instructions modifying VA slot) THEN
  IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
    THEN #GP(0);
  ELSE (* ELDBC/ELDUC *)
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_EPC_PAGE_CONFLICT;
    GOTO ERROR_EXIT;
  FI;
FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1) THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~0FFFH).PT ≠ PT_VA) )
  THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0] := DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0] := 0;

TMP_HEADER.SECINFO := SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD := SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR := DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
  (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
  (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
  THEN
    IF ( DS:TMP_SECS is not 4KByte aligned)
      THEN #GP(0) FI;
    IF (DS:TMP_SECS does not resolve within an EPC)
      THEN #PF(DS:TMP_SECS) FI;
    IF (Another instruction is currently modifying the SECS) THEN
      IF ((EAX==07h) OR (EAX==08h)) (* ELDB/ELDU *)
        THEN #GP(0);
      ELSE (* ELDBC/ELDUC *)
        RFLAGS.ZF := 1;
        RFLAGS.CF := 0;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO ERROR_EXIT;
      FI;
    FI;
  FI;
  TMP_HEADER.EID := DS:TMP_SECS.EID;
ELSE
(* TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS or PT_VA which do not have a parent SECS, and hence no EID binding *)
TMP_HEADER.EID := 0;
IF (DS:TMP_SECS ≠ 0)
THEN #GP(0) FI;

(* Copy 4KBytes SRCPGE to secure location *)
DS:RCX[32767:0] := DS:TMP_SRCPGE[32767:0];
TMP_VER := DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, (plain text, MAC *)
(* Parameters for AES_GCM_DEC (Key, Counter, ..) *)
(DS:RCX, TMP_MAC) := AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
THEN
   RFLAGS.ZF := 1;
   RAX := SGX_MACCOMPARE_FAIL;
   GOTO ERROR_EXIT;
FI;

(* Clear VA Slot *)
DS:RDX := 0

(* Commit EPCM changes *)
EPCM(DS:RCX).PT := TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX := TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING := TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED := TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR := TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS := TMP_HEADER.LINADDR;

IF ( ((EAX = 07H) or (EAX = 12H)) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA))
THEN
   EPCM(DS:RCX).BLOCKED := 1;
ELSE
   EPCM(DS:RCX).BLOCKED := 0;
FI;

IF (TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS)
   \<< store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
FI;

EPCM(DS:RCX).VALID := 1;

RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;

**Flags Affected**
Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.
Protected Mode Exceptions

#GP(0)  If a memory operand effective address is outside the DS segment limit.
        If a memory operand is not properly aligned.
        If the instruction’s EPC resource is in use by others.
        If the instruction fails to verify MAC.
        If the version-array slot is in use.
        If the parameters fail consistency checks.

#PF(error code)  If a page fault occurs in accessing memory operands.
        If a memory operand expected to be in EPC does not resolve to an EPC page.
        If one of the EPC memory operands has incorrect page type.
        If the destination EPC page is already valid.

64-Bit Mode Exceptions

#GP(0)  If a memory operand is non-canonical form.
        If a memory operand is not properly aligned.
        If the instruction’s EPC resource is in use by others.
        If the instruction fails to verify MAC.
        If the version-array slot is in use.
        If the parameters fail consistency checks.

#PF(error code)  If a page fault occurs in accessing memory operands.
        If a memory operand expected to be in EPC does not resolve to an EPC page.
        If one of the EPC memory operands has incorrect page type.
        If the destination EPC page is already valid.
**EMODPR—Restrict the Permissions of an EPC Page**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 0EH ENCLS[EMODPR]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX2</td>
<td>This leaf function restricts the access rights associated with an EPC page in an initialized enclave.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EMODPR (In)</td>
<td>Return Error Code (Out)</td>
<td>Address of a SECINFO (In)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address of the destination EPC page (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

**EMODPR Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>SECINFO</th>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read/Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EMODPR Faulting Conditions**

- The operands are not properly aligned.
- If unsupported security attributes are set.
- The Enclave is not initialized.
- SECS is locked by another thread.
- The EPC page is locked by another thread.
- RCX does not contain an effective address of an EPC page in the running enclave.
- The EPC page is not valid.

The error codes are:

**Table 38-31. EMODPR Return Value in RAX**

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EMODPR successful.</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_MODIFIABLE</td>
<td>The EPC page cannot be modified because it is in the PENDING or MODIFIED state.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.</td>
</tr>
</tbody>
</table>

**Concurrency Restrictions**

**Table 38-32. Base Concurrency Restrictions of EMODPR**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMODPR</td>
<td>Target [DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EMODPR Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Physical address of SECS to which EPC operand belongs.</td>
</tr>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:RBX.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); Fi;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); Fi;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fi;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags *)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or (SCRATCH_SECINFO.FLAGS.R is 0 and SCRATCH_SECINFO.FLAGS.W is not 0) )
    THEN #GP(0); Fi;

(* Check concurrency with SGX1 or SGX2 instructions on the EPC page *)
IF (SGX1 or other SGX2 instructions accessing EPC page)
    THEN #GP(0); Fi;

IF (EPCM(DS:RCX).VALID is 0 )
    THEN #PF(DS:RCX); Fi;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
    Fi;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_MODIFIABLE;
GOTO DONE;
Fi;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR := 1;

(* Update EPCM permissions *)

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected
Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.

64-Bit Mode Exceptions
#GP(0) If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
EMODT—Change the Type of an EPC Page

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 0FH ENCLS[EMODT]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX2</td>
<td>This leaf function changes the type of an existing EPC page.</td>
</tr>
</tbody>
</table>

Description
This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

EMODT Memory Parameter Semantics

<table>
<thead>
<tr>
<th>SECINFO</th>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read/Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

EMODT Faulting Conditions

| The operands are not properly aligned. | If unsupported security attributes are set. |
| The Enclave is not initialized. | SECS is locked by another thread. |
| The EPC page is locked by another thread. | RCX does not contain an effective address of an EPC page in the running enclave. |
| The EPC page is not valid. | |

The error codes are:

Table 38-34. EMODT Return Value in RAX

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EMODT successful.</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_MODIFIABLE</td>
<td>The EPC page cannot be modified because it is in the PENDING or MODIFIED state.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB.</td>
</tr>
</tbody>
</table>

Concurrency Restrictions

Table 38-35. Base Concurrency Restrictions of EMODT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMODT</td>
<td>Target [DS:RCX]</td>
<td>Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exclusive</td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EMODT Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Physical address of SECS to which EPC operand belongs.</td>
</tr>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:RBX.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 64Byte Aligned)
   THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
   THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
   THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (((SCRATCH_SECINFO reserved fields are not zero) or

((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or SCRATCH_SECINFO.FLAGS.PT is PT_TRIM))
   THEN #GP(0); FI; (* Check concurrency with SGX1 instructions on the EPC page *)
IF (other SGX1 instructions accessing EPC page)
   THEN
     RFLAGS.ZF := 1;
     RAX := SGX_EPC_PAGE_CONFLICT;
     GOTO DONE;

IF (EPCM(DS:RCX).VALID is 0)
   THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
   THEN
     RFLAGS.ZF := 1;
     RAX := SGX_EPC_PAGE_CONFLICT;
     GOTO DONE;
FI;

IF (((EPCM(DS:RCX).PT is PT_REG or ((EPCM(DS:RCX).PT is PT_TCS or PT_SS_FIRST or PT_SS_REST) and SCRATCH_SECINFO.FLAGS.PT is PT_TRIM)))
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
    FI;

TMP_SECS := GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Update EPCM fields *)
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).MODIFIED := 1;
EPCM(DS:RCX).R := 0;
EPCM(DS:RCX).W := 0;
EPCM(DS:RCX).X := 0;
EPCM(DS:RCX).PT := SCRATCH_SECINFO.FLAGS.PT;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected
Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
    If a memory operand is not properly aligned.
    If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
    If a memory operand is not an EPC page.

64-Bit Mode Exceptions
#GP(0) If a memory operand is non-canonical form.
    If a memory operand is not properly aligned.
    If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
    If a memory operand is not an EPC page.
EPA—Add Version Array

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 0AH ENCLS[EPA]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function adds a Version Array to the EPC.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EPA (In)</td>
<td>PT_VA (In, Constant)</td>
<td>Effective address of the EPC page (In)</td>
</tr>
</tbody>
</table>

Description

This leaf function creates an empty version array in the EPC page whose logical address is given by DS:RCX, and sets up EPCM attributes for that page. At the time of execution of this instruction, the register RBX must be set to PT_VA.

The table below provides additional information on the memory parameter of EPA leaf function.

EPA Memory Parameter Semantics

<table>
<thead>
<tr>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

Concurrency Restrictions

Table 38-37. Base Concurrency Restrictions of EPA

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPA</td>
<td>VA [DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>EPC_PAGE_CONFLICT_EXCEPTION</td>
<td></td>
</tr>
</tbody>
</table>

Table 38-38. Additional Concurrency Restrictions of EPA

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
<th>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
<td>On Conflict</td>
<td>Access</td>
<td>On Conflict</td>
</tr>
<tr>
<td></td>
<td>VA [DS:RCX]</td>
<td>Concurrent</td>
<td>L</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>

Operation

IF (RBX ≠ PT_VA or DS:RCX is not 4KByte Aligned)
   THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
   THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions accessing the page)
   THEN
      IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
THEN
  VMCS.Exit_reason := SGX_CONFLICT;
  VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
  VMCS.Exit_qualification.error := 0;
  VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
  VMCS.Guest-linear_address := DS:RCX;
  Deliver VMEXIT;
ELSE
  #GP(0);
FI;

(* Check EPC page must be empty *)
IF (EPCM(DS:RCX).VALID ≠ 0)
  THEN #PF(DS:RCX); FI;

(* Clears EPC page *)
DS:RCX[32767:0] := 0;
EPCM(DS:RCX).PT := PT_VA;
EPCM(DS:RCX).ENCLAVEADDRESS := 0;
EPCM(DS:RCX).BLOCKED := 0;
EPCM(DS:RCX).PENDING := 0;
EPCM(DS:RCX).MODIFIED := 0;
EPCM(DS:RCX).PR := 0;
EPCM(DS:RCX).RWX := 0;
EPCM(DS:RCX).VALID := 1;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit. If a memory operand is not properly aligned. If another Intel SGX instruction is accessing the EPC page. If RBX is not set to PT_VA.

#PF(error code) If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If the EPC page is valid.

64-Bit Mode Exceptions
#GP(0) If a memory operand is non-canonical form. If a memory operand is not properly aligned. If another Intel SGX instruction is accessing the EPC page. If RBX is not set to PT_VA.

#PF(error code) If a page fault occurs in accessing memory operands. If a memory operand is not an EPC page. If the EPC page is valid.
**ERDINFO—Read Type and Status Information About an EPC Page**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ERDINFO (In)</td>
<td>Return error code (Out)</td>
<td>Address of a RDINFO structure (In)</td>
</tr>
</tbody>
</table>

**Description**

This instruction reads type and status information about an EPC page and returns it in a RDINFO structure. The STATUS field of the structure describes the status of the page and determines the validity of the remaining fields. The FLAGS field returns the EPCM permissions of the page; the page type; and the BLOCKED, PENDING, MODIFIED, and PR status of the page. For enclave pages, the ENCLAVECONTEXT field of the structure returns the value of SECS.ENCLAVECONTEXT. For non-enclave pages (e.g., VA) ENCLAVECONTEXT returns 0.

For invalid or non-EPC pages, the instruction returns an information code indicating the page's status, in addition to populating the STATUS field.

ERDINFO returns an error code if the destination EPC page is being modified by a concurrent SGX instruction.

RBX contains the effective address of a RDINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of ERDINFO leaf function.

**ERDINFO Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>RDINFO</th>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access permitted by Non Enclave</td>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**ERDINFO Faulting Conditions**

- A memory operand effective address is outside the DS segment limit (32b mode).
- A memory operand is not properly aligned.
- DS segment is unusable (32b mode).
- A page fault occurs in accessing memory operands.
- A memory address is in a non-canonical form (64b mode).

The error codes are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td>ERDINFO successful.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td>Failure due to concurrent operation of another SGX instruction.</td>
</tr>
<tr>
<td>SGX_PG_INVLD</td>
<td></td>
<td>Target page is not a valid EPC page.</td>
</tr>
<tr>
<td>SGX_PG_NONEPC</td>
<td></td>
<td>Page is not an EPC page.</td>
</tr>
</tbody>
</table>
Concurrency Restrictions

**Table 38-40. Base Concurrency Restrictions of ERDINFO**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
</table>

**Table 38-41. Additional Concurrency Restrictions of ERDINFO**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERDINFO</td>
<td>Target [DS:RCX]</td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT: Access; On Conflict: Concurrent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vs. EADD, EEXTEND, EINIT: Access; On Conflict: Concurrent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vs. ETRACK, ETRACKC: Access; On Conflict: Concurrent</td>
</tr>
</tbody>
</table>

Operation

**Temp Variables in ERDINFO Operational Flow**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Physical Address</td>
<td>64</td>
<td>Physical address of the SECS of the page being modified.</td>
</tr>
<tr>
<td>TMP_RDINFO</td>
<td>Linear Address</td>
<td>64</td>
<td>Address of the RDINFO structure.</td>
</tr>
</tbody>
</table>

(* check alignment of RDINFO structure (RBX) *)
IF (DS:RBX is not 32Byte Aligned) THEN
    #GP(0); Fl;

(* check alignment of the EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); Fl;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within EPC) THEN
    RFLAGS.CF := 1;
    RFLAGS.ZF := 0;
    RAX := SGX_PG_NONEPC;
    goto DONE;
    Fl;

(* Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF = 1;
    RFLAGS.CF = 0;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
    Fl;

(* check page validity *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    RFLAGS.CF = 1;
RFLAGS.ZF = 0;
RAX = SGX_PG_INVLD;
goto DONE;
FI;

(* clear the fields of the RDINFO structure *)
TMP_RDINFO := DS:RBX;
TMP_RDINFO.STATUS := 0;
TMP_RDINFO.FLAGS := 0;
TMP_RDINFO.ENCLAVECONTEXT := 0;

(* store page info in RDINFO structure *)
TMP_RDINFO.FLAGS.RWX := EPCM(DS:RCX).RwX;
TMP_RDINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_RDINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_RDINFO.FLAGS.PR := EPCM(DS:RCX).PR;
TMP_RDINFO.FLAGS.PAGE_TYPE := EPCM(DS:RCX).PAGE_TYPE;
TMP_RDINFO.FLAGS.BLOCKED := EPMC(DS:RCX).BLOCKED;

(* read SECS.ENCLAVECONTEXT for enclave child pages *)
IF ((EPCM(DS:RCX).PAGE_TYPE = PT_REG) or
(EPCM(DS:RCX).PAGE_TYPE = PT_TCS) or
(EPCM(DS:RCX).PAGE_TYPE = PT_TRIM) or
(EPCM(DS:RCX).PAGE_TYPE = PT_SS_FIRST) or
(EPCM(DS:RCX).PAGE_TYPE = PT_SS_REST)
) THEN
TMP_SECS := Address of SECS for (DS:RCX);
TMP_RDINFO.ENCLAVECONTEXT := SECS(TMP_SECS).ENCLAVECONTEXT;
FI;

(* populate enclave information for SECS pages *)
IF (EPCM(DS:RCX).PAGE_TYPE = PT_SECS) THEN
IF ((VMX non-root mode) and
(ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1))
) THEN
TMP_RDINFO.STATUS.CHILDPRESENT :=
((SECS(DS:RCX).CHLDCNT ≠ 0) or
 SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
ELSE
TMP_RDINFO.STATUS.CHILDPRESENT := (SECS(DS:RCX).CHLDCNT ≠ 0);
TMP_RDINFO.STATUS.VIRTCHILDPRESENT :=
(SEC(SD:RCX).VIRTCHILDCNT ≠ 0);
TMP_RDINFO.ENCLAVECONTEXT := SECS(DS:RCX).ENCLAVECONTEXT;
FI;
FI;

RAX := 0;
RFLAGS.ZF := 0;
RFLAGS.CF := 0;

DONE:
(* clear flags *)
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

Flags Affected
ZF is set if ERDINFO fails due to concurrent operation with another SGX instruction; otherwise cleared.
CF is set if page is not a valid EPC page or not an EPC page; otherwise cleared.
PF, AF, OF, and SF are cleared.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
  If DS segment is unusable.
  If a memory operand is not properly aligned.
#PF(error code) If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
  If a memory operand is not properly aligned.
#PF(error code) If a page fault occurs in accessing memory operands.
**EREMOVE—Remove a page from the EPC**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 03H ENCLS[EREMOVE]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function removes a page from the EPC.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EREMOVE (In)</td>
<td>Return error code (Out)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function causes an EPC page to be un-associated with its SECS and be marked as unused. This instruction leaf can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if the operand is not properly aligned or does not refer to an EPC page or the page is in use by another thread, or other threads are running in the enclave to which the page belongs. In addition the instruction fails if the operand refers to an SECS with associations.

**EREMOVE Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EREMOVE Faulting Conditions**

<table>
<thead>
<tr>
<th>The memory operand is not properly aligned.</th>
<th>The memory operand does not resolve in an EPC page.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refers to an invalid SECS.</td>
<td>Refers to an EPC page that is locked by another thread.</td>
</tr>
<tr>
<td>Another Intel SGX instruction is accessing the EPC page.</td>
<td>RCX does not contain an effective address of an EPC page.</td>
</tr>
<tr>
<td>the EPC page refers to an SECS with associations.</td>
<td></td>
</tr>
</tbody>
</table>

The error codes are:

**Table 38-42. EEREMOVE Return Value in RAX**

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EREMOVE successful.</td>
</tr>
<tr>
<td>SGX_CHILD_PRESENT</td>
<td>If the SECS still have enclave pages loaded into EPC.</td>
</tr>
<tr>
<td>SGX_ENCLAVE_ACT</td>
<td>If there are still logical processors executing inside the enclave.</td>
</tr>
</tbody>
</table>
Concurrency Restrictions

Table 38-43. Base Concurrency Restrictions of EREMOVE

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>EREMOVE</td>
<td>Target [DS:RCX]</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>

Table 38-44. Additional Concurrency Restrictions of EREMOVE

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>EREMOVE</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Operation

Temp Variables in EREMOVE Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Effective address of the SECS destination page.</td>
</tr>
</tbody>
</table>

IF (DS:RCX is not 4KByte Aligned)
THEN #GP(0); FI;

IF (DS:RCX does not resolve to an EPC page)
THEN #PF(DS:RCX); FI;

TMP_SECS := Get_SECS_ADDRESS();

(* Check the EPC page for concurrency *)
IF (EPC page being referenced by another Intel SGX instruction)
THEN 
   IF <<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
      THEN
         VMCS.Exit_reason := SGX_CONFLICT;
         VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
         VMCS.Exit_qualification.error := 0;
         VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
         VMCS.Guest-linear_address := DS:RCX;
         Deliver VMEXIT;
      ELSE
         #GP(0);
      FI;
   FI;

(* if DS:RCX is already unused, nothing to do*)
IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT = PT_TRIM AND EPCM(DS:RCX).MODIFIED = 0))
THEN GOTO DONE;
FI;
IF ( (EPCM(DS:RCX).PT = PT_VA) OR 
((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
THEN
EPCM(DS:RCX).VALID := 0;
GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
THEN
IF (DS:RCX has an EPC page associated with it)
THEN
RFLAGS.ZF := 1;
RAX := SGX_CHILD_PRESENT;
GOTO ERROR_EXIT;
FI;
(* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
IF (<<in VMX non-root operation>> AND
<<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
(SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
THEN
RFLAGS.ZF := 1;
RAX := SGX_CHILD_PRESENT
GOTO ERROR_EXIT
FI;
EPCM(DS:RCX).VALID := 0;
GOTO DONE;
FI;

IF (Other threads active using SECS)
THEN
RFLAGS.ZF := 1;
RAX := SGX_ENCLAVE_ACT;
GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or 
(EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
THEN
EPCM(DS:RCX).VALID := 0;
GOTO DONE;
FI;

DONE:
RAX := 0;
RFLAGS.ZF := 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected
Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.
Protected Mode Exceptions

#GP(0)  If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If another Intel SGX instruction is accessing the page.

#PF(error code)  If a page fault occurs in accessing memory operands.
If the memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)  If the memory operand is non-canonical form.
If a memory operand is not properly aligned.
If another Intel SGX instruction is accessing the page.

#PF(error code)  If a page fault occurs in accessing memory operands.
If the memory operand is not an EPC page.
**ETRACK—Activates EBLOCK Checks**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 0CH, ENCLS[ETRACK]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function activates EBLOCK checks.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ETRACK (In)</td>
<td>Return error code (Out)</td>
</tr>
<tr>
<td></td>
<td>Pointer to the SECS of the EPC page (In)</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0. The content of RCX is an effective address of an EPC page. The table below provides additional information on the memory parameter of ETRACK leaf function.

### ETRACK Memory Parameter Semantics

<table>
<thead>
<tr>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The error codes are:

#### Table 38-45. ETRACK Return Value in RAX

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>ETRACK successful.</td>
</tr>
<tr>
<td>SGX_PREV_TRK_INCMPL</td>
<td>All processors did not complete the previous shoot-down sequence.</td>
</tr>
</tbody>
</table>

#### Concurrency Restrictions

#### Table 38-46. Base Concurrency Restrictions of ETRACK

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETRACK</td>
<td>SECS [DS:RCX]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 38-47. Additional Concurrency Restrictions of ETRACK

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
<td>vs. EADD, EEEXTEND, EINIT</td>
</tr>
<tr>
<td>ETRACK</td>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
Operation

IF (DS:RCX is not 4KByte Aligned)
   THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
   THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS)
   THEN
      IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
         THEN
            VMCS.Exit_reason := SGX_CONFLICT;
            VMCS.Exit_qualification.code := TRACKING_RESOURCE_CONFLICT;
            VMCS.Exit_qualification.error := 0;
            VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
            VMCS.Guest-linear_address := 0;
            Deliver VMEXIT;
         ELSE
            #GP(0);
         FI;
      FI;

IF (EPCM(DS:RCX). VALID = 0)
   THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PT ≠ PT_SECS)
   THEN #PF(DS:RCX); FI;

(* All processors must have completed the previous tracking cycle*)
IF ( (DS:RCX).TRACKING ≠ 0 )
   THEN
      IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
         THEN
            VMCS.Exit_reason := SGX_CONFLICT;
            VMCS.Exit_qualification.code := TRACKING_REFERENCE_CONFLICT;
            VMCS.Exit_qualification.error := 0;
            VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
            VMCS.Guest-linear_address := 0;
            Deliver VMEXIT;
         ELSE
            RFLAGS.ZF := 1;
            RAX := SGX_PREV_TRK_INCMPL;
            GOTO DONE;
         FI;
      FI;
   ELSE
      RFLAGS.ZF := 0;
      RAX := 0;
   FI;

DONE:
RFLAGS.CF,PF,AF,OF,SE := 0;

Flags Affected
Sets ZF if SECS is in use or invalid, otherwise cleared. Clears CF, PF, AF, OF, SF.
Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If another thread is concurrently using the tracking facility on this SECS.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0) If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If the specified EPC resource is in use.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
ETRACKC—Activates EBLOCK Checks

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ETRACK (In)</td>
<td>Return error code (Out)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Address of the destination EPC page (In, EA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Address of the SECS page (In, EA)</td>
</tr>
</tbody>
</table>

**Description**

The ETRACKC instruction is thread safe variant of ETRACK leaf and can be executed concurrently with other CPU threads operating on the same SECS.

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

### ETRACKC Memory Parameter Semantics

<table>
<thead>
<tr>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access permitted by Enclave</td>
</tr>
</tbody>
</table>

The error codes are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td>ETRACKC successful.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>7</td>
<td>Failure due to concurrent operation of another SGX instruction.</td>
</tr>
<tr>
<td>SGX_PG_INVLD</td>
<td>6</td>
<td>Target page is not a VALID EPC page.</td>
</tr>
<tr>
<td>SGX_PREV_TRK_INCMPL</td>
<td>17</td>
<td>All processors did not complete the previous tracking sequence.</td>
</tr>
<tr>
<td>SGX_TRACK_NOT_REQUIRED</td>
<td>27</td>
<td>Target page type does not require tracking.</td>
</tr>
</tbody>
</table>

**Concurrency Restrictions**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETRACKC</td>
<td>Target [DS:RCX]</td>
<td>Shared: SGX_EPC_PAGE_CONFLICT</td>
</tr>
<tr>
<td>SECS implicit</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>

Table 38-48. ETRACKC Return Value in RAX

Table 38-49. Base Concurrency Restrictions of ETRACKC
### Operation

Temp Variables in ETRACKC Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Physical Address</td>
<td>64</td>
<td>Physical address of the SECS of the page being modified.</td>
</tr>
</tbody>
</table>

(* check alignment of EPCPAGE (RCX) *)

IF (DS:RCX is not 4KByte Aligned) THEN
#GP(0); Fi;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)

IF (DS:RCX does not resolve within an EPC) THEN
#PF(DS:RCX, PFEC.SGX); Fi;

(* Check the EPC page for concurrency *)

IF (EPC page is being modified) THEN
  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_EPC_PAGE_CONFLICT;
  goto DONE_POST_LOCK_RELEASE;
Fi;

(* check to make sure the page is valid *)

IF (EPCM(DS:RCX).VALID = 0) THEN
  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_PG_INVLD;
  GOTO DONE;
Fi;

(* find out the target SECS page *)

IF (EPCM(DS:RCX).PT is PT_REG or PT_TCS or PT_TRIM or PT_SS_FIRST or PT_SS_REST) THEN
  TMP_SECS := Obtain SECS through EPCM(DS:RCX).ENCLAVESECS;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS) THEN
  TMP_SECS := Obtain SECS through (DS:RCX);
ELSE
  RFLAGS.ZF := 0;
  RFLAGS.CF := 1;
  RAX := SGX_TRACK_NOT_REQUIRED;
  GOTO DONE;
Fi;

### Table 38-50. Additional Concurrency Restrictions of ETRACKC

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>ETRACKC</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
</tr>
<tr>
<td>SECS implicit</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Vol. 3D  38-81
INTEL® SGX INSTRUCTION REFERENCES

(* Check concurrency with other Intel SGX instructions *) IF (Other Intel SGX instructions using tracking facility on this SECS) THEN
  IF ((VMX non-root mode) and (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
    VMCS.Exit reason := SGX_CONFLICT;
    VMCS.Exit qualification.code := TRACKING_RESOURCE_CONFLICT;
    VMCS.Exit qualification.error := 0;
    VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
    VMCS.Guest-linear_address := 0;
    Deliver VMEXIT;
  FI;
  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_EPC_PAGE_CONFLICT;
  GOTO DONE;
FI;

(* All processors must have completed the previous tracking cycle*) IF ( (TMP_SECS).TRACKING ≠ 0) ) THEN
  IF ((VMX non-root mode) and (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
    VMCS.Exit reason := SGX_CONFLICT;
    VMCS.Exit qualification.code := TRACKING_REFERENCE_CONFLICT;
    VMCS.Exit qualification.error := 0;
    VMCS.Guest-physical_address := SECS(TMP_SECS).ENCLAVECONTEXT;
    VMCS.Guest-linear_address := 0;
    Deliver VMEXIT;
  FI;
  RFLAGS.ZF := 1;
  RFLAGS.CF := 0;
  RAX := SGX_PREV_TRK_INCMPL;
  GOTO DONE;
FI;

DONE:
(* clear flags *)
RFLAGS.PF,AF,OF, SF := 0;

Flags Affected
ZF is set if ETRACKC fails due to concurrent operations with another SGX instructions or target page is an invalid EPC page or tracking is not completed on SECS page; otherwise cleared.
CF is set if target page is not of a type that requires tracking; otherwise cleared.
PF, AF, OF, and SF are cleared.
Protected Mode Exceptions

#GP(0)   If the memory operand violates access-control policies of DS segment.
         If DS segment is unusable.
         If the memory operand is not properly aligned.

#PF(error code)   If the memory operand expected to be in EPC does not resolve to an EPC page.
                  If a page fault occurs in access memory operand.

64-Bit Mode Exceptions

#GP(0)   If a memory address is in a non-canonical form.
         If a memory operand is not properly aligned.

#PF(error code)   If the memory operand expected to be in EPC does not resolve to an EPC page.
                  If a page fault occurs in access memory operand.
**EWB—Invalidate an EPC Page and Write out to Main Memory**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 0BH ENCLS[EWB]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function invalidates an EPC page and writes it out to main memory.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EWB (In)</td>
<td>Error code (Out)</td>
<td>Address of a PAGEINFO (In)</td>
<td>Address of the EPC page (In)</td>
</tr>
</tbody>
</table>

### Description

This leaf function copies a page from the EPC to regular main memory. As part of the copying process, the page is cryptographically protected. This instruction can only be executed when current privilege level is 0.

The table below provides additional information on the memory parameter of EPA leaf function.

### EWB Memory Parameter Semantics

<table>
<thead>
<tr>
<th>PAGEINFO</th>
<th>PAGEINFO.SRCPGE</th>
<th>PAGEINFO.PCMD</th>
<th>EPCPAGE</th>
<th>VASLOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-EPC R/W access</td>
<td>Non-EPC R/W access</td>
<td>Non-EPC R/W access</td>
<td>EPC R/W access</td>
<td>EPC R/W access</td>
</tr>
</tbody>
</table>

The error codes are:

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EWb successful.</td>
</tr>
<tr>
<td>SGX_PAGE_NOT_BLOCKED</td>
<td>If page is not marked as blocked.</td>
</tr>
<tr>
<td>SGX_NOT_TRACKED</td>
<td>If EWb is racing with ETACK instruction.</td>
</tr>
<tr>
<td>SGX_VA_SLOT_OCCUPIED</td>
<td>Version array slot contained valid entry.</td>
</tr>
<tr>
<td>SGX_CHILD_PRESENT</td>
<td>Child page present while attempting to page out enclave.</td>
</tr>
</tbody>
</table>

### Concurrency Restrictions

#### Table 38-52. Base Concurrency Restrictions of EWB

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EWb</td>
<td>Source [DS:RCX]</td>
<td>Exclusive #GP EPC_PAGE_CONFLICT_EXCEPTION</td>
</tr>
<tr>
<td>VA</td>
<td>[DS:RDX]</td>
<td>Shared #GP</td>
</tr>
</tbody>
</table>

#### Table 38-53. Additional Concurrency Restrictions of EWB

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EWb</td>
<td>Source [DS:RCX]</td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td>VA</td>
<td>[DS:RDX]</td>
<td>vs. EADD, EEXTEND, EINIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vs. ETRACK, ETRACKC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>Access</th>
<th>On Conflict</th>
<th>Access</th>
<th>On Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>EWb</td>
<td>Source [DS:RCX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Exclusive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VA</td>
<td>[DS:RDX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EWB Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SRCPGE</td>
<td>Memory page</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>TMP_PCMD</td>
<td>PCMD</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>TMP_SECS</td>
<td>SECS</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>TMP_BPEPOCH</td>
<td>UINT64</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TMP_BPREFCOUNT</td>
<td>UINT64</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TMP_HEADER</td>
<td>MAC Header</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>TMP_PCMD_ENCLAVEID</td>
<td>UINT64</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TMP_VER</td>
<td>UINT64</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TMP_PK</td>
<td>UINT128</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); Fi;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fi;

IF (DS:RDX is not 8Byte Aligned)
    THEN #GP(0); Fi;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); Fi;

(* EPCPAGE and VASLOT should not resolve to the same EPC page*)
IF (DS:RCX and DS:RDX resolve to the same EPC page)
    THEN #GP(0); Fi;

TMP_SRCPGE := DS:RBX.SRCPGE;
(* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO *)
TMP_PCMD := DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)
    THEN #GP(0); Fi;

IF ( (DS:TMP_PCMD is not 128Byte Aligned) or (DS:TMP_SRCPGE is not 4KByte Aligned) )
    THEN #GP(0); Fi;

(* Check for concurrent Intel SGX instruction access to the page *)
IF (Other Intel SGX instruction is accessing page)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason := SGX_CONFLICT;
                VMCS.Exit_qualification.code := EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error := 0;
                VMCS.Guest-physical_address := << translation of DS:RCX produced by paging >>;
VMCS.Guest-linear_address := DS:RCX;
Deliver VMEXIT;
ELSE
    #GP(0);
FI;
FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;
IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM ) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST ) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        TMP_SECS = Obtain SECS through EPCM(DS:RCX)
        (* Check that EBLOCK has occurred correctly *)
        IF (EBLOCK is not correct)
            THEN #GP(0); FI;
    FI;

RFLAGS.ZF,CF,PF,AF,OF,SF := 0;
RAX := 0;

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER) - 1 : 0] := 0;

(* Perform page-type-specific checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM ) or
    (EPCM(DS:RCX).PT is PT_SS_FIRST ) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        (* check to see if the page is evictable *)
        IF (EPCM(DS:RCX).BLOCKED = 0)
            THEN
                RAX := SGX_PAGE_NOT_BLOCKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
            FI;
        (* Check if tracking done correctly *)
        IF (Tracking not correct)
            THEN
                RAX := SGX_NOT_TRACKED;
                RFLAGS.ZF := 1;
                GOTO ERROR_EXIT;
            FI;
    FI;

(* Obtain EID to establish cryptographic binding between the paged-out page and the enclave *),
TMP_HEADER.EID := TMP_SECS.EID;

(* Obtain EID as an enclave handle for software *)
TMP_PCIE := TMP_SECS.EID;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS)
(*check that there are no child pages inside the enclave *)
IF (DS:RCX has an EPC page associated with it)
THEN
   RAX := SGX_CHILD_PRESENT;
   RFLAGS.ZF := 1;
   GOTO ERROR_EXIT;
Fl:
(* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
IF (<in VMX non-root operation> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
THEN
   RFLAGS.ZF := 1;
   RAX := SGX_CHILD_PRESENT;
   GOTO ERROR_EXIT;
FI;
TMP_HEADER.EID := 0;
(* Obtain EID as an enclave handle for software *)
TMP_PCIE := (DS:RCX).EID;
ELSE IF (EPCM(DS:RCX).PT is PT_VA)
   TMP_HEADER.EID := 0; // Zero is not a special value
   (* No enclave handle for VA pages*)
   TMP_PCIE := 0;
Fl;

TMP_HEADER.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;
TMP_HEADER.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
TMP_HEADER.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
TMP_HEADER.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
TMP_HEADER.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
TMP_HEADER.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;

(* Encrypt the page, DS:RCX could be encrypted in place. AES-GCM produces 2 values, {ciphertext, MAC}. *)
(* AES-GCM input parameters: key, GCM Counter, MAC_HDR, MAC_HDR_SIZE, SRC, SRC_SIZE*)
(DS:TMP_SRCPGE, DS:TMP_PCIE.MAC) := AES_GCM_ENC(CR_BASE_PK), (TMP_VER << 32),
   (DS:HEADER, 128, DS:RCX, 4096);

(* Write the output *)
Zero out DS:TMP_PCIE.SECINFO
DS:TMP_PCIE.SECINFO.FLAGS.PT := EPCM(DS:RCX).PT;
DS:TMP_PCIE.SECINFO.FLAGS.RWX := EPCM(DS:RCX).RWX;
DS:TMP_PCIE.SECINFO.FLAGS.PENDING := EPCM(DS:RCX).PENDING;
DS:TMP_PCIE.SECINFO.FLAGS.MODIFIED := EPCM(DS:RCX).MODIFIED;
DS:TMP_PCIE.SECINFO.FLAGS.PR := EPCM(DS:RCX).PR;
DS:TMP_PCIE.RESERVED := 0;
DS:TMP_PCIE.ENCLAVEID := TMP_PCIE.ENCLAVEID;
DS:RBX.LINADDR := EPCM(DS:RCX).ENCLAVEADDRESS;

(*Check if version array slot was empty *)
INTEL® SGX INSTRUCTION REFERENCES

IF ([DS.RDX])
   THEN
      RAX := SGX_VA_SLOT_OCCUPIED
      RFLAGS.CF := 1;
   FI;

(* Write version to Version Array slot *)
[DS.RDX] := TMP_VER;

(* Free up EPCM Entry *)
EPCM.(DS:RCX).VALID := 0;
ERROR_EXIT:

Flags Affected
ZF is set if page is not blocked, not tracked, or a child is present. Otherwise cleared.
CF is set if VA slot is previously occupied, Otherwise cleared.

Protected Mode Exceptions
#GP(0)  If a memory operand effective address is outside the DS segment limit.
   If a memory operand is not properly aligned.
   If the EPC page and VASLOT resolve to the same EPC page.
   If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
   If the tracking resource is in use.
   If the EPC page or the version array page is invalid.
   If the parameters fail consistency checks.

#PF(error code)  If a page fault occurs in accessing memory operands.
   If a memory operand is not an EPC page.
   If one of the EPC memory operands has incorrect page type.

64-Bit Mode Exceptions
#GP(0)  If a memory operand is non-canonical form.
   If a memory operand is not properly aligned.
   If the EPC page and VASLOT resolve to the same EPC page.
   If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
   If the tracking resource is in use.
   If the EPC page or the version array page in invalid.
   If the parameters fail consistency checks.

#PF(error code)  If a page fault occurs in accessing memory operands.
   If a memory operand is not an EPC page.
   If one of the EPC memory operands has incorrect page type.
38.4 INTEL® SGX USER LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.
EACCEPT—Accept Changes to an EPC Page

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 05H ENCLU[EACCEPT]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX2</td>
<td>This leaf function accepts changes made by system software to an EPC page in the running enclave.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EACCEPT (In)</td>
<td>Return Error Code (Out)</td>
<td>Address of a SECINFO (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

**EACCEPT Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>SECINFO</th>
<th>EPCPAGE (Destination)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EACCEPT Faulting Conditions**

- The operands are not properly aligned.  
- RBX does not contain an effective address in an EPC page in the running enclave.
- The EPC page is locked by another thread.  
- RCX does not contain an effective address of an EPC page in the running enclave.
- The EPC page is not valid.  
- Page type is PT_REG and MODIFIED bit is 0.
- SECINFO contains an invalid request.  
- Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.
- If security attributes of the SECINFO page make the page inaccessible.

The error codes are:

**Table 38-54. EACCEPT Return Value in RAX**

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EACCEPT successful.</td>
</tr>
<tr>
<td>SGX_PAGE_ATTRIBUTES_MISMATCH</td>
<td>The attributes of the target EPC page do not match the expected values.</td>
</tr>
<tr>
<td>SGX_NOT_TRACKED</td>
<td>The OS did not complete an ETRACK on the target page.</td>
</tr>
</tbody>
</table>
Concurrency Restrictions

Table 38-55. Base Concurrency Restrictions of EACCEPT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>EACCEPT</td>
<td>Target [DS:RCX]</td>
<td>Shared</td>
</tr>
<tr>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>

Table 38-56. Additional Concurrency Restrictions of EACCEPT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
</table>
|      |                | vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT
|      |                | vs. EADD, EEEXTEND, EINIT
|      |                | vs. ETRACK, ETRACKC
|      |                | Access | On Conflict | Access | On Conflict | Access | On Conflict |
| EACCEPT | Target [DS:RCX] | Exclusive | #GP | Concurrent | Concurrent |
| SECINFO [DS:RBX] | Concurrent | Concurrent | Concurrent |

Operation

Temp Variables in EACCEPT Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Physical address of SECS to which EPC operands belongs.</td>
</tr>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:RBX.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 64Byte Aligned)
THEN #GP(0); Fi;

IF (DS:RBX is not within CR_ELRANGE)
THEN #GP(0); Fi;

IF (DS:RBX does not resolve within an EPC)
THEN #PF(DS:RBX); Fi;

IF ([EPCM(DS:RBX &~FFFH).VALID = 0] or [EPCM(DS:RBX &~FFFH).R = 0] or [EPCM(DS:RBX &~FFFH).PENDING ≠ 0] or 
([EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0] or [EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0] or 
([EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)]) )
THEN #PF(DS:RBX); Fi;

(* Copy 64 bytes of contents *)
SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero )
THEN #GP(0); Fi;

IF (DS:RCX is not 4KByte Aligned)
THEN #GP(0); Fi;
IF (DS:RCX is not within CR_ELRANGE) THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC) THEN #PF(DS:RCX); FI;

(* Check that the combination of requested PT, PENDING, and MODIFIED is legal *) IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 0 ) THEN
  IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) and (SCRATCH_SECINFO.FLAGS.PR is 1) or (SCRATCH_SECINFO.FLAGS.PENDING is 1)) and (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) or ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or PT_TRIM) and (SCRATCH_SECINFO.FLAGS.PR is 0) and (SCRATCH_SECINFO.FLAGS.MODIFIED is 0) or (SCRATCH_SECINFO.FLAGS.PENDING MODIFIED is 1) )
    THEN #GP(0); FI ELSE
  IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) AND (SCRATCH_SECINFO.FLAGS.PR is 1) OR (SCRATCH_SECINFO.FLAGS.PENDING is 1)) AND (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) OR ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS OR PT_TRIM) AND (SCRATCH_SECINFO.FLAGS.PENDING is 0) AND (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) AND (SCRATCH_SECINFO.FLAGS.PR is 0)) OR ((SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST or PT_SS_REST) AND (SCRATCH_SECINFO.FLAGS.PENDING is 1) AND (SCRATCH_SECINFO.FLAGS.MODIFIED is 0) AND (SCRATCH_SECINFO.FLAGS.PR is 0)))
    THEN #GP(0); FI; FI;

(* Check security attributes of the destination EPC page *) IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).BLOCKED is not 0) or ((EPCM(DS:RCX).PT is not PT_REG) and (EPCM(DS:RCX).PT is not PT_TCS) and (EPCM(DS:RCX).PT is not PT_TRIM) and (EPCM(DS:RCX).PT is not PT_SS_FIRST) and (EPCM(DS:RCX).PT is not PT_SS_REST)) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
    THEN #PF(DS:RCX); FI;

(* Check the destination EPC page for concurrency *) IF ( EPC page in use )
  THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *) IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
  THEN #PF(DS:RCX); FI;

(* Verify that accept request matches current EPC page settings *) IF ( (EPCM(DS:RCX).ENClarEADRESS ≠ DS:RCX) or (EPCM(DS:RCX).PENDING ≠ SCRATCH_SECINFO.FLAGS.PENDING) or (EPCM(DS:RCX).MODIFIED ≠ SCRATCH_SECINFO.FLAGS.MODIFIED) or (EPCM(DS:RCX).R ≠ SCRATCH_SECINFO.FLAGS.R) or (EPCM(DS:RCX).W ≠ SCRATCH_SECINFO.FLAGS.W) or (EPCM(DS:RCX).X ≠ SCRATCH_SECINFO.FLAGS.X) or (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) )
THEN
   RFLAGS.ZF := 1;
   RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
   GOTO DONE;
FI;
(* Check that all required threads have left enclave *)
IF (Tracking not correct)
   THEN
      RFLAGS.ZF := 1;
      RAX := SGX_NOT_TRACKED;
      GOTO DONE;
FI;

(* Get pointer to the SECS to which the EPC page belongs *)
TMP_SECS = << Obtain physical address of SECS through EPCM(DS:RCX)>>
(* For TCS pages, perform additional checks *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
   THEN
      IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
   (* Check that TCS.FLAGS.DBGOPTIN, TCS stack, and TCS status are correctly initialized *)
   (* check that TCS.PREVSSP is 0 *)
   IF ( ((DS:RCX).FLAGS.DBGOPTIN is not 0) or ((DS:RCX).CSSA ≥ (DS:RCX).NSSA) or ((DS:RCX).AEP is not 0) or ((DS:RCX).STATE is not 0)
      or ((CPUID.(EAX=07H, ECX=0H):ECX[CET_SS] = 1) AND ((DS:RCX).PREVSSP != 0)))
      THEN #GP(0); FI;
   (* Check consistency of FS & GS Limit *)
   IF ( (TMP_SECS.ATTRIBUTES.MODE64BIT is 0) and ((DS:RCX.FSLIMIT & FFFH ≠ FFFH) or (DS:RCX.GSLIMIT & FFFH ≠ FFFH)) )
      THEN #GP(0); FI;
   (* Clear PENDING/MODIFIED flags to mark accept operation complete *)
   EPCM(DS:RCX).PENDING := 0;
   EPCM(DS:RCX).MODIFIED := 0;
   EPCM(DS:RCX).PR := 0;

(* Clear EAX and ZF to indicate successful completion *)
RFLAGS.ZF := 0;
RAX := 0;
DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;

**Flags Affected**
Sets ZF if page cannot be accepted, otherwise cleared. Clears CF, PF, AF, OF, SF
Protected Mode Exceptions

#GP(0) If executed outside an enclave.
If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0) If executed outside an enclave.
If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
If EPC page has incorrect page type or security attributes.
**EACCEPTCOPY—Initialize a Pending Page**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 07H ENCLU[EACCEPTCOPY]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX2</td>
<td>This leaf function initializes a dynamically allocated EPC page from another page in the EPC.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EACCEPTCOPY (In)</td>
<td>Return Error Code (Out)</td>
<td>Address of SECINFO (In)</td>
<td>Address of the destination EPC page (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function copies the contents of an existing EPC page into an uninitialized EPC page (created by EAUG). After initialization, the instruction may also modify the access rights associated with the destination EPC page. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX and RDX each contain the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPTCOPY leaf function.

**EACCEPTCOPY Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>SECINFO</th>
<th>EPCPAGE (Destination)</th>
<th>EPCPAGE (Source)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read/Write access permitted by Enclave</td>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EACCEPTCOPY Faulting Conditions**

- The operands are not properly aligned. If security attributes of the SECINFO page make the page inaccessible.
- The EPC page is locked by another thread. If security attributes of the source EPC page make the page inaccessible.
- The EPC page is not valid. RBX does not contain an effective address in an EPC page in the running enclave.
- SECINFO contains an invalid request. RCX/RDX does not contain an effective address of an EPC page in the running enclave.

The error codes are:

**Table 38-57. EACCEPTCOPY Return Value in RAX**

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>EACCEPTCOPY successful.</td>
</tr>
<tr>
<td>SGX_PAGE_ATTRIBUTES_MISMATCH</td>
<td>The attributes of the target EPC page do not match the expected values.</td>
</tr>
</tbody>
</table>
Concurrent Restrictions

Table 38-58. Base Concurrency Restrictions of EACCEPTCOPY

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCEPTCOPY</td>
<td>Target [DS:RCX]</td>
<td>Concurrent</td>
</tr>
<tr>
<td>Source [DS:RDX]</td>
<td>Concurrent</td>
<td></td>
</tr>
<tr>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>

Table 38-59. Additional Concurrency Restrictions of EACCEPTCOPY

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCEPTCOPY</td>
<td>Target [DS:RCX]</td>
<td>Exclusive #GP Concurrent Concurrency</td>
<td>Concurrent</td>
<td></td>
</tr>
<tr>
<td>Source [DS:RDX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td></td>
</tr>
<tr>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>

Operation

Temp Variables in EACCEPTCOPY Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:RBX.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0);FI;

IF ((DS:RCX is not 4KByte Aligned) or (DS:RDX is not 4KByte Aligned))
    THEN #GP(0);FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) or (DS:RDX is not within CR_ELRANGE))
    THEN #GP(0);FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX);FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX);FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX);FI;

IF ((EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or
    (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ DS:RBX))
    THEN #PF(DS:RBX);FI;
(* Copy 64 bytes of contents *)
SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags *)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or (SCRATCH_SECINFO.FLAGS.R=0) AND (SCRATCH_SECINFO.FLAGS.W#0 ) or
(SCRATCH_SECINFO.FLAGS.PT is not PT_REG ) )
THEN #GP(0); FI;

(* Check security attributes of the source EPC page *)
IF ( (EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING # 0) or (EPCM(DS:RDX).MODIFIED # 0) or
(EPCM(DS:RDX).BLOCKED # 0) or (EPCM(DS:RDX).PT # PT_REG) or (EPCM(DS:RDX).ENCLAVESECS # CR_ACTIVE_SECS) or
(EPCM(DS:RDX).ENCLAVEADDRESS # DS:RDX))
THEN #PF(DS:RDX); FI;

(* Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING # 1) or (EPCM(DS:RCX).MODIFIED # 0) or
(EPCM(DS:RCX).BLOCKED # 0) or (EPCM(DS:RCX).PT # PT_REG) or (EPCM(DS:RCX).ENCLAVESECS # CR_ACTIVE_SECS) )
THEN
RFLAGS.ZF := 1;
RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
GOTO DONE;
FI;

(* Check the destination EPC page for concurrency *)
IF (destination EPC page in use )
THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING # 1) or (EPCM(DS:RCX).MODIFIED # 0) or
(EPCM(DS:RCX).R # 1) or (EPCM(DS:RCX).W # 1) or (EPCM(DS:RCX).X # 0) or
(EPCM(DS:RCX).PT # SCRATCH_SECINFO.FLAGS.PT) or (EPCM(DS:RCX).ENCLAVESECS # CR_ACTIVE_SECS) or
(EPCM(DS:RCX).ENCLAVEADDRESS # DS:RCX))
THEN
RFLAGS.ZF := 1;
RAX := SGX_PAGE_ATTRIBUTES_MISMATCH;
GOTO DONE;
FI;

(* Copy 4KB bytes form the source to destination EPC page *)
DS:RCX[32767:0] := DS:RDX[32767:0];

(* Update EPCM permissions *)
EPCM(DS:RCX).R := SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W := SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X := SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PENDING := 0;
RFLAGS.ZF := 0;
RAX := 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF := 0;
Flags Affected
Sets ZF if page is not modifiable, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0) If executed outside an enclave.
If a memory operand effective address is outside the DS segment limit.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0) If executed outside an enclave.
If a memory operand is non-canonical form.
If a memory operand is not properly aligned.
If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
If a memory operand is not an EPC page.
If EPC page has incorrect page type or security attributes.
EDECCSSA—Decrements TCS.CSSA

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 09H ENCLU[EDECCSSA]</td>
<td>IR</td>
<td>V/V</td>
<td>EDECCSSA</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EDECCSSA (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function changes the current SSA frame by decrementing TCS.CSSA for the current enclave thread. If the enclave has enabled CET shadow stacks or indirect branch tracking, then EDECCSSA also changes the current CET state save frame. This instruction leaf can only be executed inside an enclave.

**EDECCSSA Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>TCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EDECCSSA Faulting Conditions**

| TCS.CSSA is 0. | TCS is not valid or available or locked. |
| The SSA frame is not valid or in use. |

**Concurrency Restrictions**

**Table 38-60. Base Concurrency Restrictions of EDECCSSA**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDECCSSA</td>
<td>TCS [CR_TCS_PA]</td>
<td>Shared</td>
<td>#GP</td>
<td></td>
</tr>
</tbody>
</table>

**Table 38-61. Additional Concurrency Restrictions of EDECCSSA**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Access</td>
<td>On Conflict</td>
<td>Access</td>
</tr>
<tr>
<td>EDECCSSA</td>
<td>TCS [CR_TCS_PA]</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EDECCSSA Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SSA</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of current SSA frame.</td>
</tr>
<tr>
<td>TMP_XSIZE</td>
<td>Integer</td>
<td>64</td>
<td>Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.</td>
</tr>
<tr>
<td>TMP_SSA_PAGE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Pointer used to iterate over the SSA pages in the target frame.</td>
</tr>
<tr>
<td>TMP_GPR</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of the GPR area within the target SSA frame.</td>
</tr>
<tr>
<td>TMP_XSAVE_PAGE_PA_n</td>
<td>Physical Address</td>
<td>32/64</td>
<td>Physical address of the nth page within the target SSA frame.</td>
</tr>
<tr>
<td>TMP_CET_SAVE_AREA</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of the current CET save area.</td>
</tr>
<tr>
<td>TMP_CET_SAVE_PAGE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of the current CET save area page.</td>
</tr>
</tbody>
</table>

IF (CR_TCS_PA.CSSA = 0)
   THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := CR_TCS_PA.OSSA + CR_ACTIVE_SECS.BASEADDR + 4096 * CR_ACTIVE_SECS.SSAFRAMESIZE * (CR_TCS_PA.CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(CR_ACTIVE_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
   (* Check page is read/write accessible *)
   Check that DS:TMP_SSA_PAGE is read/write accessible;
   If a fault occurs, release locks, abort and deliver that fault;
   IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
      THEN #PF(DS:TMP_SSA_PAGE); FI;
   IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
      THEN #PF(DS:TMP_SSA_PAGE); FI;
   IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
      THEN #PF(DS:TMP_SSA_PAGE); FI;
   IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
      THEN #PF(DS:TMP_SSA_PAGE); FI;
   IF ((EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or
       (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
       (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS) or
       (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0))
      THEN #PF(DS:TMP_SSA_PAGE); FI;
   TMP_XSAVE_PAGE_PA_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * CR_ACTIVE_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
   THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or
    (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP.MODE64 = 0)
    THEN
        IF (TMP_GPR + (sizeof(GPRSGX_AREA) -1) is not in DS segment)
            THEN #GP(0); FI;
    FI;
    THEN
        IF ((CR_ACTIVE_SECS.CET_ATTRIBUTES.SH_STK_EN == 1) OR (CR_ACTIVE_SECS.CET_ATTRIBUTES.ENDBR_EN == 1))
            THEN
                (* Compute linear address of what will become new CET state save area and cache its PA *)
                TMP_CET_SAVE_PAGE := TMP_CET_SAVE_AREA & ~0xFFF;
                Check the TMP_CET_SAVE_PAGE page is read/write accessible
                If fault occurs release locks, abort and deliver fault

                (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
                IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
                    (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS))
                    THEN #PF(DS:TMP_CET_SAVE_PAGE); FI;
                FI;

        (* At this point, the instruction is guaranteed to complete *)
        CR_TCS_PA.CSSA := CR_TCS_PA.CSSA - 1;
        CR_GPR_PA := Physical_Address(DS:TMP_GPR);

        FOR EACH TMP_XSAVE_PAGE_n
            CR_XSAVE_PAGE_n := TMP_XSAVE_PAGE_PA_n;
        ENDFOR

            THEN
IF ((TMP_SECS.CET_ATTRIBUTES.SH_STK_EN == 1) OR (TMP_SECS.CET_ATTRIBUTES.ENDBR_EN == 1))
    THEN
        CR_CET_SAVE_AREA_PA := Physical_Address(DS:TMP_CET_SAVE_AREA);
    FI;
FI;

Flags Affected
None

Protected Mode Exceptions
#GP(0) If executed outside an enclave.
    If CR_TCS_PA.CSSA = 0.
#PF(error code) If a page fault occurs in accessing memory.
    If one or more pages of the target SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
    If CET is enabled for the enclave and the target CET SSA frame is not readable/writable, or does not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions
#GP(0) If executed outside an enclave.
    If CR_TCS_PA.CSSA = 0.
#PF(error code) If a page fault occurs in accessing memory.
    If one or more pages of the target SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
    If CET is enabled for the enclave and the target CET SSA frame is not readable/writable, or does not resolve to a valid PT_REG EPC page.
**EENTER—Enters an Enclave**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 02H ENCLU[EENTER]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function is used to enter an enclave.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EENTER (In)</td>
<td>Content of RBX.CSSA (Out)</td>
<td>Address of a TCS (In)</td>
<td>Address of AEP (In)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address of IP following EENTER (Out)</td>
<td></td>
</tr>
</tbody>
</table>

### Description

The ENCLU[EENTER] instruction transfers execution to an enclave. At the end of the instruction, the logical processor is executing in enclave mode at the RIP computed as EnclaveBase + TCS.OENTRY. If the target address is not within the CS segment (32-bit) or is not canonical (64-bit), a #GP(0) results.

### EENTER Memory Parameter Semantics

<table>
<thead>
<tr>
<th>TCS</th>
<th>Enclave access</th>
</tr>
</thead>
</table>

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

- Address in RBX is not properly aligned.
- Any TCS.FLAGS's must-be-zero bit is not zero.
- TCS pointed to by RBX is not valid or available or locked.
- Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
- The SECS is in use.
- Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
- Any one of DS, ES, CS, SS is not zero.
- If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM \(\neq 3\).
- CR4.OFSXSR \(\neq 1\).
- If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
- If SECS.ATTRIBUTES.AEXNOTIFY \(\neq\) TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

The following operations are performed by EENTER:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or interrupt.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSIZELIM/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 40.1.2):
  - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 40.2.5).
  - On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 40.2.2).
• All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
• On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 40.2.3):
  — All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  — PEBS is suppressed.
  — AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set
  — If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 38-62. Base Concurrency Restrictions of EENTER

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EENTER</td>
<td>TCS [DS:RBX]</td>
<td>Shared #GP</td>
</tr>
</tbody>
</table>

Table 38-63. Additional Concurrency Restrictions of EENTER

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EENTER</td>
<td>TCS [DS:RBX]</td>
<td>Concurrent On Conflict</td>
</tr>
</tbody>
</table>

Operation

Temp Variables in EENTER Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_FSBASE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Proposed base address for FS segment.</td>
</tr>
<tr>
<td>TMP_GSBASE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Proposed base address for FS segment.</td>
</tr>
<tr>
<td>TMP_FSLIMIT</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Highest legal address in proposed FS segment.</td>
</tr>
<tr>
<td>TMP_GSLIMIT</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Highest legal address in proposed GS segment.</td>
</tr>
<tr>
<td>TMP_XSIZE</td>
<td>integer</td>
<td>64</td>
<td>Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.</td>
</tr>
<tr>
<td>TMP_SSA_PAGE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Pointer used to iterate over the SSA pages in the current frame.</td>
</tr>
<tr>
<td>TMP_GPR</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of the GPR area within the current SSA frame.</td>
</tr>
</tbody>
</table>

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( (DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1 )))
  THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
  THEN
IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); Fl;
IF(ES usable and ES.base ≠ 0) #GP(0); Fl;
IF(SS usable and SS.base ≠ 0) #GP(0); Fl;
IF(SS usable and SS.B = 0) #GP(0); Fl;
Fi;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); Fl;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); Fl;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical))
    THEN #GP(0); Fl;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions are operating on TCS)
    THEN #GP(0); Fl;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); Fl;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); Fl;

IF ((EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT # PT_TCS))
    THEN #PF(DS:RBX); Fl;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); Fl;

IF ((DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); Fl;

(* Check proposed FS and GS *)
IF (((DS:RBX).OFSBASE is not 4KByte Aligned) or ((DS:RBX).OGSBASE is not 4KByte Aligned))
    THEN #GP(0); Fl;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF (( (DS:RBX).FLAGS & FFFFFFFFFFFFFFCH) ≠ 0)
    THEN #GP(0); Fl;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); Fl;

(* make sure the logical processor’s operating mode matches the enclave *)
IF ((TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT))
    THEN #GP(0); Fl;
IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM # 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) # TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
    FI;

IF ((DS:RBX).CSSA.FLAGS.DBGOPTIN = 0) and (DS:RBX).CSSA.FLAGS.AEXNOTIFY ≠ TMP_SECS.ATTRIBUTES.AEXNOTIFY)
    THEN #GP(0); FI;

(* Make sure the SSA contains at least one more frame *)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);
FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort, and deliver that fault;

    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS # DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS # EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
If a fault occurs; release locks, abort, and deliver that fault;

IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
    IF (EPCM(DS:TMP_GPR).VALID = 0)
        THEN #PF(DS:TMP_GPR); FI;
    IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
        THEN #PF(DS:TMP_GPR); FI;
    IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
        THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
  THEN
    IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
  FI;

CR_GPR_PA := Physical_Address (DS: TMP_GPR);

(* Validate TCS.OENTRY *)
TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
IF (TMP_MODE64 = 1)
  THEN
    IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
  ELSE
    IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
  FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
  THEN
    TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
    TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
    (* if FS wrap-around, make sure DS has no holes*)
    IF (TMP_FSLIMIT < TMP_FSBASE)
      THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
      ELSE
        IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
      FI;
    (* if GS wrap-around, make sure DS has no holes*)
    IF (TMP_GSLIMIT < TMP_GSBASE)
      THEN
        IF (DS.limit < 4GB) THEN #GP(0); FI;
      ELSE
        IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
      FI;
    ELSE
      TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
      TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
      IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
        THEN #GP(0); FI;
    FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE)
  THEN #GP(0); FI;

TMP_IA32_U_CET := 0
TMP_SSP := 0

    IF ( CR4.CET = 0 ) THEN 
        (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *) 
        IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 ) #GP(0); FI; 
        FI; 
    IF ( TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1 ) THEN 
        IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI; 
        FI; 
    FI; 

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN) THEN 
    (* Setup CET state from SECS, note tracker goes to IDLE *) 
    TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES; 
    IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1 ) THEN 
        TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR; 
        TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE; 
        FI; 
    FI; 

    (* Compute linear address of what will become new CET state save area and cache its PA *) 
    TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16 
    TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF; 

    Check the TMP_CET_SAVE_PAGE page is read/write accessible 
    If fault occurs release locks, abort, and deliver fault  
    (* Read the EPCM VALID, PENDING, MODIFIED, BLOCKED, and PT fields atomically *) 
    IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).PT # PT_SS_REST) OR 
        (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS # EPCM(DS:RBX).ENCLAVESECS)) THEN 
        #PF(DS:TMP_CET_SAVE_PAGE); 
        FI; 
    CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA) 

    IF TMP_IA32_U_CET.SH_STK_EN = 1 THEN 
        TMP_SSP = TCS.PREVSSP; 
        FI; 
    FI;
FI;

CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_EL RANGE := (TMPSECS.BASEADDR, TMP_SECS.SIZE);

(* Save state for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 := XCR0;
    XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
FI;

RCX := RIP;
RIP := TMP_TARGET;
RAX := (DS:RBX).CSSA;

(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
DS:TMP_SSA.U_RSP := RSP;
DS:TMP_SSA.U_RBP := RBP;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DSL;
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_EL RANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress_all_code_breakpoints_that_overlap_with_EL RANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress_monitor_trap_flag_for_the_source_of_the_execution_of_the_enclave;
        Suppress_any_pending_debug_exceptions;
        Suppress_any_pending_MTF_VM_exit;
    ELSE
        IF RFLAGS.TF = 1
            THEN pend a single-step #DB at the end of EENTER; Fi;
        IF the "monitor trap flag" VM-execution control is set
            THEN pend an MTF VM exit at the end of EENTER; Fi;
    Fi;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP
            Fi;
        IA32_U_CET := TMP_IA32_U_CET;
    Fi;

Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;

**Flags Affected**

RFLAGS.TF is cleared on opt-out entry.
Protected Mode Exceptions

#GP(0)  If DS:RBX is not page aligned.
       If the enclave is not initialized.
       If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
       If the thread is not in the INACTIVE state.
       If CS, DS, ES or SS bases are not all zero.
       If executed in enclave mode.
       If any reserved field in the TCS FLAG is set.
       If the target address is not within the CS segment.
       If CR4.OSFXSR = 0.
       If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
       If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
       If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

#PF(error code)  If a page fault occurs in accessing memory.
       If DS:RBX does not point to a valid TCS.
       If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)  If DS:RBX is not page aligned.
       If the enclave is not initialized.
       If the thread is not in the INACTIVE state.
       If CS, DS, ES or SS bases are not all zero.
       If executed in enclave mode.
       If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
       If the target address is not canonical.
       If CR4.OSFXSR = 0.
       If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
       If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
       If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

#PF(error code)  If a page fault occurs in accessing memory operands.
       If DS:RBX does not point to a valid TCS.
       If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
**EEXIT—Exits an Enclave**

**Description**
The ENCLU[EEXIT] instruction exits the currently executing enclave and branches to the location specified in RBX. RCX receives the current AEP. If RBX is not within the CS (32-bit mode) or is not canonical (64-bit mode), a #GP(0) results.

**EEXIT Memory Parameter Semantics**
If RBX specifies an address that is inside the enclave, the instruction will complete normally. The fetch of the next instruction will occur in non-enclave mode, but will attempt to fetch from inside the enclave. This fetch returns a fixed data pattern.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

**Concurrency Restrictions**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEXIT</td>
<td></td>
<td>Access</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEXIT</td>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>EEXIT</td>
<td></td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
Operation

Temp Variables in EEXIT Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_RIP</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Saved copy of CRIP for use when creating LBR.</td>
</tr>
</tbody>
</table>

\[\text{TMP\_MODE64} := ((\text{IA32\_EFER.LMA} = 1) \&\& (\text{CS.L} = 1));\]

\[\text{IF} (\text{TMP\_MODE64} = 1) \text{ THEN}\]
\[\text{IF (RBX is not canonical) THEN \#GP(0); FI; ELSE IF (RBX > CS limit) THEN \#GP(0); FI; FI;}

\[\text{TMP\_RIP} := \text{CRIP};\]
\[\text{RIP} := \text{RBX};\]

\[\text{(* Return current AEP in RCX *)}\]
\[\text{RCX} := \text{CR\_TCS\_PA\_AEP};\]

\[\text{(* Do the FS/GS swap *)}\]
\[\text{FS.selector} := \text{CR\_SAVE\_FS.selector};\]
\[\text{FS.base} := \text{CR\_SAVE\_FS.base};\]
\[\text{FS.limit} := \text{CR\_SAVE\_FS.limit};\]
\[\text{FS.access\_rights} := \text{CR\_SAVE\_FS.access\_rights};\]
\[\text{GS.selector} := \text{CR\_SAVE\_GS.selector};\]
\[\text{GS.base} := \text{CR\_SAVE\_GS.base};\]
\[\text{GS.limit} := \text{CR\_SAVE\_GS.limit};\]
\[\text{GS.access\_rights} := \text{CR\_SAVE\_GS.access\_rights};\]

\[\text{(* Restore XCR0 if needed *)}\]
\[\text{IF (CR4.OSXSAVE} = 1) \text{ THEN}\]
\[\text{XCR0} := \text{CR\_SAVE\_XCR0};\]
\[\text{FI; Unsuppress\_all\_code\_breakpoints\_that\_are\_outside\_ELRANGE;}\]

\[\text{IF (CR\_DBGOPTIN} = 0) \text{ THEN}\]
\[\text{UnSuppress\_all\_code\_breakpoints\_that\_overlap\_with\_ELRANGE;}
\text{Restoresuppressedbreakpointmatches;}
\text{RFLAGS.TF} := \text{CR\_SAVE\_TF};
\text{UnSuppress\_monitor\_trap\_flag;}
\text{UnSuppress\_LBR\_Generation;}
\text{UnSuppress\_performance\_monitoring\_activity;}
\text{RestoreperformancemonitoringcounterAnyThreaddemotiontoMyThreadinenclavedbacktoAnyThread}
\text{FI;}

\[\text{IF RFLAGS.TF} = 1 \text{ THEN Pend Single-Step \#DB at the end of EEXIT;}
\text{FI;}

Vol. 3D 38-113
IF the “monitor trap flag” VM-execution control is set
  THEN pend a MTF VM exit at the end of EEXIT;
FI;

  THEN
    (* Record PREVSSP *)
    IF (IA32_U_CET.SH_STK_EN == 1)
      THEN CR_TCS_PA.PREVSSP = SSP; FI;
  FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 1)
  THEN
    (* Restore enclosing app's CET state from the save registers *)
    IA32_U_CET := CR_SAVE_IA32_U_CET;
    IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
      THEN SSP := CR_SAVE_SSP; FI;
    (* Update enclosing app's TRACKER if enclosing app has indirect branch tracking enabled *)
    IF (CR4.CET = 1 AND IA32_U_CET.ENDBR_EN = 1)
      THEN
        IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
        IA32_U_CET.SUPPRESS := 0
      FI;
  FI;

CR_ENCLAVE_MODE := 0;
CR_TCS_PA.STATE := INACTIVE;

(* Assure consistent translations *)
Flush_linear_context;

**Flags Affected**

RFLAGS.TF is restored from the value previously saved in EENTER or ERESUME.

**Protected Mode Exceptions**

- **#GP(0)** If executed outside an enclave.
  If RBX is outside the CS segment.
- **#PF(error code)** If a page fault occurs in accessing memory.

**64-Bit Mode Exceptions**

- **#GP(0)** If executed outside an enclave.
  If RBX is not canonical.
- **#PF(error code)** If a page fault occurs in accessing memory operands.
EGETKEY—Retrieves a Cryptographic Key

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 01H ENCLU[EGETKEY]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function retrieves a cryptographic key.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EGETKEY (In)</td>
<td>Return error code (Out)</td>
<td>Address to a KEYREQUEST (In)</td>
</tr>
</tbody>
</table>

Description

The ENCLU[EGETKEY] instruction returns a 128-bit secret key from the processor specific key hierarchy. The register RBX contains the effective address of a KEYREQUEST structure, which the instruction interprets to determine the key being requested. The Requesting Keys section below provides a description of the keys that can be requested. The RCX register contains the effective address where the key will be returned. Both the addresses in RBX & RCX should be locations inside the enclave.

EGETKEY derives keys using a processor unique value to create a specific key based on a number of possible inputs. This instruction leaf can only be executed inside an enclave.

EEGETKEY Memory Parameter Semantics

<table>
<thead>
<tr>
<th>KEYREQUEST</th>
<th>OUTPUTDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enclave read access</td>
<td>Enclave write access</td>
</tr>
</tbody>
</table>

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on the Table 38-66.
- Computes derived key using the derivation data and package specific value.
- Outputs the calculated key to the address in RCX.

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.(ZF, CF, AF, OF, SF, PF). The instruction returns an error code if the user tries to request a key based on an invalid CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX_INVALID_SVN, SGX_INVALID_ATTRIBUTE, SGX_INVALID_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

Requesting Keys

The KEYREQUEST structure (see Section 35.18.1) identifies the key to be provided. The Keyrequest.KeyName field identifies which type of key is requested.

Deriving Keys

Key derivation is based on a combination of the enclave specific values (see Table 38-66) and a processor key. Depending on the key being requested a field may either be included by definition or the value may be included from the KeyRequest. A "yes" in Table 38-66 indicates the value for the field is included from its default location, identified in the source row, and a "request" indicates the values for the field is included from its corresponding KeyRequest field.
Keys that permit the specification of a CPU or ISV's code's, or enclave configuration's SVN's have additional requirements. The caller may not request a key for an SVN beyond the current CPU, ISV or enclave configuration's SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN_KEY be set and SECS.MRSIGNER equal IA32_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcode PKCS padding constant (352 byte string):

\[
\text{HARDCODED_PKCS1_5_PADDING}[15:0] := 0100H;
\]
\[
\text{HARDCODED_PKCS1_5_PADDING}[2655:16] := \text{SignExtend330Byte}(-1); // 330 bytes of \text{0FFH}
\]
\[
\text{HARDCODED_PKCS1_5_PADDING}[2815:2656] := 2004000501020403650148866009060D30313000H;
\]

The error codes are:

### Table 38-66. Key Derivation

<table>
<thead>
<tr>
<th>Source</th>
<th>Key Name</th>
<th>Attributes</th>
<th>Owner Epoch</th>
<th>CPU SVN</th>
<th>ISV SVN</th>
<th>ISV PRODID</th>
<th>ISVEXT PRODID</th>
<th>ISVFAM ILYID</th>
<th>MRENCLAVE</th>
<th>MRSIGNER</th>
<th>CONFIG ID</th>
<th>CONFIG SVN</th>
<th>RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Dependent Constant</td>
<td>Y := SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIB UTES;</td>
<td>R := AttribMask &amp; SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIB UTES;</td>
<td>R := Request</td>
<td>Y := CPUSVN Register;</td>
<td>Y := CPUSVN Register;</td>
<td>R := Request</td>
<td>SECS.ISVID</td>
<td>SECS.ISVEXTPR ODID</td>
<td>SECS.ISVFAMIL YID</td>
<td>SECS.MRENCLAVE</td>
<td>SECS.MRSIGNER</td>
<td>SECS.CONFIG ID</td>
<td>SECS.CONFIG SVN</td>
</tr>
<tr>
<td>EINITTOKEN</td>
<td>Yes</td>
<td>Request</td>
<td>Yes</td>
<td>Request</td>
<td>Request</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Report</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Request</td>
</tr>
<tr>
<td>Seal</td>
<td>Yes</td>
<td>Request</td>
<td>Yes</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
</tr>
<tr>
<td>Provisioning</td>
<td>Yes</td>
<td>Request</td>
<td>No</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
</tr>
<tr>
<td>Provisioning Seal</td>
<td>Yes</td>
<td>Request</td>
<td>No</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>Request</td>
<td>No</td>
<td>Yes</td>
<td>Request</td>
</tr>
</tbody>
</table>

### Table 38-67. EGETKEY Return Value in RAX

<table>
<thead>
<tr>
<th>Error Code (see Table 38-4)</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td>EGETKEY successful.</td>
</tr>
<tr>
<td>SGX_INVALID_ATTRIBUTE</td>
<td>The KEYREQUEST contains a KEYNAME for which the enclave is not authorized.</td>
<td></td>
</tr>
<tr>
<td>SGX_INVALID_CPUSVN</td>
<td>If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value.</td>
<td></td>
</tr>
<tr>
<td>SGX_INVALID_ISVSVN</td>
<td>If KEYREQUEST software SVN (ISVSVN or CONFIGSVN) is greater than the enclave’s corresponding SVN.</td>
<td></td>
</tr>
<tr>
<td>SGX_INVALID_KEYNAME</td>
<td>If KEYREQUEST.KEYNAME is an unsupported value.</td>
<td></td>
</tr>
</tbody>
</table>

### Concurrency Restrictions

#### Table 38-68. Base Concurrency Restrictions of EGETKEY

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGETKEY</td>
<td>KEYREQUEST [DS:RBX]</td>
<td>Concurrent</td>
</tr>
<tr>
<td>OUTPUTDATA [DS:RCX]</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>
**Operation**

**Temp Variables in EGETKEY Operational Flow**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (Bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_CURRENTSECS</td>
<td></td>
<td></td>
<td>Address of the SECS for the currently executing enclave.</td>
</tr>
<tr>
<td>TMP_KEYDEPENDENCIES</td>
<td></td>
<td></td>
<td>Temp space for key derivation.</td>
</tr>
<tr>
<td>TMP_ATTRIBUTES</td>
<td></td>
<td>128</td>
<td>Temp Space for the calculation of the sealable Attributes.</td>
</tr>
<tr>
<td>TMP_ISVEXTPRODID</td>
<td></td>
<td>16 bytes</td>
<td>Temp Space for ISVEXTPRODID.</td>
</tr>
<tr>
<td>TMP_ISVPRODID</td>
<td></td>
<td>2 bytes</td>
<td>Temp Space for ISVPRODID.</td>
</tr>
<tr>
<td>TMP_ISVFAMILYID</td>
<td></td>
<td>16 bytes</td>
<td>Temp Space for ISVFAMILYID.</td>
</tr>
<tr>
<td>TMP_CONFIGID</td>
<td></td>
<td>64 bytes</td>
<td>Temp Space for CONFIGID.</td>
</tr>
<tr>
<td>TMP_CONFIGSVN</td>
<td></td>
<td>2 bytes</td>
<td>Temp Space for CONFIGSVN.</td>
</tr>
<tr>
<td>TMP_OUTPUTKEY</td>
<td></td>
<td>128</td>
<td>Temp Space for the calculation of the key.</td>
</tr>
</tbody>
</table>

(* Make sure KEYREQUEST is properly aligned and inside the current enclave *)

IF ( (DS:RBX is not 512Byte aligned) or (DS:RBX is not within CR_ELRANGE) )
    THEN #GP(0); Fi;

(* Make sure DS:RBX is an EPC address and the EPC page is valid *)

IF ( (DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0) )
    THEN #PF(DS:RBX); Fi;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); Fi;

(* Check page parameters for correctness *)

IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH)) or (EPCM(DS:RBX).R = 0) )
    THEN #PF(DS:RBX);
Fi;

(* Make sure OUTPUTDATA is properly aligned and inside the current enclave *)

IF ( (DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); Fi;

(* Make sure DS:RCX is an EPC address and the EPC page is valid *)

IF ( (DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0) )

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter [DS:RBX]</th>
<th>Additional Concurrency Restrictions</th>
<th>Additional Concurrency Restrictions</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGETKEY</td>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
<td>vs. EADD, EEXTEND, EINIT</td>
<td>vs. ETRACK, ETRACKC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access On Conflict</td>
<td>Access On Conflict</td>
<td>Access On Conflict</td>
</tr>
</tbody>
</table>
THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH)) or (EPCM(DS:RCX).W = 0) )
    THEN #PF(DS:RCX);
FI;

(* Verify RESERVED spaces in KEYREQUEST are valid *)
IF ( (DS:RBX).RESERVED ≠ 0) or (DS:RBX.KEYPOLICY.RESERVED ≠ 0) )
    THEN #GP(0); FI;

TMP_CURRENTSECS := CR_ACTIVE_SECS;

(* Verify that CONFIGSVN & New Policy bits are not used if KSS is not enabled *)
IF (((TMP_CURRENTSECS.ATTRIBUTES.KSS == 0) AND ((DS:RBX.KEYPOLICY & 0x003C ≠ 0) OR (DS:RBX.CONFIGSVN > 0))))
    THEN #GP(0); FI;

(* Determine which enclave attributes that must be included in the key. Attributes that must always be include INIT & DEBUG *)
REQUIRED_SEALING_MASK[127:0] := 00000000 00000000 00000000 00000003H;
TMP_ATTRIBUTES := (DS:RBX.ATTRIBUTEMASK | REQUIRED_SEALING_MASK) & TMP_CURRENTSECS.ATTRIBUTES;

(* Compute MISCESELECT fields to be included *)
TMP_MISCESELECT := DS:RBX.MISCMASK & TMP_CURRENTSECS.MISCESELECT

(* Compute CET_ATTRIBUTES fields to be included *)
    THEN TMP_CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES_ MASK & TMP_CURRENTSECS.CET_ATTRIBUTES; FI;
TMP_KEYDEPENDENCIES := 0;

CASE (DS:RBX.KEYNAME)
    SEAL_KEY:
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_INVALID_CPUSVN;
                GOTO EXIT;
            FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_INVALID_ISVSVN;
                GOTO EXIT;
            FI;
        IF (DS:RBX.CONFIGSVN > TMP_CURRENTSECS.CONFIGSVN)
            THEN
                RFLAGS.ZF := 1;
                RAX := SGX_INVALID_ISVSVN;
                GOTO EXIT;
            FI;
        (*Include enclave identity?*)
TMP_MRENCLAVE := 0;
IF (DS:RBX.KEYPOLICY.MRENCLAVE = 1)
    THEN TMP_MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
Fi;
(*Include enclave author?*)
TMP_MRSIGNER := 0;
IF (DS:RBX.KEYPOLICY.MRSIGNER = 1)
    THEN TMP_MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
FI;

(* Include enclave product family ID? *)
TMP_ISVFAMILYID := 0;
IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
    THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
FI;

(* Include enclave product ID? *)
TMP_ISVPRODID := 0;
IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
    TMP_ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
FI;

(* Include enclave Config ID? *)
TMP_CONFIGID := 0;
TMP_CONFIGSVN := 0;
IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
    TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;
    TMP_CONFIGSVN := DS:RBX.CONFIGSVN;
FI;

(* Include enclave extended product ID? *)
TMP_ISVEXTPRODID := 0;
IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
    TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
FI;

//Determine values key is based on
TMP_KEYDEPENDENCIES.KEYNAME := SEAL_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTEMASK := DS:RBX.ATTRIBUTEMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_MRENCLAVE;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEALEU FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
  TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
  TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := DS:RBX.CET_ATTRIBUTES _MASK;
FI;
BREAK;

REPORT_KEY:

  //Determine values key is based on
  TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
  TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
  TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
  TMP_KEYDEPENDENCIES.ISVPRODID := 0;
  TMP_KEYDEPENDENCIES.ISVSVN := 0;
  TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
  TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
  TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
  TMP_KEYDEPENDENCIES.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
  TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
  TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
  TMP_KEYDEPENDENCIES.SEALED_KEY_FUSES := CR_SEAL_FUSES;
  TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
  TMP_KEYDEPENDENCIES.PADDING := HARDCODED_PKCS1_5_PADDING;
  TMP_KEYDEPENDENCIES.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
  TMP_KEYDEPENDENCIES.MISCMASK := 0;
  TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
  TMP_KEYDEPENDENCIES.CONFIGID := TMP_CURRENTSECS.CONFIGID;
  TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
  TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES;
  TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
FI;
BREAK;
EINITTOKEN_KEY:

  (* Check ENCLAVE has EINITTOKEN Key capability *)
  IF (TMP_CURRENTSECS.ATTRIBUTES.EINITTOKEN_KEY = 0) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
  FI;
  IF (DS:RBX.CPUSVN is beyond current CPU configuration) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_CPUSVN;
    GOTO EXIT;
  FI;
  IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN) THEN
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_ISVSVN;
    GOTO EXIT;
  FI;
(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := DS:RBX.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;
TMP_KEYDEPENDENCIES.CONFIGSVN := 0;
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK := 0;
    FI;
BREAK;
PROVISION_KEY:
(* Check ENCLAVE has PROVISIONING capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
    FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
    FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
    FI;
(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTEMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := 0;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := 0;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := 0;

    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_M := 0;
    FI;
BREAK;

PROVISION SEAL KEY:
(* Check ENCLAVE has PROVISIONING capability *)
IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ATTRIBUTE;
        GOTO EXIT;
    FI;
IF (DS:RBX.CPUSVN is beyond current CPU configuration)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_CPUSVN;
        GOTO EXIT;
    FI;
IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
    THEN
        RFLAGS.ZF := 1;
        RAX := SGX_INVALID_ISVSVN;
        GOTO EXIT;
    FI;

(* Include enclave product family ID? *)
TMP_ISVFAMILYID := 0;
IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
    THEN TMP_ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
    FI;

(* Include enclave product ID? *)
TMP_ISVPROID := 0;
IF (DS:RBX.KEYPOLICY.NOISVPROID = 0)
    THEN TMP_ISVPROID := TMP_CURRENTSECS.ISVPROID;
    FI;

(* Include enclave Config ID? *)
TMP_CONFIGID := 0;
TMP_CONFIGSVN := 0;
IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
    THEN TMP_CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_CONFIGSVN := DS:RBX.CONFIGSVN;

(* Include enclave extended product ID? *)
TMP_ISVEXTPRODID := 0;
IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
  TMP_ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
FI;

(* Determine values key is based on *)
TMP_KEYDEPENDENCIES.KEYNAME := PROVISION_SEAL_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := TMP_ISVFAMILYID;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := TMP_ISVEXTPRODID;
TMP_KEYDEPENDENCIES.ISVPRODID := TMP_ISVPRODID;
TMP_KEYDEPENDENCIES.ISVSVN := DS:RBX.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := 0;
TMP_KEYDEPENDENCIES.ATTRIBUTES := TMP_ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := DS:RBX.ATTRIBUTEMASK;
TMP_KEYDEPENDENCIES.MRENCLAVE := 0;
TMP_KEYDEPENDENCIES.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_KEYDEPENDENCIES.KEYID := 0;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := DS:RBX.CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := TMP_MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := ~DS:RBX.MISCMASK;
TMP_KEYDEPENDENCIES.KEYPOLICY := DS:RBX.KEYPOLICY;
TMP_KEYDEPENDENCIES.CONFIGID := TMP_CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := TMP_CONFIGSVN;
  THEN
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := TMP_CET_ATTRIBUTES;
    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := 0;
  FI;
DEFAULT:
  (* The value of KEYNAME is invalid *)
  RFLAGS.ZF := 1;
  RAX := SGX_INVALID_KEYNAME;
  GOTO EXIT:
ESAC;

(* Calculate the final derived key and output to the address in RCX *)
TMP_OUTPUTKEY := derivekey(TMP_KEYDEPENDENCIES);
DS:RCX[15:0] := TMP_OUTPUTKEY;
RAX := 0;
RFLAGS.ZF := 0;
EXIT:
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;
Flags Affected
ZF is cleared if successful, otherwise ZF is set. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions
#GP(0) If executed outside an enclave.
If a memory operand effective address is outside the current enclave.
If an effective address is not properly aligned.
If an effective address is outside the DS segment limit.
If KEYREQUEST format is invalid.
#PF(error code) If a page fault occurs in accessing memory.

64-Bit Mode Exceptions
#GP(0) If executed outside an enclave.
If a memory operand effective address is outside the current enclave.
If an effective address is not properly aligned.
If an effective address is not canonical.
If KEYREQUEST format is invalid.
#PF(error code) If a page fault occurs in accessing memory operands.
**EMODPE—Extend an EPC Page Permissions**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 06H ENCLU[EMODPE]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX2</td>
<td>This leaf function extends the access rights of an existing EPC page.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EMODPE (In)</td>
<td>Address of a SECINFO (In)</td>
<td>Address of the destination EPC page (In)</td>
</tr>
</tbody>
</table>

**Description**

This leaf function extends the access rights associated with an existing EPC page in the running enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not extend the page permissions will have no effect. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPE leaf function.

**EMODPE Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>SECINFO</th>
<th>EPCPAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Non Enclave</td>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EMODPE Faulting Conditions**

- The operands are not properly aligned. If security attributes of the SECINFO page make the page inaccessible.
- The EPC page is locked by another thread. RBX does not contain an effective address in an EPC page in the running enclave.
- The EPC page is not valid. RCX does not contain an effective address of an EPC page in the running enclave.
- SECINFO contains an invalid request.

**Concurrency Restrictions**

**Table 38-70. Base Concurrency Restrictions of EMODPE**

<table>
<thead>
<tr>
<th>Leaf Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMODPE Target [DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 38-71. Additional Concurrency Restrictions of EMODPE**

<table>
<thead>
<tr>
<th>Leaf Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</th>
<th>Access</th>
<th>On Conflict</th>
<th>vs. EADD, EEXTEND, EINIT</th>
<th>Access</th>
<th>On Conflict</th>
<th>vs. ETRACK, ETRACKC</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMODPE Target [DS:RCX]</td>
<td>Exclusive</td>
<td>#GP</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SECINFO [DS:RBX]</td>
<td>Concurrent</td>
<td></td>
<td>Concurrent</td>
<td>Concurrent</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Temp Variables in EMODPE Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRATCH_SECINFO</td>
<td>SECINFO</td>
<td>512</td>
<td>Scratch storage for holding the contents of DS:RBX.</td>
</tr>
</tbody>
</table>

IF (DS:RBX is not 64Byte Aligned) THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned) THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) ) THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC) THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC) THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RBX).VALID = 0) or (EPCM(DS:RBX).PENDING ≠ 0) or (EPCM(DS:RBX).MODIFIED ≠ 0) or (EPCM(DS:RBX).BLOCKED ≠ 0) or (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0xFFF)) ) THEN #PF(DS:RBX); FI;

SCRATCH_SECINFO := DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero ) THEN #GP(0); FI;

(* Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or (EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) ) THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction) THEN #GP(0); FI;

(* Re-Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX)) THEN #PF(DS:RCX); FI;

(* Check for misconfigured SECINFO flags*)
IF ( (EPCM(DS:RCX).R = 0) and (SCRATCH_SECINFO.FLAGS.R = 0) and (SCRATCH_SECINFO.FLAGS.W ≠ 0) ) THEN #GP(0); FI;
(* Update EPCM permissions *)

**Flags Affected**
None

**Protected Mode Exceptions**

#GP(0) If executed outside an enclave.
  If a memory operand effective address is outside the DS segment limit.
  If a memory operand is not properly aligned.
  If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.

**64-Bit Mode Exceptions**

#GP(0) If executed outside an enclave.
  If a memory operand is non-canonical form.
  If a memory operand is not properly aligned.
  If a memory operand is locked.

#PF(error code) If a page fault occurs in accessing memory operands.
EREPORT—Create a Cryptographic Report of the Enclave

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 00H ENCLU[EREPORT]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function creates a cryptographic report of the enclave.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EREPORT (In)</td>
<td>Address of TARGETINFO (In)</td>
<td>Address of REPORTDATA (In)</td>
<td>Address where the REPORT is written to in an OUTPUTDATA (In)</td>
</tr>
</tbody>
</table>

Description

This leaf function creates a cryptographic REPORT that describes the contents of the enclave. This instruction leaf can only be executed when inside the enclave. The cryptographic report can be used by other enclaves to determine that the enclave is running on the same platform.

RBX contains the effective address of the MRENCLAVE value of the enclave that will authenticate the REPORT output, using the REPORT key delivered by EGETKEY command for that enclave. RCX contains the effective address of a 64-byte REPORTDATA structure, which allows the caller of the instruction to associate data with the enclave from which the instruction is called. RDX contains the address where the REPORT will be output by the instruction.

EREPORT Memory Parameter Semantics

<table>
<thead>
<tr>
<th>TARGETINFO</th>
<th>REPORTDATA</th>
<th>OUTPUTDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access by Enclave</td>
<td>Read access by Enclave</td>
<td>Read/Write access by Enclave</td>
</tr>
</tbody>
</table>

This instruction leaf perform the following:
1. Validate the 3 operands (RBX, RCX, RDX) are inside the enclave.
2. Compute a report key for the target enclave, as indicated by the value located in RBX(TARGETINFO).
3. Assemble the enclave SECS data to complete the REPORT structure (including the data provided using the RCX (REPORTDATA) operand).
4. Computes a cryptographic hash over REPORT structure.
5. Add the computed hash to the REPORT structure.
6. Output the completed REPORT structure to the address in RDX (OUTPUTDATA).

The instruction fails if the operands are not properly aligned.
CR_REPORT_KEYID, used to provide key wearout protection, is populated with a statistically unique value on boot of the platform by a trusted entity within the SGX TCB.

The instruction faults if any of the following:

EREPORT Faulting Conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>An effective address not properly aligned.</td>
<td>An memory address does not resolve in an EPC page.</td>
</tr>
<tr>
<td>If accessing an invalid EPC page.</td>
<td>If the EPC page is blocked.</td>
</tr>
<tr>
<td>May page fault.</td>
<td></td>
</tr>
</tbody>
</table>

38-128 Vol. 3D
Concurrence Restrictions

Table 38-72. Base Concurrence Restrictions of EREPORT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrence Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EREPORT</td>
<td>TARGETINFO [DS:RBX]</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>REPORTDATA [DS:RCX]</td>
<td>Concurrent</td>
</tr>
<tr>
<td></td>
<td>OUTPUTDATA [DS:RDX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Table 38-73. Additional Concurrence Restrictions of EREPORT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrence Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TARGETINFO [DS:RBX]</td>
<td>Concurrent, Concurrent, Concurrent</td>
</tr>
<tr>
<td></td>
<td>REPORTDATA [DS:RCX]</td>
<td>Concurrent, Concurrent, Concurrent</td>
</tr>
<tr>
<td></td>
<td>OUTPUTDATA [DS:RDX]</td>
<td>Concurrent, Concurrent, Concurrent</td>
</tr>
</tbody>
</table>

Operation

Temp Variables in EREPORT Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_ATTRIBUTES</td>
<td>32</td>
<td>Physical address of SECS of the enclave to which source operand belongs.</td>
<td></td>
</tr>
<tr>
<td>TMP_CURRENTSECS</td>
<td></td>
<td>Address of the SECS for the currently executing enclave.</td>
<td></td>
</tr>
<tr>
<td>TMP_KEYDEPENDENCIES</td>
<td></td>
<td>Temp space for key derivation.</td>
<td></td>
</tr>
<tr>
<td>TMP_REPORTKEY</td>
<td>128</td>
<td>REPORTKEY generated by the instruction.</td>
<td></td>
</tr>
<tr>
<td>TMP_REPORT</td>
<td>3712</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Address verification for TARGETINFO (RBX) *)

IF ( (DS:RBX is not 512Byte Aligned) or (DS:RBX is not within CR_ELRANGE) )
    THEN #GP(0); Fi;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); Fi;

IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); Fi;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); Fi;

(* Check page parameters for correctness *)

IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH)) or (EPCM(DS:RBX).R = 0) )
THEN #PF(DS:RBX); Fl;

(* Verify RESERVED spaces in TARGETINFO are valid *)
IF (DS:RBX.RESERVED != 0)
    THEN #GP(0); Fl;

(* Address verification for REPORTDATA (RCX) *)
IF ( (DS:RCX is not 128Byte Aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); Fl;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); Fl;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); Fl;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); Fl;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH)) or (EPCM(DS:RCX).R = 0) )
    THEN #PF(DS:RCX);
Fl;

(* Address verification for OUTPUTDATA (RDX) *)
IF ( (DS:RDX is not 512Byte Aligned) or (DS:RDX is not within CR_ELRANGE) )
    THEN #GP(0); Fl;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); Fl;

IF (EPCM(DS:RDX).VALID = 0)
    THEN #PF(DS:RDX); Fl;

IF (EPCM(DS:RDX).BLOCKED = 1)
    THEN #PF(DS:RDX); Fl;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH)) or (EPCM(DS:RDX).W = 0) )
    THEN #PF(DS:RDX);
Fl;

(* REPORT MAC needs to be computed over data which cannot be modified *)
TMP_REPORT.CPUSVN := CR_CPUSVN;
TMP_REPORT.ISVFAMILYID := TMP_CURRENTSECS.ISVFAMILYID;
TMP_REPORT.ISVEXTPRODID := TMP_CURRENTSECS.ISVEXTPRODID;
TMP_REPORT.ISVPRODID := TMP_CURRENTSECS.ISVPRODID;
TMP_REPORT.ISVSVN := TMP_CURRENTSECS.ISVSVN;
TMP_REPORT.ATTRIBUTES := TMP_CURRENTSECS.ATTRIBUTES;
TMP_REPORT.REPORTDATA := DS:RCX[511:0];
TMP_REPORT.MRENCLAVE := TMP_CURRENTSECS.MRENCLAVE;
TMP_REPORT.MRSIGNER := TMP_CURRENTSECS.MRSIGNER;
TMP_REPORT.MRRESERVED := 0;
TMP_REPORT.KEYID[255:0] := CR_REPORT_KEYID;
TMP_REPORT.MISCSELECT := TMP_CURRENTSECS.MISCSELECT;
TMP_REPORT.CONFIGID := TMP_CURRENTSECS.CONFIGID;
TMP_REPORT.CONFIGSVN := TMP_CURRENTSECS.CONFIGSVN;
   THEN TMP_REPORT.CET_ATTRIBUTES := TMP_CURRENTSECS.CET_ATTRIBUTES; Fi;

(* Derive the report key *)
TMP_KEYDEPENDENCIES.KEYNAME := REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID := 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID := 0;
TMP_KEYDEPENDENCIES.ISVPRODID := 0;
TMP_KEYDEPENDENCIES.ISVSVN := 0;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH := CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES := DS:RBX.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK := 0;
TMP_KEYDEPENDENCIES.MRENCLAVE := DS:RBX.MEASUREMENT;
TMP_KEYDEPENDENCIES.MRSIGNER := 0;
TMP_KEYDEPENDENCIES.KEYID := TMP_REPORT.KEYID;
TMP_KEYDEPENDENCIES.SEALES_KEY_FUSES := CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN := CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING := TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT := DS:RBX.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK := 0;
TMP_KEYDEPENDENCIES.KEYPOLICY := 0;
TMP_KEYDEPENDENCIES.CONFIGID := DS:RBX.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN := DS:RBX.CONFIGSVN;
   THEN
      TMP_KEYDEPENDENCIES.CET_ATTRIBUTES := DS:RBX.CET_ATTRIBUTES;
      TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK := 0;
   Fi;

(* Calculate the derived key*)
TMP_REPORTKEY := derivekey(TMP_KEYDEPENDENCIES);

(* call cryptographic CMAC function, CMAC data are not including MAC&KEYID *)
TMP_REPORT.MAC := cmac(TMP_REPORTKEY, TMP_REPORT[3071:0]);
DS:RDX[3455: 0] := TMP_REPORT;

Flags Affected
None

Protected Mode Exceptions

#GP(0) If executed outside an enclave.
If the address in RCS is outside the DS segment limit.
If a memory operand is not properly aligned.
If a memory operand is not in the current enclave.
#PF(error code) If a page fault occurs in accessing memory operands.
64-Bit Mode Exceptions

#GP(0) If executed outside an enclave.
   If RCX is non-canonical form.
   If a memory operand is not properly aligned.
   If a memory operand is not in the current enclave.

#PF(error code) If a page fault occurs in accessing memory operands.
ERESUME—Re-Enters an Enclave

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 03H ENCLU[ERESUME]</td>
<td>IR</td>
<td>V/V</td>
<td>SGX1</td>
<td>This leaf function is used to re-enter an enclave after an interrupt.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>RAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ERESUME (In)</td>
<td>Address of a TCS (In)</td>
<td>Address of AEP (In)</td>
</tr>
</tbody>
</table>

**Description**

The ENCLU[ERESUME] instruction resumes execution of an enclave that was interrupted due to an exception or interrupt, using the machine state previously stored in the SSA.

**ERESUME Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>TCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enclave read/write access</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following occurs:

<table>
<thead>
<tr>
<th>Address in RBX is not properly aligned.</th>
<th>Any TCS.FLAGS’s must-be-zero bit is not zero.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCS pointed to by RBX is not valid or available or locked.</td>
<td>Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.</td>
</tr>
<tr>
<td>The SECS is in use by another enclave.</td>
<td>Either of TCS-specified FS and GS segment is not a subset of the current DS segment.</td>
</tr>
<tr>
<td>Any one of DS, ES, CS, SS is not zero.</td>
<td>If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.</td>
</tr>
<tr>
<td>CR4.OSFXSR ≠ 1.</td>
<td>If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.</td>
</tr>
<tr>
<td>Offsets 520-535 of the XSAVE area not 0.</td>
<td>The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.</td>
</tr>
<tr>
<td>The SSA frame is not valid or in use.</td>
<td>If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN ≠ 0.</td>
</tr>
</tbody>
</table>

The following operations are performed by ERESUME:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or an asynchronous exit due to any Interrupt event.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 40.1.2):
  - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 40.2.5).
  - On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 40.2.3).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
• On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 40.2.3):
  — All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  — PEBS is suppressed.
  — AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set.
  — If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERESUME</td>
<td>TCS [DS:RBX]</td>
<td>Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shared</td>
</tr>
</tbody>
</table>

Table 38-74. Base Concurrency Restrictions of ERESUME

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access</td>
</tr>
<tr>
<td>ERESUME</td>
<td>TCS [DS:RBX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Table 38-75. Additional Concurrency Restrictions of ERESUME

Operation

Temp Variables in ERESUME Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_FSBASE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Proposed base address for FS segment.</td>
</tr>
<tr>
<td>TMP_GSBASE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Proposed base address for FS segment.</td>
</tr>
<tr>
<td>TMP_FSLIMIT</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Highest legal address in proposed FS segment.</td>
</tr>
<tr>
<td>TMP_GSLIMIT</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Highest legal address in proposed GS segment.</td>
</tr>
<tr>
<td>TMP_TARGET</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of first instruction inside enclave at which execution is to resume.</td>
</tr>
<tr>
<td>TMP_SECS</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Physical address of SECS for this enclave.</td>
</tr>
<tr>
<td>TMP_SSA</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of current SSA frame.</td>
</tr>
<tr>
<td>TMP_XSIZE</td>
<td>Integer</td>
<td>64</td>
<td>Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.</td>
</tr>
<tr>
<td>TMP_SSA_PAGE</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Pointer used to iterate over the SSA pages in the current frame.</td>
</tr>
<tr>
<td>TMP_GPR</td>
<td>Effective Address</td>
<td>32/64</td>
<td>Address of the GPR area within the current SSA frame.</td>
</tr>
<tr>
<td>TMP.Branch.Record</td>
<td>LBR Record</td>
<td>From/to addresses to be pushed onto the LBR stack.</td>
<td></td>
</tr>
<tr>
<td>TMP_NOTIFY</td>
<td>Boolean</td>
<td>1</td>
<td>When set to 1, deliver an AEX notification.</td>
</tr>
</tbody>
</table>

TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1))})
THEN #GP(O); Fl;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
  THEN
    IF (CS.base ≠ 0 or DS.base ≠ 0) #GP(O); Fl;
    IF (ES usable and ES.base ≠ 0) #GP(O); Fl;
    IF (SS usable and SS.base ≠ 0) #GP(O); Fl;
    IF (SS usable and SS.B = 0) #GP(O); Fl;
  Fl;
IF (DS:RBX is not 4KByte Aligned)
  THEN #GP(O); Fl;
IF (DS:RBX does not resolve within an EPC)
  THEN #PF(DS:RBX); Fl;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical))
  THEN #GP(O); Fl;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions are operating on TCS)
  THEN #GP(O); Fl;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
  THEN #PF(DS:RBX); Fl;
IF (EPCM(DS:RBX).BLOCKED = 1)
  THEN #PF(DS:RBX); Fl;
IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
  THEN #PF(DS:RBX); Fl;
IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS))
  THEN #PF(DS:RBX); Fl;
IF ( (DS:RBX).OSSA is not 4KByte Aligned)
  THEN #GP(O); Fl;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned))
  THEN #GP(O); Fl;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Make sure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFCH) ≠ 0)
  THEN #GP(O); Fl;

(* SECS must exist and enclave must have previously been EINITed *)
IF (the enclave is not already initialized)
THEN #GP(0); Fl;

(* make sure the logical processor's operating mode matches the enclave *)
IF ((TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT))
    THEN #GP(0); Fl;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); Fl;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); Fl;
        ELSE
            IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); Fl;
        FI;
    FI;

IF ((DS:RBX).CSSA.FLAGS.DBGOPTIN = 0) and (DS:RBX).CSSA.FLAGS.AEXNOTIFY ≠ TMP_SECS.ATTRIBUTES.AEXNOTIFY))
    THEN #GP(0); Fl;

(* Make sure the SSA contains at least one active frame *)
IF ((DS:RBX).CSSA = 0)
    THEN #GP(0); Fl;

(* Compute linear address of SSA frame *)
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); Fl;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); Fl;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); Fl;
    IF (EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); Fl;
    IF (EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0)
        THEN #PF(DS:TMP_SSA_PAGE); Fl;
    CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); Fl;
IF (EPCM(DS:TMP_GPR).VALID = 0)
THEN #PF(DS:TMP_GPR); Fi;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); Fi;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); Fi;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
    THEN #PF(DS:TMP_GPR); Fi;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); Fi;
    Fi;

CR_GPR_PA := Physical_Address (DS: TMP_GPR);

IF (((DS:RBX).FLAGS.AEXNOTIFY = 1) and (DS:TMP_GPR.AEXNOTIFY[0] = 1))
    THEN
        TMP_NOTIFY := 1;
    ELSE
        TMP_NOTIFY := 0;
    FI;

IF (TMP_NOTIFY = 1)
    THEN
        (* Make sure the SSA contains at least one more frame *)
            THEN #GP(0); Fi;
        TMP_SSA := TMP_SSA + 4096 * TMP_SECS.SSAFRAMESIZE;
        TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);
        FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
            (* Check page is read/write accessible *)
            Check that DS:TMP_SSA_PAGE is read/write accessible;
            If a fault occurs, release locks, abort and deliver that fault;
            IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
                THEN #PF(DS:TMP_SSA_PAGE); Fi;
            IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
                THEN #PF(DS:TMP_SSA_PAGE); Fi;
            IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
                THEN #PF(DS:TMP_SSA_PAGE); Fi;
            IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or
                (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
                THEN #PF(DS:TMP_SSA_PAGE); Fi;
            IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or
                (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
                (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
                (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0))
                THEN #PF(DS:TMP_SSA_PAGE); Fi;
            CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
        ENDFOR
(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);

If a fault occurs; release locks, abort and deliver that fault;

IF (DS:TMP_GPR does not resolve to EPC page)
   THEN #PF(DS:TMP_GPR); Fl;
IF (EPCM(DS:TMP_GPR).VALID = 0)
   THEN #PF(DS:TMP_GPR); Fl;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
   THEN #PF(DS:TMP_GPR); Fl;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
   THEN #PF(DS:TMP_GPR); Fl;
IF ((EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or
        (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
        (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
   THEN #PF(DS:TMP_GPR); Fl;

IF (TMP_MODE64 = 0)
   THEN
      IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); Fl;
   Fl;
CR_GPR_PA := Physical_Address (DS: TMP_GPR);
TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
ELSE
   TMP_TARGET := (DS:TMP_GPR).RIP;
Fi;

IF (TMP_MODE64 = 1)
   THEN
      IF (TMP_TARGET is not canonical) THEN #GP(0); Fl;
   ELSE
      IF (TMP_TARGET > CS limit) THEN #GP(0); Fl;
   Fi;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
   THEN
      TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
      TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
      (* if FS wrap-around, make sure DS has no holes*)
      IF (TMP_FSLIMIT < TMP_FSBASE)
         THEN
            IF (DS.limit < 4GB) THEN #GP(0); Fl;
         ELSE
            IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); Fl;
         Fi;
      (* if GS wrap-around, make sure DS has no holes*)
      IF (TMP_GSLIMIT < TMP_GSBASE)
         THEN

38-138 Vol. 3D
IF (DS.limit < 4GB) THEN #GP(0); FI;
ELSE
  IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
ENDIF;
ELSE
  IF (TMP_NOTIFY = 1)
    THEN
      TMP_FSBASE := (DS:RBX).OFBASE + TMP_SECS.BASEADDR;
      TMP_GSBASE := (DS:RBX).OGBASE + TMP_SECS.BASEADDR;
    ELSE
      TMP_FSBASE := DS:TMP_GPR.FSBASE;
      TMP_GSBASE := DS:TMP_GPR.GSBASE;
    FI;
  IF ((TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
    THEN #GP(0); FI;
ENDIF;
ENDIF;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE)
  THEN #GP(0); FI;

TMP_IA32_U_CET := 0
TMP_SSP := 0

  THEN
    IF (CR4.CET = 0) THEN
      IF (TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) #GP(0); FI;
    (* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail ERESUME *)
    IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1)
      THEN
        IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
      FI;
    FI;
  FI;

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
  THEN
    (* Setup CET state from SECS, note tracker goes to IDLE *)
    TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
    IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1)
      THEN
        TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
        TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
      FI;
    (* Compute linear address of what will become new CET state save area and cache its PA *)
    IF (TMP_NOTIFY = 1)
      THEN
        TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16;
      ELSE
        TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA - 1) * 16;
      FI;
TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

Check the TMP_CET_SAVE_PAGE page is read/write accessible
If fault occurs release locks, abort and deliver fault

(* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
(EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
THEN
#PF(DS:TMP_CET_SAVE_PAGE);
FI;

CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)
IF (TMP_NOTIFY = 1)
THEN
IF TMP_IA32_U_CET.SH_STK_EN = 1
THEN TMP_SSP = TCS.PREVSSP; FI;
ELSE
TMP_SSP = CR_CET_SAVE_AREA_PA.SSP
TMP_IA32_U_CET.TRACKER = CR_CET_SAVE_AREA_PA.TRACKER;
TMP_IA32_U_CET.SUPPRESS = CR_CET_SAVE_AREA_PA.SUPPRESS;
IF ( (TMP_MODE64 = 1 AND TMP_SSP is not canonical) OR
(TMP_MODE64 = 0 AND (TMP_SSP & 0xFFFFFFFF00000000) ≠ 0) OR
(TMP_SSP is not 4 byte aligned) OR
(TMP_IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH AND TMP_IA32_U_CET.SUPPRESS = 1) OR
(CR_CET_SAVE_AREA_PA.Reserved ≠ 0)) #GP(0); FI;
FI;
FI;
IF (TMP_NOTIFY = 0)
THEN
(* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)
(* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)
XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);
IF (XRSTOR failed with #GP)
THEN
DS:RBX.STATE := INACTIVE;
#GP(0);
FI;
FI;

CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_ELRANGE := (TMP_SECS.BASEADDR, TMP_SECS.SIZE);
(* Save state for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;

IF (TMP_NOTIFY = 1)
    THEN
        (* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
        IF (CR4.OSXSAVE = 1)
            THEN
                CR_SAVE_XCR0 := XCR0;
                XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
            FI;
        FI;

RIP := TMP_TARGET;

IF (TMP_NOTIFY = 1)
    THEN
        RCX := RIP;
        RAX := (DS:RBX).CSSA;
        (* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
        DS:TMP_SSA.U_RSP := RSP;
        DS:TMP_SSA.U_RBP := RBP;
    ELSE
        Restore_GPRs from DS:TMP_GPR;

        (* Restore the RFLAGS values from SSA *)
        RFLAGS.CF := DS:TMP_GPR.RFLAGS.CF;
        RFLAGS.PF := DS:TMP_GPR.RFLAGS.PF;
        RFLAGS.AF := DS:TMP_GPR.RFLAGS.AF;
        RFLAGS.ZF := DS:TMP_GPR.RFLAGS.ZF;
        RFLAGS.SF := DS:TMP_GPR.RFLAGS.SF;
        RFLAGS.DF := DS:TMP_GPR.RFLAGS.DF;
        RFLAGS.OF := DS:TMP_GPR.RFLAGS.OF;
        RFLAGS.NT := DS:TMP_GPR.RFLAGS.NT;
        RFLAGS.AC := DS:TMP_GPR.RFLAGS.AC;
        RFLAGS.ID := DS:TMP_GPR.RFLAGS.ID;
        RFLAGS.RF := DS:TMP_GPR.RFLAGS.RF;
        RFLAGS.VM := 0;
        IF (RFLAGS.IOPL = 3)
            THEN RFLAGS.IF := DS:TMP_GPR.RFLAGS.IF; FI;
    FI;

IF (TCS.FLAGS.OPTIN = 0)
THEN RFLAGS.TF := 0; FI;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
THEN
    CR_SAVE_XCR0 := XCR0;
    XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
FI;

(* Pop the SSA stack*)
(DS:RBX).CSSA := (DS:RBX).CSSA -1;
FI;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS.L;
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
THEN
    Suppress all code breakpoints that overlap with ELRANGE;
    CR_SAVE_TF := RFLAGS.TF;
    RFLAGS.TF := 0;
    Suppress any MTF VM exits during execution of the enclave;
    Clear all pending debug exceptions;
    Clear any pending MTF VM exit;
ELSE
IF (TMP_NOTIFY = 1)
    THEN
        IF RFLAGS.TF = 1
            THEN pend a single-step #DB at the end of ERESUME; FI;
        IF the "monitor trap flag" VM-execution control is set
            THEN pend an MTF VM exit at the end of ERESUME; FI;
    ELSE
        Clear all pending debug exceptions;
        Clear pending MTF VM exits;
    FI;
FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP;
            FI;
        IA32_U_CET := TMP_IA32_U_CET;
        FI;

(* Assure consistent translations *)
Flush_linear_context;
Clear_Monitor_FSM;
Allow_front_end_to_begin_fetch_at_new_RIP;

Flags Affected
RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0) If DS:RBX is not page aligned.
    If the enclave is not initialized.
    If the thread is not in the INACTIVE state.
    If CS, DS, ES or SS bases are not all zero.
    If executed in enclave mode.
    If part or all of the FS or GS segment specified by TCS is outside the DS segment.
    If any reserved field in the TCS FLAG is set.
    If the target address is not within the CS segment.
    If CR4.OSFXSR = 0.
    If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
    If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
    If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

#PF(error code) If a page fault occurs in accessing memory.
    If DS:RBX does not point to a valid TCS.
    If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
64-Bit Mode Exceptions

#GP(0) If DS:RBX is not page aligned.
If the enclave is not initialized.
If the thread is not in the INACTIVE state.
If CS, DS, ES or SS bases are not all zero.
If executed in enclave mode.
If part or all of the FS or GS segment specified by TCS is outside the DS segment.
If any reserved field in the TCS FLAG is set.
If the target address is not canonical.
If CR4.OSFXSR = 0.
If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

#PF(error code) If a page fault occurs in accessing memory operands.
If DS:RBX does not point to a valid TCS.
If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.
38.5 INTEL® SGX VIRTUALIZATION LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLV instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.
EDECVIRTCHILD—Decrement VIRTCHILDCNT in SECS

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 00H ENCLV[EDECVIRTCHILD]</td>
<td>IR</td>
<td>V/V</td>
<td>EAX[5]</td>
<td>This leaf function decrements the SECS VIRTCHILDCNT field.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EDECVIRTCHILD (In)</td>
<td>Return error code (Out)</td>
<td>Address of an enclave page (In)</td>
</tr>
</tbody>
</table>

### Description

This instruction decrements the SECS VIRTCHILDCNT field. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

### EDECVIRTCHILD Memory Parameter Semantics

<table>
<thead>
<tr>
<th>EPCPAGE</th>
<th>SECS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access permitted by Non Enclave</td>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

### EDECVIRTCHILD Faulting Conditions

- A memory operand effective address is outside the DS segment limit (32b mode).
- DS segment is unusable (32b mode).
- A memory address is in a non-canonical form (64b mode).
- A memory operand is not properly aligned.
- A page fault occurs in accessing memory operands.
- RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
- RCX does not refer to an SECS page.
- RBX does not refer to an enclave page associated with SECS referenced in RCX.

### Concurrency Restrictions

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
<td></td>
</tr>
</tbody>
</table>

Table 38-76. Base Concurrency Restrictions of EDECVIRTCHILD
Operation

Temp Variables in EDECVIRTCHILD Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Physical Address</td>
<td>64</td>
<td>Physical address of the SECS of the page being modified.</td>
</tr>
<tr>
<td>TMP_VIRTCHILDCNT</td>
<td>Integer</td>
<td>64</td>
<td>Number of virtual child pages.</td>
</tr>
</tbody>
</table>

EDECVIRTCHILD Return Value in RAX

<table>
<thead>
<tr>
<th>Error</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td>EDECVIRTCHILD Successful.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td>Failure due to concurrent operation of another SGX instruction.</td>
</tr>
<tr>
<td>SGX_INVALID_COUNTER</td>
<td></td>
<td>Attempt to decrement counter that is already zero.</td>
</tr>
</tbody>
</table>

(* check alignment of DS:RBX *)
IF (DS:RBX is not 4K aligned) THEN
  #GP(0); Fi;

(* check DS:RBX is an linear address of an EPC page *)
IF (DS:RBX does not resolve within an EPC) THEN
  #PF(DS:RBX, PFEC.SGX); Fi;

(* check DS:RCX is an linear address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
  #PF(DS:RCX, PFEC.SGX); Fi;

(* Check the EPCPAGE for concurrency *)
IF (EPCPAGE is being modified) THEN
  RFLAGS.ZF = 1;
  RAX = SGX_EPC_PAGE_CONFLICT;
  goto DONE;
Fi;

(* check that the EPC page is valid *)
IF (EPCM(DS:RBX).VALID = 0) THEN
  #PF(DS:RBX, PFEC.SGX); Fi;

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or...
(EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
(EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
(EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))
    THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS := Address of SECS for (DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
    (* get the physical address of DS:RBX *)
    TMP_SECS := Physical_Address(DS:RBX);
ELSE
    (* EDECVIRTCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); Fi;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); Fi;

(* Atomically decrement virtchild counter and check for underflow *)
Locked_Decrement(SECS(TMP_SECS).VIRTCHILDCNT);
IF (There was an underflow) THEN
    Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);
    RFLAGS.ZF := 1;
    RAX := SGX_INVALID_COUNTER;
    goto DONE;
Fi;

RFLAGS.ZF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

Flags Affected
ZF is set if EDECVIRTCHILD fails due to concurrent operation with another SGX instruction, or if there is a VIRT-
CHILDCNT underflow. Otherwise cleared.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the DS segment limit.
If DS segment is unusable.
If a memory operand is not properly aligned.
RBX does not refer to an enclave page associated with SECS referenced in RCX.

#PF(error code) If a page fault occurs in accessing memory operands.
If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
If RCX does not refer to an SECS page.
64-Bit Mode Exceptions

#GP(0) If a memory address is in a non-canonical form.
If a memory operand is not properly aligned.
RBX does not refer to an enclave page associated with SECS referenced in RCX.

#PF(error code) If a page fault occurs in accessing memory operands.
If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
If RCX does not refer to an SECS page.
EINCVIRTCHILD—Increment VIRTCHILDCNT in SECS

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX = 01H ENCLV[EINCVIRTCHILD]</td>
<td>IR</td>
<td>V/V</td>
<td>EAX[5]</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RBX</th>
<th>RCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>EINCVIRTCHILD (In)</td>
<td>Return error code (Out)</td>
<td>Address of an enclave page (In)</td>
</tr>
</tbody>
</table>

**Description**

This instruction increments the SECS VIRTCHILDCNT field. This instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create a linear address. Segment override is not supported.

**EINCVIRTCHILD Memory Parameter Semantics**

<table>
<thead>
<tr>
<th>EPCPAGE</th>
<th>SECS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write access permitted by Non Enclave</td>
<td>Read access permitted by Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

**EINCVIRTCHILD Faulting Conditions**

- A memory operand effective address is outside the DS segment limit (32b mode).
- A page fault occurs in accessing memory operands.
- DS segment is unusable (32b mode).
- RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
- A memory address is in a non-canonical form (64b mode).
- RCX does not refer to an SECS page.
- A memory operand is not properly aligned.
- RBX does not refer to an enclave page associated with SECS referenced in RCX.

**Concurrency Restrictions**

**Table 38-78. Base Concurrency Restrictions of EINCVIRTCHILD**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Access</th>
<th>On Conflict</th>
<th>SGX_CONFLICT VM Exit Qualification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINCVIRTCHILD</td>
<td>Target [DS:RBX]</td>
<td>Shared</td>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
</tr>
<tr>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Operation**

### Temp Variables in EINCVIRTCHELD Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Physical Address</td>
<td>64</td>
<td>Physical address of the SECS of the page being modified.</td>
</tr>
</tbody>
</table>

### EINCVIRTCHELD Return Value in RAX

<table>
<thead>
<tr>
<th>Error</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td>EINCVIRTCHELD Successful.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td>Failure</td>
<td>Failure due to concurrent operation of another SGX instruction.</td>
</tr>
</tbody>
</table>

(* check alignment of DS:RBX *)
IF (DS:RBX is not 4K aligned) THEN
  #GP(0); Fl;

(* check DS:RBX is an linear address of an EPC page *)
IF (DS:RBX does not resolve within an EPC) THEN
  #PF(DS:RBX, PFEC.SGX); Fl;

(* check DS:RCX is an linear address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
  #PF(DS:RCX, PFEC.SGX); Fl;

(* Check the EPCPAGE for concurrency *)
IF (EPCPAGE is being modified) THEN
  RFLAGS.ZF = 1;
  RAX = SGX_EPC_PAGE_CONFLICT;
  goto DONE;
  Fl;

(* check that the EPC page is valid *)
IF (EPCM(DS:RBX).VALID = 0) THEN
  #PF(DS:RBX, PFEC.SGX); Fl;

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST))

---

**Table 38-79. Additional Concurrency Restrictions of EINCVIRTCHELD**

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINCVIRTCHELD</td>
<td>Target [DS:RBX]</td>
<td>Concurrent Access, Concurrent On Conflict</td>
</tr>
<tr>
<td></td>
<td>SECS [DS:RCX]</td>
<td>Concurrent Access, Concurrent On Conflict</td>
</tr>
</tbody>
</table>

---

**Vol. 3D 38-151**
THEN
 (* get the SECS of DS:RBX *)
 TMP_SECS := Address of SECS for (DS:RBX);
 ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
 (* get the physical address of DS:RBX *)
 TMP_SECS := Physical_Address(DS:RBX);
 ELSE
 (* EINCVIRTCHILD called on page of incorrect type *)
 #PF(DS:RBX, PFEC.SGX); Fl;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
 #GP(0); Fl;

(* Atomically increment virtchild counter *)
Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);

RFLAGS.ZF := 0;
RAX := 0;

DONE:
(* clear flags *)
RFLAGS.CF := 0;
RFLAGS.PF := 0;
RFLAGS.AF := 0;
RFLAGS.OF := 0;
RFLAGS.SF := 0;

**Flags Affected**
ZF is set if EINCVIRTCHILD fails due to concurrent operation with another SGX instruction; otherwise cleared.

**Protected Mode Exceptions**

#GP(0) If a memory operand effective address is outside the DS segment limit.
If DS segment is unusable.
If a memory operand is not properly aligned.
RBX does not refer to an enclave page associated with SECS referenced in RCX.

#PF(error code) If a page fault occurs in accessing memory operands.
If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
If RCX does not refer to an SECS page.

**64-Bit Mode Exceptions**

#GP(0) If a memory address is in a non-canonical form.
If a memory operand is not properly aligned.
RBX does not refer to an enclave page associated with SECS referenced in RCX.

#PF(error code) If a page fault occurs in accessing memory operands.
If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
If RCX does not refer to an SECS page.
ESETCONTEXT—Set the ENCLAVECONTEXT Field in SECS

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>EAX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>ESETCONTEXT (In)</td>
<td>Return error code (Out)</td>
<td>Address of the destination EPC page (In, EA)</td>
</tr>
</tbody>
</table>

### Description

The ESETCONTEXT leaf overwrites the ENCLAVECONTEXT field in the SECS. ECREATE and ELD of an SECS set the ENCLAVECONTEXT field in the SECS to the address of the SECS (for access later in ERDINFO). The ESETCONTEXT instruction allows a VMM to overwrite the default context value if necessary, for example, if the VMM is emulating ECREATE or ELD on behalf of the guest.

The content of RCX is an effective address of the SECS page to be updated, RDX contains the address pointing to the value to be stored in the SECS. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if:
- The operand is not properly aligned.
- RCX does not refer to an SECS page.

### ESETCONTEXT Memory Parameter Semantics

<table>
<thead>
<tr>
<th>EPCPAGE</th>
<th>CONTEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read access permitted by Enclave</td>
<td>Read/Write access permitted by Non Enclave</td>
</tr>
</tbody>
</table>

The instruction faults if any of the following:

### ESETCONTEXT Faulting Conditions

- A memory operand effective address is outside the DS segment limit (32b mode).
- A memory operand is not properly aligned.
- DS segment is unusable (32b mode).
- A page fault occurs in accessing memory operands.
- A memory address is in a non-canonical form (64b mode).

### Concurrency Restrictions

Table 38-80. Base Concurrency Restrictions of ESETCONTEXT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Base Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESETCONTEXT</td>
<td>SECS [DS:RCX]</td>
<td>Shared SGX_EPC_PAGE_CONFLICT</td>
</tr>
</tbody>
</table>
**Operation**

### Temp Variables in ESETCONTEXT Operational Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP_SECS</td>
<td>Physical Address</td>
<td>64</td>
<td>Physical address of the SECS of the page being modified.</td>
</tr>
<tr>
<td>TMP_CONTEXT</td>
<td>CONTEXT</td>
<td>64</td>
<td>Data Value of CONTEXT.</td>
</tr>
</tbody>
</table>

### ESETCONTEXT Return Value in RAX

<table>
<thead>
<tr>
<th>Error</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Error</td>
<td>0</td>
<td>ESETCONTEXT Successful.</td>
</tr>
<tr>
<td>SGX_EPC_PAGE_CONFLICT</td>
<td></td>
<td>Failure due to concurrent operation of another SGX instruction.</td>
</tr>
</tbody>
</table>

(*) check alignment of the EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); Fl;

(*) check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); Fl;

(*) check alignment of the CONTEXT field (RDX) *)
IF (DS:RDX is not 8Byte Aligned) THEN
    #GP(0); Fl;

(*) Load CONTEXT into local variable *)
TMP_CONTEXT := DS:RDX

(*) Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF := 1;
    RFLAGS.CF := 0;
    RAX := SGX_EPC_PAGE_CONFLICT;
    goto DONE;
    Fl;

(*) check page validity *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    #PF(DS:RCX, PFEC.SGX);
    Fl;

(*) check EPC page is an SECS page *)

Table 38-81. Additional Concurrency Restrictions of ESETCONTEXT

<table>
<thead>
<tr>
<th>Leaf</th>
<th>Parameter</th>
<th>Additional Concurrency Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT</td>
<td>Access</td>
</tr>
<tr>
<td></td>
<td>vs. EADD, EEXTEND, EINIT</td>
<td>Access</td>
</tr>
<tr>
<td></td>
<td>vs. ETRACK, ETRACKC</td>
<td>Access</td>
</tr>
<tr>
<td>ESETCONTEXT</td>
<td>SECS [DS:RCX]</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;

(* load the context value into SECS(DS:RCX).ENCLAVECONTEXT *)
SECS(DS:RCX).ENCLAVECONTEXT := TMP_CONTEXT;

RAX := 0;
RFLAGS.ZF := 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF := 0;

Flags Affected
ZF is set if ESETCONTEXT fails due to concurrent operation with another SGX instruction; otherwise cleared.
CF, PF, AF, OF, and SF are cleared.

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the DS segment limit.
    If DS segment is unusable.
    If a memory operand is not properly aligned.
#PF(error code) If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions
#GP(0) If a memory address is in a non-canonical form.
    If a memory operand is not properly aligned.
#PF(error code) If a page fault occurs in accessing memory operands.
Intel® SGX provides Intel® Architecture with a collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel® 64 architectures. These Intel SGX instructions are designed to work with legacy software and the various IA32 and Intel 64 modes of operation.

### 39.1 INTEL® SGX AVAILABILITY IN VARIOUS PROCESSOR MODES

The Intel SGX extensions (see Table 34-1) are available only when the processor is executing in protected mode of operation. Additionally, the extensions are not available in System Management Mode (SMM) of operation or in Virtual 8086 (VM86) mode of operation. Finally, all leaf functions of ENCLU and ENCLS require CR0.PG enabled. The exact details of exceptions resulting from illegal modes and their priority are listed in the reference pages of ENCLS and ENCLU.

### 39.2 IA32_FEATURE_CONTROL

IA32_FEATURE_CONTROL MSR provides two new bits related to two aspects of Intel SGX: using the instruction extensions and launch control configuration.

#### 39.2.1 Availability of Intel SGX

IA32_FEATURE_CONTROL[bit 18] allows BIOS to control the availability of Intel SGX extensions. For Intel SGX extensions to be available on a logical processor, bit 18 in the IA32_FEATURE_CONTROL MSR on that logical processor must be set, and IA32_FEATURE_CONTROL MSR on that logical processor must be locked (bit 0 must be set). See Section 34.7.1 for additional details. OS is expected to examine the value of bit 18 prior to enabling Intel SGX on the thread, as the settings of bit 18 is not reflected by CPUID.

#### 39.2.2 Intel SGX Launch Control Configuration

The IA32_SGXLEPUBKEYHASHn MSRs used to configure authorized launch enclaves' MRSIGNER digest value. They are present on logical processors that support the collection of SGX1 leaf functions (i.e., CPUID.(EAX=12H, ECX=00H):EAX[0] = 1) and that CPUID.(EAX=07H, ECX=00H):ECX[30] = 1. IA32_FEATURE_CONTROL[bit 17] allows BIOS to enable write access to these MSRs. If IA32_FEATURE_CONTROL.LE_WR (bit 17) is set to 1 and IA32_FEATURE_CONTROL is locked on that logical processor, IA32_SGXLEPUBKEYHASH MSRs on that logical processor are writeable. If this bit 17 is not set or IA32_FEATURE_CONTROL is not locked, IA32_SGXLEPUBKEYHASH MSRs are read only. See Section 36.1.4 for additional details.

### 39.3 INTERACTIONS WITH SEGMENTATION

#### 39.3.1 Scope of Interaction

Intel SGX extensions are available only when the processor is executing in a protected mode operation (see Section 39.1 for Intel SGX availability in various processor modes). Enclaves abide by all the segmentation policies set up by the OS, but they can be more restrictive than the OS.

Intel SGX interacts with segmentation at two levels:

- The Intel SGX instruction (see the enclave instruction in Table 34-1).
• While executing inside an enclave (legacy instructions and enclave instructions permitted inside an enclave).

### 39.3.2 Interactions of Intel® SGX Instructions with Segment, Operand, and Addressing Prefixes

All the memory operands used by the Intel SGX instructions are interpreted as offsets within the data segment (DS). The segment-override prefix on Intel SGX instructions is ignored.

Operand size is fixed for each enclave instruction. The operand-size prefix is reserved, and results in a #UD exception if used.

All address sizes are determined by the operating mode of the processor. The address-size prefix is ignored. This implies that while operating in 64-bit mode of operation, the address size is always 64 bits, and while operating in 32-bit mode of operation, the address size is always 32 bits. Additionally, when operating in 16-bit addressing, memory operands used by enclave instructions use 32 bit addressing; the value of CS.D is ignored.

### 39.3.3 Interaction of Intel® SGX Instructions with Segmentation

All leaf functions of ENCLU and ENCLS instructions require that the DS segment be usable, and be an expand-up segment. Failing this check results in generation of a #GP(0) exception.

The Intel SGX leaf functions used for entering the enclave (ENCLU[EENTER] and ENCLU[ERESUME]) operate as follows:

• All usable segment registers except for FS and GS have a zero base.
• The contents of the FS/GS segment registers (including the hidden portion) is saved in the processor.
• New FS and GS values compatible with enclave security are loaded from the TCS.
• The linear ranges and access rights available under the newly-loaded FS and GS must abide to OS policies by ensuring they are subsets of the linear-address range and access rights available for the DS segment.
• The CS segment mode (64-bit, compatible, or 32 bit modes) must be consistent with the segment mode for which the enclave was created, as indicated by the SECS.ATTRIBUTES.MODE64 bit, and that the CPL of the logical processor is 3.

An exit from the enclave either via ENCLU[EEXIT] or via an AEX restores the saved values of FS/GS segment registers.

### 39.3.4 Interactions of Enclave Execution with Segmentation

During the course of execution, enclave code abides by all segmentation policies as dictated by IA32 and Intel 64 Architectures, and generates appropriate exceptions on violations.

Additionally, any attempt by software executing inside an enclave to modify the processor’s segmentation state (e.g., via MOV seg register, POP seg register, LDS, far jump, etc; excluding WRFSBASE/WRGSBASE) results in the generation of a #UD. See Section 36.6.1 for more information.

Upon enclave entry via the EENTER leaf function, FS is loaded from the (TCS.OFSBASE + SECS.BASEADDR) and TCS.FSLIMIT fields and GS is loaded from the (TCS.OGSBASE + SECS.BASEADDR) and TCS.GSLIMIT fields.

Execution of WRFSBASE and WRGSBASE from inside a 64-bit enclave is allowed. The processor will save the new values into the current SSA frame on an asynchronous exit (AEX) and restore them back on enclave entry via ENCLU[ERESUME] instruction.

### 39.4 INTERACTIONS WITH PAGING

Intel SGX instructions are available only when the processor is executing in a protected mode of operation. Additionally, all Intel SGX leaf functions except for EDBGRD and EDBGWR are available only if paging is enabled. Any attempt to execute these leaf functions with paging disabled results in an invalid-opcode exception (#UD). As with
segmentation, enclaves abide by all the paging policies set up by the OS, but they can be more restrictive than the OS.

All the memory operands passed into Intel SGX instructions are interpreted as offsets within the DS segment, and the linear addresses generated by combining these offsets with DS segment register are subject to paging-based access control if paging is enabled at the time of the execution of the leaf function.

Since the ENCLU[EENTER] and ENCLU[ERESUME] can only be executed when paging is enabled, and since paging cannot be disabled by software running inside an enclave (recall that enclaves always run with CPL = 3), enclave execution is always subject to paging-based access control. The Intel SGX access control itself is implemented as an extension to the existing paging modes. See Section 35.5 for details.

Execution of Intel SGX instructions may set accessed and dirty flags on accesses to EPC pages that do not fault even if the instruction later causes a fault for some other reason.

39.5 INTERACTIONS WITH VMX

Intel SGX functionality (including SGX1 and SGX2) can be made available to software running in either VMX root operation or VMX non-root operation, as long as the processor is using a legal mode of operation (see Section 39.1).

A VMM has the flexibility to configure a VMCS to permit a guest to use any subset of the ENCLS leaf functions. Availability of the ENCLU leaf functions in VMX non-root operation has the same requirement as ENCLU leaf functions outside of a virtualized environment.

Details of the VMCS control to allow VMM to configure support of Intel SGX in VMX non-root operation is described in Section 39.5.1

39.5.1 VMM Controls to Configure Guest Support of Intel® SGX

Intel SGX capabilities are primarily exposed to the software via the CPUID instruction. VMMs can virtualize CPUID instruction to expose/hide this capability to/from guests.

Some of Intel SGX resources are exposed/controlled via model-specific registers (see Section 34.7). VMMs can virtualize these MSRs for the guests using the MSR bitmaps referenced by pointers in the VMCS.

The VMM can partition the Enclave Page Cache, and assign various partitions to (a subset of) its guests via the usual memory-virtualization techniques such as paging or the extended page table mechanism (EPT).

The VMM can set the “enable ENCLS exiting” VM-execution controls to cause a VM exit when the ENCLS instruction is executed in VMX non-root operation. If the “enable ENCLS exiting” control is 0, all of the ENCLS leaf functions are permitted in VMX non-root operation. If the “enable ENCLS exiting” control is 1, execution of ENCLS leaf functions in VMX non-root operation is governed by consulting the bits in a new 64-bit VM-execution control field called the ENCLS-exiting bitmap (Each bit in the bitmap corresponds to an ENCLS leaf function with an EAX value that is identical to the bit’s position). When bits in the “ENCLS-exiting bitmap” are set, attempts to execute the corresponding ENCLS leaf functions in VMX non-root operation causes VM exits. The checking for these VM exits occurs immediately after checking that CPL = 0.

39.5.2 Interactions with the Extended Page Table Mechanism (EPT)

Intel SGX instructions are fully compatible with the extended page-table mechanism (EPT; see Section 29.3).

All the memory operands passed into Intel SGX instructions are interpreted as offsets within the DS segment, and the linear addresses generated by combining these offsets with DS segment register are subject to paging and EPT. As with paging, enclaves abide by all the policies set up by the VMM.

The Intel SGX access control itself is implemented as an extension to paging and EPT, and may be more restrictive. See Section 39.4 for details of this extension.

An execution of an Intel SGX instruction may set accessed and dirty flags for EPT (when enabled; see Section 29.3.5) on accesses to EPC pages that do not fault or cause VM exits even if the instruction later causes a fault or VM exit for some other reason.
### 39.5.3 Interactions with APIC Virtualization

This section applies to Intel SGX in VMX non-root operation when the "virtualize APIC accesses" VM-execution control is 1.

A memory access by an enclave instruction that implicitly uses a cached physical address is never checked for overlap with the APIC-access page. Such accesses never cause APIC-access VM exits and are never redirected to the virtual-APIC page. Implicit memory accesses can only be made to the SECS, the TCS, or the SSA of an enclave (see Section 35.5.3.2).

An explicit Enclave Access (a linear memory access which is either from within an enclave into its ELRANGE, or an access by an Intel SGX instruction that is expected to be in the EPC) that overlaps with the APIC-access page causes a #PF exception (APIC page is expected to be outside of EPC).

Non-Enclave accesses made either by an Intel SGX instruction or by a logical processor inside an enclave to an address that without SGX would have caused redirection to the virtual-APIC page instead cause an APIC-access VM exit.

Other than implicit accesses made by Intel SGX instructions, guest-physical and physical accesses are not considered "enclave accesses"; consequently, such accesses result in undefined behavior if these accesses eventually reach EPC. This applies to any non-enclave physical accesses.

While a logical processor is executing inside an enclave, an attempt to execute an instruction outside of ELRANGE results in a #GP(0), even if the linear address would translate to a physical address that overlaps the APIC-access page.

### 39.5.4 Interactions with VT and SGX concurrency

In some cases, a VMM is required to handle conflicts between its own operation and a guest operation on EPC pages that are present in both guest and VMM address space. These conflict would otherwise cause the guest to experience an unexpected behavior (vs. running directly on the h/w). These conflict cases are:

- ETRACK/ETRACKC failure due to Entry Epoch Object Lock conflict or reference tracking check failure.
- EPC Page Resource conflict.

A new exit reason is defined for all those cases: SGX_CONFLICT (value 71). The VMCS exit qualification field details the specific case as follows:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Size (bits)</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:0</td>
<td>16</td>
<td>Code</td>
<td>Exit qualification code. The following values are defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: TRACKINGRESOURCECONFLICT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: TRACKINGREFERENCECONFLICT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2: EPC_PAGECONFLICTEXCEPTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3: EPC_PAGECONFLICTERROR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other: Reserved</td>
</tr>
<tr>
<td>31:16</td>
<td>16</td>
<td>Error</td>
<td>Error code. Applicable only if the exit qualification code is EPC_PAGECONFLICTERROR; contains the error code that would be returned in RAX if the instruction was executed on bare metal platform or if the ENABLE_EPCVIRTUALIZATION_EXTENSIONS bit in the secondary processor control field is not set. In other cases this field is reserved as 0.</td>
</tr>
<tr>
<td>63:32</td>
<td>32</td>
<td>Reserved</td>
<td>Always 0.</td>
</tr>
</tbody>
</table>

This SGX_CONFLICT exiting behavior is controlled by a VM execution control called ENABLE_EPC_VIRTUALIZATION_EXTENSIONS (bit 29 of the secondary processor control field).

Details for various SGX_CONFLICT VMEXIT cases are provided in the following sections.
39.5.5 Virtual Child Tracking

SGX oversubscription support adds the ability to associate virtual children with each enclave using the ENCLV[EINCVIRTCHILD] and ENCLV[EDECVIRTCHILD] instructions. The VMM enables checking of the virtual child count by EREMOVE and EWB in guests with a new VM execution control called ENABLE_EPC_VIRTUALIZATION_EXTENSIONS.

When in VMX non-root operation and the ENABLE_EPC_VIRTUALIZATION_EXTENSIONS control enabled, the following instructions change their behavior:

- EWB and EREMOVE return the SGX_CHILD_PRESENT error code if any virtual or physical children are associated with the enclave.
- ERDINFO set STATUS.CHILDPRESENT if any virtual or physical children are associated with the enclave.

39.5.6 Handling EPCM Entry Lock Conflicts

When performing paging within a VMM, it is possible for a contention on the EPC page to happen in the following case:

- The VMM performs an ELDB/ELDU/ELDBC/ELDUC of an enclave page, and the guest attempts to perform some SGX instruction (e.g., EREMOVE) where the same SECS parent page is required.

A similar conflict may occur if the VMM uses EINCVIRTCHILD or EDECVIRTCHILD pointing to an SECS page. In all other cases where a SGX instruction executed by the VMM the applicable EPC page should not be mapped to the guest, thus no resource conflict occurs.

This conflicting situation can cause the guest’s instruction to fail and cause guest instability. To help the VMM manage such conflicts, the SGX VMM paging extensions introduce a new VM-Exit that will be triggered whenever the guest encounters a resource conflict.

The exit reason is SGX_CONFLICT. The exit qualification field is used to distinguish the two kinds of resource conflicts:

- A value of EPCMRESOURCE_CONFLICT_EXCEPTION (2) in the exit qualification code field indicates that a resource conflict occurred that would result in a #GP. In that case, the exit qualification error field is set to zero.
- A value of EPC_PAGE_CONFLICT_ERROR (3) in the exit qualification code field indicates that a resource conflict occurred that would result in an error code being return in RAX. In that case, the exit qualification error field is set to SGX_EPC_PAGE_CONFLICT.

The Guest Linear Address and Guest Physical Address fields are set to the guest linear and guest physical addresses respectively of the EPC page on which the conflict occurred. The VMM may determine which instruction induced the exit by reading RAX. The exit also populates the VM-exit instruction length field.

The VMM can determine whether the conflict may be due to its own operation, e.g., by setting a per-enclave busy indicator before executing ELD*, and clearing it afterwards. In that case, the VMM can handle an SGX Conflict (EPCM_PAGE_CONFLICT_*) exit by resuming guest execution at the same instruction, allowing the guest to re-execute the instruction. The VMM may also take steps to throttle its own paging thread to reduce contention with the guest.

If the VMM determines that the conflict is not due to its own operation, it may inject a #GP (in case of EPC_PAGE_-_CONFLICT_EXCEPTION), or emulate an error code as the guest instruction would return (in case of EPC_PAGE_-_CONFLICT_ERROR) by setting ZF and copying the error value provided in the exit qualification to guest RAX.

To gracefully handle resource contention on the VMM side, the VMM should use the new ELDBC and ELDUC instructions. These are similar to ELDB and ELDU respectively, except that on EPC resource contention they return an SGX_EPC_PAGE_CONFLICT error instead of issuing a #GP. In case of an error, the VMM can retry the instruction, possibly throttling the guest to assure progress.

When using EDECVIRTCHILD and EINCVIRTCHILD, the VMM should preferably point to the enclave child page, not to the SECS page, avoiding resource conflict on the SECS. If the VMM chooses to point to the SECS page, it should handle conflicts in the same way as handling the ELD* case.
39.5.7  Context Tracking

The ENCLAVECONTEXT field in the SECS is available for use by the VMM to track context information associated with that enclave, such as the GPA of the SECS in the context of the appropriate guest. This field is initialized by the successful execution of ECREATE and ELD of an SECS page. The value stored in the ENCLAVECONTEXT field will be the translation of the target page address produced by paging (GPA in VMMs that have EPTs turned on). VMMs may override this default value by calling the ENCV[ESETCONTEXT] instruction, which allows the VMM to store an arbitrary 64-bit value in the ENCLAVECONTEXT field. The VMM may later access the ENCLAVECONTEXT field by calling ENCLS[ERDINFO] on any member page of the enclave, including the SECS.

For nested virtualization cases, the lowest level VMM can make SGX oversubscription instructions higher level guest VMMs. In that case the lower level VMM can simply inject #GP to higher level VMMs when attempting to execute these instructions.

However, if VMMs expose SGX oversubscription instructions to higher level VMMs, then VMMs have to use ENCV[ESETCONTEXT] instruction to properly manage the ENCLAVECONTEXT field of SECS during paging operations. That may involve emulating ECREATE, ELD, ESETCONTEXT, and ERDINFO instructions apart from managing ENCLAVECONTEXT values.

39.6  INTEL® SGX INTERACTIONS WITH ARCHITECTURALLY-VISIBLE EVENTS

All architecturally visible events (exceptions, interrupts, SMI, NMI, INIT, VM exit) can be detected while inside an enclave and will cause an asynchronous enclave exit if they are not blocked. Additionally, INT3, and the SignalTXTMsg[SENTER] (i.e., GETSEC[SENTER]’s rendezvous event message) events also cause asynchronous enclave exits. Note that SignalTXTMsg[SEXIT] (i.e., GETSEC[SEXIT]’s teardown message) does not cause an AEX.

On an AEX, information about the event causing the AEX is stored in the SSA (see Section 37.4 for details of AEX). The information stored in the SSA only describes the first event that triggered the AEX. If parsing/delivery of the first event results in detection of further events (e.g., VM exit, double fault, etc.), then the event information in the SSA is not updated to reflect these subsequently detected events.

39.7  INTERACTIONS WITH THE PROCESSOR EXTENDED STATE AND MISCELLANEOUS STATE

39.7.1  Requirements and Architecture Overview

Processor extended states are the ISA features that are enabled by the settings of CR4.OSXSAVE and the XCR0 register. Processor extended states are normally saved/restored by software via XSAVE/XRSTOR instructions. Details of discovery of processor extended states and management of these states are described in Chapter 13 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Additionally, the following requirements apply to Intel SGX:

• On an AEX, the Intel SGX architecture must protect the processor extended state and miscellaneous state by saving them in the enclave’s state-save area (SSA), and clear the secrets from the processor extended state that is used by an enclave.

• Intel SGX architecture must verify that the SSA frame size is large enough to contain all the processor extended states and miscellaneous state used by the enclave.

• Intel SGX architecture must ensure that enclaves can only use processor extended state that is enabled by system software in XCR0.

• Enclave software should be able to discover only those processor extended state and miscellaneous state for which such protection is enabled.

• The processor extended states that are enabled inside the enclave must be approved by the enclave developer:
  — Certain processor extended state (e.g., Memory Protection Extensions, see Appendix E, “Intel® Memory Protection Extensions,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1)
modify the behavior of the legacy ISA software. If such features are enabled for enclaves that do not
understand those features, then such a configuration could lead to a compromise of the enclave's security.

- The processor extended states that are enabled inside the enclave must form an integral part of the enclave's
  identity. This requirement has two implications:
  - Service providers may decide to assign different trust level to the same enclave depending on the ISA
    features the enclave is using.

To meet these requirements, the Intel SGX architecture defines a sub-field called X-Feature Request Mask (XFRM)
in the ATTRIBUTES field of the SECS. On enclave creation (ENCLS[ECREATE] leaf function), the required SSA frame
size is calculated by the processor from the list of enabled extended and miscellaneous states and verified against
the actual SSA frame size defined by SECS.SSAFRAMESIZE.

On enclave entry, after verifying that XFRM is only enabling features that are already enabled in XCR0, the value in
the XCR0 is saved internally by the processor, and is replaced by the XFRM. On enclave exit, the original value of
XCR0 is restored. Consequently, while inside the enclave, the processor extended states enabled in XFRM are in
enabled state, and those that are disabled in XFRM are in disabled state.

The entire ATTRIBUTES field, including the XFRM subfield is integral part of enclave's identity (i.e., its value is
included in reports generated by ENCLU[EREPORT], and select bits from this field can be included in key-derivation
for keys obtained via the ENCLU[EGETKEY] leaf function).

Enclave developers can create their enclave to work with certain features and fallback to another code path in case
those features aren’t available (e.g., optimize for AVX and fallback to SSE). For this purpose Intel SGX provides the
following fields in SIGSTRUCT: ATTRIBUTES, ATTRIBUTESMASK, MISCSELECT, and MISCMASK. EINIT ensures that
the final SECS.ATTRIBUTES and SECS.MISCSELECT comply with the enclave developer's requirements as follows:

SIGSTRUCT.ATTRIBUTES & SIGSTRUCT.ATTRIBUTEMASK = SECS.ATTRIBUTES & SIGSTRUCT.ATTRIBUTEMASK
SIGSTRUCT.MISCSELECT & SIGSTRUCT.MISCMASK = SECS.MISCMASK & SIGSTRUCT.MISCMASK.

On an asynchronous enclave exit, the processor extended states enabled by XFRM are saved in the current SSA
frame, and overwritten by synthetic state (see Section 37.3 for the definition of the synthetic state). When the
interrupted enclave is resumed via the ENCLU[ERESUME] leaf function, the saved state for processor extended
states enabled by XFRM is restored.

### Relevant Fields in Various Data Structures

#### 39.7.2.1 SECS.ATTRIBUTES.XFRM

The ATTRIBUTES field of the SECS data structure (see Section 35.7) contains a sub-field called XSAVE-Feature
Request Mask (XFRM). Software populates this field at the time of enclave creation according to the features that
are enabled by the operating system and approved by the enclave developer.

Intel SGX architecture guarantees that during enclave execution, the processor extended state configuration of the
processor is identical to what is required by the XFRM sub-field. All the processor extended states enabled in XFRM
are saved on AEX from the enclave and restored on ERESUME.

The XFRM sub-field has the same layout as XCR0, and has consistency requirements that are similar to those for
XCR0. Specifically, the consistency requirements on XFRM values depend on the processor implementation and the
set of features enabled in CR4.

Legal values for SECS.ATTRIBUTES.XFRM conform to these requirements:

- XFRM[1:0] must be set to 0x3.
- If the processor does not support XSAVE, or if the system software has not enabled XSAVE, then XFRM[63:2]
  must be zero.
- If the processor does support XSAVE, XFRM must contain a value that would be legal if loaded into XCR0.

The various consistency requirements are enforced at different times in the enclave's life cycle, and the exact
enforcement mechanisms are elaborated in Section 39.7.3 through Section 39.7.6.

On processors not supporting XSAVE, software should initialize XFRM to 0x3. On processors supporting XSAVE,
software should initialize XFRM to be a subset of XCR0 that would be present at the time of enclave execution.
Because bits 0 and 1 of XFRM must always be set, the use of Intel SGX requires that SSE be enabled (CR4.OSFXSR = 1).

### 39.7.2.2 SECS.SSAFRAMESIZE

The SSAFRAMESIZE field in the SECS data structure specifies the number of pages which software allocated\(^\d\) for each SSA frame, including both the GPRSGX area, MISC area, the XSAVE area (x87 and XMM states are stored in the latter area), and optionally padding between the MISC and XSAVE area. The GPRSGX area must hold all the general-purpose registers and additional Intel SGX specific information. The MISC area must hold the Miscellaneous state as specified by SECS.MISCSELECT, the XSAVE area holds the set of processor extended states specified by SECS.ATTRIBUTES.XFRM (see Section 35.9 for the layout of SSA and Section 39.7.3 for ECREATE’s consistency checks). The SSA is always in non-compacted format.

If the processor does not support XSAVE, the XSAVE area will always be 576 bytes; a copy of XFRM (which will be set to 0x3) is saved at offset 512 on an AEX.

If the processor does support XSAVE, the length of the XSAVE area depends on SECS.ATTRIBUTES.XFRM. The length would be equal to what CPUID.(EAX=0DH, ECX=0):EBX would return if XCR0 were set to XFRM. The following pseudo code illustrates how software can calculate this length using XFRM as the input parameter without modifying XCR0:

\[
\begin{align*}
&\text{offset} = 576; \\
&\text{size}_\text{last}_x = 0; \\
&\text{For } x = 2 \text{ to } 63 \\
&\quad \text{IF } (\text{XFRM}[x] \neq 0) \text{ Then} \\
&\quad \quad \text{tmp}_\text{offset} = \text{CPUID}(\text{EAX}=0DH, \text{ECX}=x) : \text{EBX}[31:0]; \\
&\quad \quad \text{IF } (\text{tmp}_\text{offset} \geq \text{offset} + \text{size}_\text{last}_x) \text{ Then} \\
&\quad \quad \quad \text{offset} = \text{tmp}_\text{offset}; \\
&\quad \quad \quad \text{size}_\text{last}_x = \text{CPUID}(\text{EAX}=0DH, \text{ECX}=x) : \text{EAX}[31:0]; \\
&\quad \text{FI}; \\
&\text{FI}; \\
&\text{EndFor} \\
&\text{return } (\text{offset} + \text{size}_\text{last}_x); (* \text{compute_xsave_size(XFRM)}, \text{see "ECREATE—Create an SECS page in the Enclave Page Cache"}*)
\end{align*}
\]

Where the non-zero bits in XFRM are a subset of non-zero bit fields in XCR0.

The size of the MISC region depends on the setting of SECS.MISCSELECT and can be calculated using the layout information described in Section 35.9.2

### 39.7.2.3 XSAVE Area in SSA

The XSAVE area of an SSA frame begins at offset 0 of the frame.

### 39.7.2.4 MISC Area in SSA

The MISC area of an SSA frame is positioned immediately before the GPRSGX region.

### 39.7.2.5 SIGSTRUCT Fields

Intel SGX provides the flexibility for an enclave developer to choose the enclave’s code path according to the features that are enabled on the platform (e.g., optimize for AVX and fallback to SSE). See Section 39.7.1 for details.

---

1. It is the responsibility of the enclave to actually allocate this memory.
SIGSTRUCT includes the following fields:
SIGSTRUCT.ATTRIBUTES, SIGSTRUCT.ATTRIBUTEMASK, SIGSTRUCT.MISCSELECT, SIGSTRUCT.MISCMASK.

39.7.2.6 REPORT.ATTRIBUTES.XFRM and REPORT.MISCSELECT
The processor extended states and miscellaneous states that are enabled inside the enclave form an integral part of the enclave's identity and are therefore included in the enclave's report, as provided by the ENCLU[EREPORT] leaf function. The REPORT structure includes the enclave's XFRM and MISCESELECT configurations.

39.7.2.7 KEYREQUEST
An enclave developer can specify which bits out of XFRM and MISCESELECT ENCLU[EGETKEY] should include in the derivation of the sealing key by specifying ATTRIBUTEMASK and MISCMASK in the KEYREQUEST structure.

39.7.3 Processor Extended States and ENCLS[ECREATE]
The ECREATE leaf function of the ENCLS instruction enforces a number of consistency checks described earlier. The execution of ENCLS[ECREATE] leaf function results in a #GP(0) in any of the following cases:
• SECS.ATTRIBUTES.XFRM[1:0] is not 3.
• The processor does not support XSAVE and any of the following is true:
  — SECS.ATTRIBUTES.XFRM[63:2] is not 0.
  — SECS.SSAFRAMESIZE is 0.
• The processor supports XSAVE and any of the following is true:
  — XSETBV would fault on an attempt to load XFRM into XCR0.
  — XFRM[63]=1.
  — The SSAFRAME is too small to hold required, enabled states (see Section 39.7.2.2).

39.7.4 Processor Extended States and ENCLU[EENTER]

39.7.4.1 Fault Checking
The EENTER leaf function of the ENCLU instruction enforces a number of consistency requirements described earlier. The execution of the ENCLU[EENTER] leaf function results in a #GP(0) in any of the following cases:
• If CR4.OSFXSR=0.
• If the processor supports XSAVE and either of the following is true:
  — CR4.OSXSAVE=0 and SECS.ATTRIBUTES.XFRM is not 3.
  — (SECS.ATTRIBUTES.XFRM & XCR0) != SECS.ATTRIBUTES.XFRM

39.7.4.2 State Loading
If ENCLU[EENTER] is successful, the current value of XCR0 is saved internally by the processor and replaced by SECS.ATTRIBUTES.XFRM.
39.7.5 Processor Extended States and AEX

39.7.5.1 State Saving

On an AEX, processor extended states are saved into the XSAVE area of the SSA frame in a compatible format with XSAVE that was executed with EDX:EAX = SECS.ATTRIBUTES.XFRM, with the memory operand being the XSAVE area, and (for 64-bit enclaves) as if REX.W=1. The XSTATE_BV part of the XSAVE header is saved with 0 for every bit that is 0 in XFRM. Other bits may be saved as 0 if the state saved is initialized.

Note that enclave entry ensures that if CR4.OSXSAVE is set to 0, then SECS.ATTRIBUTES.XFRM is set to 3. It should also be noted that it is not possible to enter an enclave with FXSAVE disabled.

39.7.5.2 State Synthesis

After saving the extended state, the processor restores XCR0 to the value it held at the time of the most recent enclave entry.

The state of features corresponding to bits set in XFRM is synthesized. In general, these states are initialized. Details of state synthesis on AEX are documented in Section 37.3.1.

39.7.6 Processor Extended States and ENCLU[ERESUME]

39.7.6.1 Fault Checking

The ERESUME leaf function of the ENCLU instruction enforces a number of consistency requirements described earlier. Specifically, the ENCLU[ERESUME] leaf function results in a #GP(0) in any of the following cases:

- CR4.OSFXSR=0.
- The processor supports XSAVE and either of the following is true:
  - CR4.OSXSAVE=0 and SECS.ATTRIBUTES.XFRM is not 3.
  - (SECS.ATTRIBUTES.XFRM & XCR0) != SECS.ATTRIBUTES.XFRM.

A successful execution of ENCLU[ERESUME] loads state from the XSAVE area of the SSA frame in a fashion similar to that used by the XRSTOR instruction. Data in the XSAVE area that would cause the XRSTOR instruction to fault will cause the ENCLU[ERESUME] leaf function to fault. Examples include, but are not restricted to the following:

- A bit is set in the XSTATE_BV field and clear in XFRM.
- The required bytes in the header are not clear.
- Loading data would set a reserved bit in MXCSR.

Any of these conditions will cause ERESUME to fault, even if CR4.OSXSAVE=0.

39.7.6.2 State Loading

If ENCLU[ERESUME] is successful, the current value of XCR0 is saved internally by the processor and replaced by SECS.ATTRIBUTES.XFRM.

State is loaded from the XSAVE area of the SSA frame as if the XRSTOR instruction were executed with XCR0=XFRM, EDX:EAX = XFRM, with the memory operand being the XSAVE area, and (for 64-bit enclaves) as if REX.W=1.

ENCLU[ERESUME] ensures that a subsequent execution of XSAVEOPT inside the enclave will operate properly (e.g., by marking all state as modified).

39.7.7 Processor Extended States and ENCLU[EEXIT]

The ENCLU[EEXIT] leaf function does not perform any X-feature specific consistency checks, nor performs any state synthesis. It is the responsibility of enclave software to clear any sensitive data from the registers before
executing EEXIT. However, successful execution of the ENCLU[EEXIT] leaf function restores XCR0 to the value it held at the time of the most recent enclave entry.

### 39.7.8 Processor Extended States and ENCLU[EREPORT]

The ENCLU[EREPORT] leaf function creates the MAC-protected REPORT structure that reports on the enclave’s identity. ENCLU[EREPORT] includes in the report the values of SECS.ATTRIBUTES.XFRM and SECS.MISCSELECT.

### 39.7.9 Processor Extended States and ENCLU[EGETKEY]

The ENCLU[EGETKEY] leaf function returns a cryptographic key based on the information provided by the KEYREQUEST structure. Intel SGX provides the means for isolation between different operating conditions by allowing an enclave developer to select which bits out of XFRM and MISCSELECT need to be included in the derivation of the keys.

### 39.8 INTERACTIONS WITH SMM

#### 39.8.1 Availability of Intel® SGX instructions in SMM

Enclave instructions are not available in SMM, and any attempt to execute ENCLS or ENCLU instructions inside SMM results in an invalid-opcode exception (#UD).

#### 39.8.2 SMI while Inside an Enclave

If the logical processor executing inside an enclave receives an SMI, the logical processor exits the enclave asynchronously. The response to an SMI received while executing inside an enclave depends on whether the dual-monitor treatment is enabled. For detailed discussion of transfer to SMM, see Chapter 32, “System Management Mode,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

If the logical processor executing inside an enclave receives an SMI when dual-monitor treatment is not enabled, the logical processor exits the enclave asynchronously, and transfers the control to the SMM handler. In addition to saving the synthetic architectural state to the SMRAM State Save Map (SSM), the logical processor also sets the “Enclave Interruption” bit in the SMRAM SSM (bit position 1 in SMRAM field at offset 7EE0H).

If the logical processor executing inside an enclave receives an SMI when dual-monitor treatment is enabled, the logical processor exits the enclave asynchronously, and transfers the control to the SMM monitor via SMM VM exit. The SMM VM exit sets the “Enclave Interruption” bit in the Exit Reason (see Table 39-2) and in the Guest Interruptibility State field (see Table 39-3) of the SMM VMCS.

#### 39.8.3 SMRAM Synthetic State of AEX Triggered by SMI

All processor registers saved in the SMRAM have the same synthetic values listed in Section 37.3. Additional SMRAM fields that are treated specially on SMI are:

<table>
<thead>
<tr>
<th>Position</th>
<th>Field</th>
<th>Value</th>
<th>Writable</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMRAM Offset 07EE0H.Bit 1</td>
<td>ENCLAVE_INTERRUPT</td>
<td>Set to 1 if exit occurred in enclave mode</td>
<td>No</td>
</tr>
</tbody>
</table>
39.9 INTERACTIONS OF INIT, SIPI, AND WAIT-FOR-SIPI WITH INTEL® SGX

INIT received inside an enclave, while the logical processor is not in VMX operation, causes the logical processor to exit the enclave asynchronously. After the AEX, the processor's architectural state is initialized to "Power-on" state (Table 9.1 in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). If the logical processor is BSP, then it proceeds to execute the BIOS initialization code. If the logical processor is an AP, it enters wait-for-SIPI state.

INIT received inside an enclave, while the logical processor (LP) is in VMX root operation, follows regular Intel Architecture behavior and is blocked.

INIT received inside an enclave, while the logical processor is in VMX non-root operation, causes an AEX. Subsequent to the AEX, the INIT causes a VM exit with the Enclave Interruption bit in the exit-reason field in the VMCS. A processor cannot be inside an enclave in the wait-for-SIPI state. Consequently, a SIPI received while inside an enclave is lost.

Intel SGX does not change the behavior of the processor in the wait-for-SIPI state.

The SGX-related processor states after INIT-SIPI-SIPI is as follows:
- EPC Settings: Unchanged
- EPCM: Unchanged
- CPUID.LEAF_12H.*: Unchanged
- ENCLAVE_MODE: 0 (LP exits enclave asynchronously)
- MEE state: Unchanged

Software should be aware that following INIT-SIPI-SIPI, the EPC might contain valid pages and should take appropriate measures such as initialize the EPC with the EREMOVE leaf function.

39.10 INTERACTIONS WITH DMA

DMA is not allowed to access any Processor Reserved Memory.

39.11 INTERACTIONS WITH TXT

39.11.1 Enclaves Created Prior to Execution of GETSEC

Enclaves which have been created before the GETSEC[SENTER] leaf function are available for execution after the successful completion of GETSEC[SENTER] and the corresponding SINIT ACM. Actions that a TXT Launched Environment performs in preparation to execute code in the Launched Environment, also applies to enclave code that would run after GETSEC[SENTER].

39.11.2 Interaction of GETSEC with Intel® SGX

All leaf functions of the GETSEC instruction are illegal inside an enclave, and results in an invalid-opcode exception (#UD).

Responding Logical Processors (RLP) which are executing inside an enclave at the time a GETSEC[SENTER] event occurs perform an AEX from the enclave and then enter the Wait-for-SIPI state.

RLP executing inside an enclave at the time of GETSEC[SEXIT], behave as defined for GETSEC[SEXIT]-that is, the RLPs pause during execution of SEXIT and resume after the completion of SEXIT.

The execution of a TXT launch does not affect Intel SGX configuration or security parameters.
### 39.11.3 Interactions with Authenticated Code Modules (ACMs)

Intel SGX only allows launching ACMs with an Intel SGX SVN that is at the same level or higher than the expected Intel SGX SVN. The expected Intel SGX SVN is specified by BIOS and locked down by the processor on the first successful execution of an Intel SGX instruction that doesn’t return an error code. Intel SGX provides interfaces for system software to discover whether a non-faulting Intel SGX instruction has been executed, and evaluate the suitability of the Intel SGX SVN value of any ACM that is expected to be launched by the OS or the VMM.

These interfaces are provided through a read-only MSR called the IA32_SGX_SVN_STATUS MSR (MSR address 500h). The IA32_SGX_SVN_STATUS MSR has the format shown in Table 39-3.

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Name</th>
<th>ACM Module ID</th>
<th>Value</th>
</tr>
</thead>
</table>
| 0            | Lock         | N.A.          | • If 1, indicates that a non-faulting Intel SGX instruction has been executed, consequently, launching a properly signed ACM but with Intel SGX SVN value less than the BIOS specified Intel SGX SVN threshold would lead to an TXT shutdown.  
• If 0, indicates that the processor will allow a properly signed ACM to launch irrespective of the Intel SGX SVN value of the ACM. |
| 15:1         | RSVD         | N.A.          | 0                                                                     |
| 23:16        | SGX_SVN_SINIT| SINIT ACM     | • If CPUID.01H:ECX.SMx =1, this field reflects the expected threshold of Intel SGX SVN for the SINIT ACM.  
• If CPUID.01H:ECX.SMx =0, this field is reserved (0). |
| 63:24        | RSVD         | N.A.          | 0                                                                     |

OS/VMM that wishes to launch an architectural ACM such as SINIT is expected to read the IA32_SGX_SVN_STATUS MSR to determine whether the ACM can be launched or a new ACM is needed:

• If either the Intel SGX SVN of the ACM is greater than the value reported by IA32_SGX_SVN_STATUS, or the lock bit in the IA32_SGX_SVN_STATUS is not set, then the OS/VMM can safely launch the ACM.

• If the Intel SGX SVN value reported in the corresponding component of the IA32_SGX_SVN_STATUS is greater than the Intel SGX SVN value in the ACM’s header, and if bit 0 of IA32_SGX_SVN_STATUS is 1, then the OS/VMM should not launch that version of the ACM. It should obtain an updated version of the ACM either from the BIOS or from an external resource.

However, OSVs/VMMs are strongly advised to update their version of the ACM any time they detect that the Intel SGX SVN of the ACM carried by the OS/VMM is lower than that reported by IA32_SGX_SVN_STATUS MSR, irrespective of the setting of the lock bit.

### 39.12 Interactions with Caching of Linear-Address Translations

Entering and exiting an enclave causes the logical processor to flush all the global linear-address context as well as the linear-address context associated with the current VPID and PCID. The MONITOR FSM is also cleared.

### 39.13 Interactions with Intel® Transactional Synchronization Extensions (Intel® TSX)

1. ENCLU or ENCLS instructions inside an HLE region will cause the flow to be aborted and restarted non-speculatively. ENCLU or ENCLS instructions inside an RTM region will cause the flow to be aborted and transfer control to the fallback handler.

2. If XBEGIN is executed inside an enclave, the processor does NOT check whether the address of the fallback handler is within the enclave.

3. If an RTM transaction is executing inside an enclave and there is an attempt to fetch an instruction outside the enclave, the transaction is aborted and control is transferred to the fallback handler. No #GP is delivered.
4. If an RTM transaction is executing inside an enclave and there is a data access to an address within the enclave that denied due to EPCM content (e.g., to a page belonging to a different enclave), the transaction is aborted and control is transferred to the fallback handler. No #GP is delivered.

5. If an RTM transaction executing inside an enclave aborts and the address of the fallback handler is outside the enclave, a #GP is delivered after the abort (EIP reported is that of the fallback handler).

### 39.13.1 HLE and RTM Debug

RTM debug will be suppressed on opt-out enclave entry. After opt-out entry, the logical processor will behave as if IA32_DEBUG_CTL[15]=0. Any #DB detected inside an RTM transaction region will just cause an abort with no exception delivered.

After opt-in entry, if either DR7[11] = 0 OR IA32_DEBUGCTL[15] = 0, any #DB or #BP detected inside an RTM transaction region will just cause an abort with no exception delivered.

After opt-in entry, if DR7[11] = 1 AND IA32_DEBUGCTL[15] = 1, any #DB or #BP detected inside an RTM transaction will
- terminate speculative execution,
- set RIP to the address of the XBEGIN instruction, and
- be delivered as #DB (implying an Intel SGX AEX; any #BP is converted to #DB).

- DR6[16] will be cleared, indicating RTM debug (if the #DB causes a VM exit, DR6 is not modified but bit 16 of the pending debug exceptions field in the VMCS will be set).

### 39.14 INTEL® SGX INTERACTIONS WITH S STATES

Whenever an Intel SGX enabled processor enters S3-S5 state, enclaves are destroyed. This is due to the EPC being destroyed when power down occurs. It is the application runtime’s responsibility to re-instantiate an enclave after a power transition for which the enclaves were destroyed.

### 39.15 INTEL® SGX INTERACTIONS WITH MACHINE CHECK ARCHITECTURE (MCA)

#### 39.15.1 Interactions with MCA Events

All architecturally visible machine check events (#MC and CMCI) that are detected while inside an enclave cause an asynchronous enclave exit.

Any machine check exception (#MC) that occurs after Intel SGX is first enables causes Intel SGX to be disabled, (CPUID.SGX_Leaf.0:EAX[SGX1] == 0). It cannot be enabled until after the next reset.

#### 39.15.2 Machine Check Enables (IA32_MCI_CTL)

All supported IA32_MCI_CTL bits for all the machine check banks must be set for Intel SGX to be available (CPUID.SGX_Leaf.0:EAX[SGX1] == 1). Any act of clearing bits from '1 to '0 in any of the IA32_MCI_CTL register may disable Intel SGX (set CPUID.SGX_Leaf.0:EAX[SGX1] to 0) until the next reset.

#### 39.15.3 CR4.MCE

CR4.MCE can be set or cleared with no interactions with Intel SGX.
39.16 INTEL® SGX INTERACTIONS WITH PROTECTED MODE VIRTUAL INTERRUPTS

ENCLS[ENTER] modifies neither EFLAGS.VIP nor EFLAGS.VIF.
ENCLS[ERESUME] loads EFLAGS in a manner similar to that of an execution of IRET with CPL = 3. This means that ERESUME modifies neither EFLAGS.VIP nor EFLAGS.VIF regardless of the value of the EFLAGS image in the SSA frame.
AEX saves EFLAGS.VIP and EFLAGS.VIF unmodified into the EFLAGS image in the SSA frame. AEX modifies neither EFLAGS.VIP nor EFLAGS.VIF after saving EFLAGS.
If CR4.PVI = 1, CPL = 3, EFLAGS.VM = 0, IOPL < 3, EFLAGS.VIP = 1, and EFLAGS.VIF = 0, execution of STI causes a #GP fault. In this case, STI modifies neither EFLAGS.IF nor EFLAGS.VIF. This behavior applies without change within an enclave (where CPL is always 3). Note that, if IOPL = 3, STI always sets EFLAGS.IF without fault; CR4.PVI, EFLAGS.VIP, and EFLAGS.VIF are neither consulted nor modified in this case.

39.17 INTEL SGX INTERACTION WITH PROTECTION KEYS

SGX interactions with PKRU are as follows:
- CPUID(EAX=12H, ECX=1):ECX.PKRU indicates whether SECS.ATTRIBUTES.XFRM.PKRU can be set. If SECS.ATTRIBUTES.XFRM.PKRU is set, then PKRU is saved and cleared as part of AEX and is restored as part of ERESUME. If CR4.PKE is set, an enclave can execute RDPKRU and WRKRU independent of whether SECS.ATTRIBUTES.XFRM.PKRU is set.
SGX interactions with domain permission checks are as follows:
1) If CR4.PKE is not set, then legacy and SGX permission checks are not effected.
2) If CR4.PKE is set, then domain permission checks are applied to all non-enclave access and enclave accesses to user pages in addition to legacy and SGX permission checks at a higher priority than SGX permission checks.
3) Implicit accesses aren't subject to domain permission checks.
Intel® SGX is architected to provide protection for production enclaves and permit enclave code developers to use an SGX-aware debugger to effectively debug a non-production enclave (debug enclave). Intel SGX also allows a non-SGX-aware debugger to debug non-enclave portions of the application without getting confused by enclave instructions.

40.1 CONFIGURATION AND CONTROLS

40.1.1 Debug Enclave vs. Production Enclave

The SECS of each enclave provides a bit, SECS.ATTRIBUTES.DEBUG, indicating whether the enclave is a debug enclave (if set) or a production enclave (if 0). If this bit is set, software outside the enclave can use EDBGRD/EDBGWR to access the EPC memory of the enclave. The value of DEBUG is not included in the measurement of the enclave and therefore doesn't require an alternate SIGSTRUCT to be generated to debug the enclave.

The ATTRIBUTES field in the SECS is reported in the enclave's attestation, and is included in the key derivation. Enclave secrets that were protected by the enclave using Intel SGX keys when it ran as a production enclave will not be accessible by the debug enclave. A debugger needs to be aware that special debug content might be required for a debug enclave to run in a meaningful way.

EPC memory belonging to a debug enclave can be accessed via the EDBGRD/EDBGWR leaf functions (see Section 38.4), while that belonging to a non-debug enclave cannot be accessed by these leaf functions.

40.1.2 Tool-Chain Opt-in

The TCS.FLAGS.DBGOPTIN bit controls interactions of certain debug and profiling features with enclaves, including code/data breakpoints, TF, RF, monitor trap flag, BTF, LBRs, BTM, BTS, Intel Processor Trace, and performance monitoring. This bit is forced to zero when EPC pages are added via EADD. A debugger can set this bit via EDBGWR to the TCS of a debug enclave.

An enclave entry through a TCS with the TCS.FLAGS.DBGOPTIN set to 0 is called an opt-out entry. Conversely, an enclave entry through a TCS with TCS.FLAGS.DBGOPTIN set to 1 is called an opt-in entry.

40.1.3 Debugging an Enclave That Uses Asynchronous Enclave Exit Notify

Whenever an opt-in enclave entry is used to perform enclave code debugging or profiling, the debugger or profiling tool may clear TCS.FLAGS.AEXNOTIFY to prevent AEX notifications from being delivered at each interrupt, breakpoint, or other exception.

40.2 SINGLE STEP DEBUG

40.2.1 Single Stepping ENCLS Instruction Leaf

If the RFLAGS.TF bit is set at the beginning of ENCLS, then a single-step debug exception is pending as a trap-class exception on the instruction boundary immediately after the ENCLS instruction. Additionally, if the instruction is executed in VMX non-root operation and the "monitor trap flag" VM-execution control is 1, an MTF VM exit is pending on the instruction boundary immediately after the instruction if the instruction does not fault.
40.2.2  Single Stepping ENCLU Instruction Leafs

The interactions of the unprivileged Intel SGX instruction ENCLU are leaf dependent.

An enclave entry via EENTER/ERESUME leaf functions of the ENCLU, in certain cases, may mask the RFLAGS.TF bit, and mask the setting of the "monitor trap flag" VM-execution control. In such situations, an exit from the enclave, either via the EEXIT leaf function or via an AEX unmask the RFLAGS.TF bit and the "monitor trap flag" VM-execution control. The details of this masking/unmasking and the pending of single stepping events across EENTER/ERESUME/EEXIT/AEX are covered in detail in Section 40.2.3.

If the EFLAGS.TF bit is set at the beginning of EREPORT or EGETKEY leafs, and if the EFLAGS.TF is not masked by the preceding enclave entry, then a single-step debug exception is pending on the instruction boundary immediately after the ENCLU instruction. Additionally, if the instruction is executed in VMX non-root operation and the "monitor trap flag" VM-execution control is 1, and if the monitor trap flag is not masked by the preceding enclave entry, then an MTF VM exit is pending on the instruction boundary immediately after the instruction.

If the instruction under consideration results in a fault, then the control flow goes to the fault handler, and no single-step debug exception is asserted. In such a situation, if the instruction is executed in VMX non-root operation and the "monitor trap flag" VM-execution control is 1, an MTF VM exit is pending after the delivery of the fault (or any nested exception). No MTF VM exit occurs if another VM exit occurs before reaching that boundary on which an MTF VM exit would be pending.

40.2.3  Single-Stepping Enclave Entry with Opt-out Entry

40.2.3.1  Single Stepping without AEX

Figure 40-1 shows the most common case for single-stepping after an opt-out entry.

In this scenario, if the RFLAGS.TF bit is set at the time of the enclave entry, then a single step debug exception is pending on the instruction boundary after EEXIT. Additionally, if the enclave is executing in VMX non-root operation and the "monitor trap flag" VM-execution control is 1, an MTF VM exit is pending after the delivery of the fault (or any nested exception). No MTF VM exit occurs if another VM exit occurs before reaching that boundary on which an MTF VM exit would be pending.

40.2.3.2  Single Step Preempted by AEX Due to Non-SMI Event

Figure 40-2 shows the interaction of single stepping with AEX due to a non-SMI event after an opt-out entry.
In this scenario, if the enclave is executing in VMX non-root operation and the “monitor trap flag” VM-execution control is 1, an MTF VM exit is pending on the instruction boundary after the AEX. No MTF VM exit occurs if another VM exit happens before reaching that instruction boundary.

The value of the RFLAGS.TF bit at the end of AEX is the same as the value of RFLAGS.TF at the time of the enclave entry.

### 40.2.4 RFLAGS.TF Treatment on AEX

The value of EFLAGS.TF at the end of AEX from an opt-out enclave is same as the value of EFLAGS.TF at the time of the enclave entry. The value of EFLAGS.TF at the end of AEX from an opt-in enclave is unmodified. The EFLAGS.TF saved in GPR portion of the SSA on an AEX is 0. For more detail see EENTER and ERESUME in Chapter 5.

### 40.2.5 Restriction on Setting of TF after an Opt-Out Entry

Enclave entered through an opt-out entry is not allowed to set EFLAGS.TF. The POPF instruction forces RFLAGS.TF to 0 if the enclave was entered through opt-out entry.

### 40.2.6 Trampoline Code Considerations

Any AEX from the enclave which results in the RFLAGS.TF =1 on the reporting stack will result in a single-step #DB after the first instruction of the trampoline code if the trampoline is entered using the IRET instruction.

### 40.3 CODE AND DATA BREAKPOINTS

#### 40.3.1 Breakpoint Suppression

Following an opt-out entry:
- Instruction breakpoints are suppressed during execution in an enclave.
- Data breakpoints are not triggered on accesses to the address range defined by ELRANGE.
- Data breakpoints are triggered on accesses to addresses outside the ELRANGE.
Following an opt-in entry instruction and data breakpoints are not suppressed. The processor does not report any matches on debug breakpoints that are suppressed on enclave entry. However, the processor does not clear any bits in DR6 that were already set at the time of the enclave entry.

### 40.3.2 Reporting of Instruction Breakpoint on Next Instruction on a Debug Trap

A debug exception caused by the single-step execution mode or when a data breakpoint condition was met causes the processor to perform an AEX. Following such an AEX, the processor reports in the debug status register (DR6) matches of the new instruction pointer (the AEP address) in a breakpoint address register setup to detect instruction execution.

### 40.3.3 RF Treatment on AEX

RF flag value saved in SSA is the same as what would have been pushed on stack if the exception or event causing the AEX occurred when executing outside an enclave (see Section 18.3.1.1). Following an AEX, the RF flag is 0 in the synthetic state.

### 40.3.4 Breakpoint Matching in Intel® SGX Instruction Flows

Implicit accesses made by Intel SGX instructions to EPC regions do not trigger data breakpoints. Explicit accesses made by ENCLS[ECREATE], ENCLS[EADD], ENCLS[EEXTEND], ENCLS[EINIT], ENCLS[EREMOVE], ENCLS[ETRACK], ENCLS[EBLOCK], ENCLS[EPA], ENCLS[EWB], ENCLS[EEXTEND], ENCLS[EDBGRD], ENCLS[EDBGWR], ENCLU[EENTER], and ENCLU[ERESUME] to the EPC operands do not trigger data breakpoints.

Explicit accesses made by the Intel SGX instructions (ENCLU[EGETKEY] and ENCLU[EREPORT]) executed by an enclave following an opt-in entry, trigger data breakpoints on accesses to their EPC operands. All Intel SGX instructions trigger data breakpoints on accesses to their non-EPC operands.

### 40.4 CONSIDERATION OF THE INT1 AND INT3 INSTRUCTIONS

This section considers the operation of the INT1 and INT3 instructions when executed inside an enclave. These are the instructions with opcodes F1 and CC, respectively, and not INT \( n \) (with opcode CD) with value 1 or 3 for \( n \).

#### 40.4.1 Behavior of INT1 and INT3 Inside an Enclave

An execution of either INT1 or INT3 inside an enclave results in a fault-class exception. Following an opt-out entry, execution of either instruction results in an invalid-opcode exception (#UD). Following opt-in entry, INT1 results in a debug exception (#DB) and INT3 delivers a breakpoint exception (#BP). The normal requirement for INT3 (that the CPL not be greater than the DPL of descriptor 3 in the IDT) is not enforced.

Because execution of INT1 or INT3 inside an enclave results in a fault, the RIP saved in the SSA on AEX references the INT1 or INT3 instruction (and not the following instruction). The RIP value saved on the stack (or in the TSS or VMCS) is that of the AEP.

If execution of INT1 or INT3 inside an enclave causes a VM exit, the event type in the VM-exit interruption information field indicates a hardware exception (type 3),\(^1\) and the VM-exit instruction length field is saved as zero.

#### 40.4.2 Debugger Considerations

A debugger using INT3 inside an enclave should account for the modified behavior described in Section 40.4.1. Because INT3 is fault-like inside an enclave, the RIP saved in the SSA on AEX is that of the INT3 instruction. Conse-

---

1. INT1 would normally indicate a privileged software exception (type 5), and INT3 would normally indicate a software exception (type 6).
quently, the debugger must not decrement SSA.RIP for #BP coming from an enclave to re-execute the instruction at the RIP of the INT3 instruction on a subsequent enclave entry.

### 40.4.3 VMM Considerations

As described in Section 40.4.1, execution of INT3 inside an enclave delivers #BP with “interruption type” of 3. A VMM that re-injects #BP into the guest should establish the VM-entry interruption information field using data saved into the appropriate VMCS fields by the VM exit incident to the #BP (as recommended in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C). VMMs that create the VM-entry interruption information based solely on the exception vector should take care to use event type 3 (instead of 6) when they detect a VM exit incident to enclave mode that is due to an exception with vector 3.

### 40.5 BRANCH TRACING

#### 40.5.1 BTF Treatment

When software enables single-stepping on branches then:

- Following an opt-in entry using EENTER the processor generates a single-step debug exception.
- Following an EEXIT the processor generates a single-step debug exception

Enclave entry using ERESUME (opt-in or opt-out) and an AEX from the enclave do not cause generation of the single-step debug exception.

#### 40.5.2 LBR Treatment

##### 40.5.2.1 LBR Stack on Opt-in Entry

Following an opt-in entry into an enclave, last branch recording facilities if enabled continued to store branch records in the LBR stack MSRs as follows:

- On enclave entry using EENTER/ERESUME, the processor push the address of EENTER/ERESUME instruction into MSR_LASTBRANCH_n_FROM_IP, and the destination address of the EENTER/ERESUME into MSR_LASTBRANCH_n_TO_IP.
- On EEXIT, the processor pushes the address of EEXIT instruction into MSR_LASTBRANCH_n_FROM_IP, and the address of EEXIT destination into MSR_LASTBRANCH_n_TO_IP.
- On AEX, the processor pushes RIP saved in the SSA into MSR_LASTBRANCH_n_FROM_IP, and the address of AEP into MSR_LASTBRANCH_n_TO_IP.
- For every branch inside the enclave, a branch record is pushed on the LBR stack.

Figure 40-3 shows an example of LBR stack manipulation after an opt-in entry. Every arrow in this picture indicates a branch record pushed on the LBR stack. The “From IP” of the branch record contains the linear address of the instruction located at the start of the arrow, while the “To IP” of the branch record contains the linear address of the instruction at the end of the arrow.
40.5.2.2 LBR Stack on Opt-out Entry

An opt-out entry into an enclave suppresses last branch recording facilities, and enclave exit after an opt-out entry un-suppresses last branch recording facilities.

Opt-out entry into an enclave does not push any record on LBR stack.

If last branch recording facilities were enabled at the time of enclave entry, then EEXIT following such an enclave entry pushes one record on LBR stack. The MSR_LASTBRANCH_n_FROM_IP of such record holds the linear address of the instruction (EENTER or ERESUME) that was used to enter the enclave, while the MSR_LASTBRANCH_n_TO_IP of such record holds linear address of the destination of EEXIT.

Additionally, if last branch recording facilities were enabled at the time of enclave entry, then an AEX after such an entry pushes one record on LBR stack, before pushing record for the event causing the AEX if the event pushes a record on LBR stack. The MSR_LASTBRANCH_n_FROM_IP of the new record holds linear address of the instruction (EENTER or ERESUME) that was used to enter the enclave, while MSR_LASTBRANCH_n_TO_IP of the new record holds linear address of the AEP. If the event causing AEX pushes a record on LBR stack, then the MSR_LASTBRANCH_n_FROM_IP for that record holds linear address of the AEP.

Figure 40-4 shows an example of LBR stack manipulation after an opt-out entry. Every arrow in this picture indicates a branch record pushed on the LBR stack. The “From IP” of the branch record contains the linear address of the instruction located at the start of the arrow, while the “To IP” of the branch record contains the linear address of the instruction at the end of the arrow.
40.5.2.3 Mispredict Bit, Record Type, and Filtering

All branch records resulting from Intel SGX instructions/AEXs are reported as predicted branches, and consequently, bit 63 of MSR_LASTBRANCH_n_FROM_IP for such records is set. Branch records due to these Intel SGX operations are always non-HLE/non-RTM records.

EENTER, ERESUME, EEXIT, and AEX are considered to be far branches. Consequently, bit 8 in MSR_LBR_SELECT controls filtering of the new records introduced by Intel SGX.

40.6 INTERACTION WITH PERFORMANCE MONITORING

40.6.1 IA32_PERF_GLOBAL_STATUS Enhancement

On processors supporting Intel SGX, the IA32_PERF_GLOBAL_STATUS MSR provides a bit indicator, known as “Anti Side-channel Interference” (ASCI) at bit position 60. If this bit is 0, the performance monitoring data in various performance monitoring counters are accumulated normally as defined by relevant architectural/microarchitectural conditions. If the ASCI bit is set, the contents in various performance monitoring counters can be affected by the direct or indirect consequence of Intel SGX protection of enclave code executing in the processor.

40.6.2 Performance Monitoring with Opt-in Entry

An opt-in enclave entry allow performance monitoring logic to observe the contribution of enclave code executing in the processor. Thus the contents of performance monitoring counters does not distinguish between contribution originating from enclave code or otherwise. All counters, events, precise events, etc. continue to work as defined in the IA32/Intel 64 Software Developer Manual. Consequently, bit 60 of IA32_PERF_GLOBAL_STATUS MSR is not set.
40.6.3 Performance Monitoring with Opt-out Entry

In general, performance monitoring activities are suppressed when entering an opt-out enclave. This applies to all thread-specific, configured performance monitoring, except for the cycle-counting fixed counter, IA32_FIXED_CTR1 and IA32_FIXED_CTR2. Upon entering an opt-out enclave, IA32_FIXED_CTR0, IA32_PMCx will stop accumulating counts. Additionally, if PEBS is configured to capture PEBS record for this thread, PEBS record generation will also be suppressed. Consequently, bit 60 of IA32_PERF_GLOBAL_STATUS MSR is set.

Performance monitoring on the sibling thread may also be affected. Any one of IA32_FIXED_CTRx or IA32_PMCx on the sibling thread configured to monitor thread-specific eventing logic with AnyThread = 1 is demoted to count only MyThread while an opt-out enclave is executing on the other thread.

40.6.4 Enclave Exit and Performance Monitoring

When a logical processor exits an enclave, either via ENCLU[EXIT] or via AEX, all performance monitoring activity (including PEBS) on that logical processor that was suppressed is unsuppressed. Any counters that were demoted from AnyThread to MyThread on the sibling thread are promoted back to AnyThread.

40.6.5 PEBS Record Generation on Intel® SGX Instructions

All leaf functions of the ENCLS instruction report “Eventing RIP” of the ENCLS instruction if a PEBS record is generated at the end of the instruction execution. Additionally, the EGETKEY and EREPORT leaf functions of the ENCLU instruction report “Eventing RIP” of the ENCLU instruction if a PEBS record is generated at the end of the instruction execution.

If the EENTER and ERESUME leaf functions are performing an opt-in entry report “Eventing RIP” of the ENCLU instruction if a PEBS record is generated at the end of the instruction execution. On the other hand, if these leaf functions are performing an opt-out entry, then these leaf functions result in PEBS being suppressed, and no PEBS record is generated at the end of these instructions.

A PEBS record is generated if there is a PEBS event pending at the end of EEXIT (due to a counter overflowing during enclave execution or during EEXIT execution). This PEBS record contains the architectural state of the logical processor at the end of EEXIT. If the enclave was entered via an opt-in entry, then this record reports the “Eventing RIP” as the linear address of the ENCLU[EXIT] instruction. If the enclave was entered via an opt-out entry, then the record reports the “Eventing RIP” as the linear address of the ENCLU[EENTER/ERESUME] instruction that performed the last enclave entry.

A PEBS record is generated after the AEX if there is a PEBS event pending at the end of AEX (due to a counter overflowing during enclave execution or during AEX execution). This PEBS record contains the synthetic state of the logical processor that is established at the end of AEX. For opt-in entry, this record has the EVENTING_RIP set to the RIP saved in the SSA. For opt-out entry, the record has the EVENTING_RIP set to the linear address of EENTER/ERESUME used for the last enclave entry.

If the enclave was entered via an opt-in entry, then this record reports the “Eventing RIP” as the linear address in the SSA of the enclave (a.k.a., the “Eventing LIP” inside the enclave). If the enclave was entered via an opt-out entry, then the record reports the “Eventing RIP” as the linear address of the ENCLU[EENTER/ERESUME] instruction that performed the last enclave entry.

A second PEBS event may be pended during the Enclave Exiting Event (EEE). If the PEBS event is taken at the end of delivery of the EEE then the “Eventing RIP” in this second PEBS record is the linear address of the AEP.

40.6.6 Exception-Handling on PEBS/BTS Loads/Stores after AEX

As noted in Section 18.4.9.2, recording in the BTS buffer or in the PEBS buffer may not operate properly if accesses to any of the DS save area sections cause page faults or VM exits. Such page faults or VM exits, if they occur, are delivered immediately to the OS or VMM, and generation of a BTS or PEBS record is skipped and may leave the buffers in a state where they have a partial BTS or PEBS records.

However, any events that are detected during PEBS/BTS record generation at the end of AEX and before delivering the Enclave Exiting Event (EEE) cannot be reported immediately to the OS/VMM, as an event window is not open at
the end of AEX. Consequently, fault-like events such as page faults, EPT faults, EPT mis-configuration, and accesses to APIC-access page detected on stores to the PEBS/BTS buffer are not reported, and generation of the PEBS and/or BTS record at the end of AEX is aborted (this may leave the buffers in a state where they have partial PEBS or BTS records). Trap-like events detected on stores to the PEBS/BTS buffer (such as debug traps) are pended until the next instruction boundary, where they are handled according to the architecturally defined priority. The processor continues the handling of the Enclave Exiting Event (SMI, NMI, interrupt, exception delivery, VM exit, etc.) after aborting the PEBS/BTS record generation.

40.6.6.1 Other Interactions with Performance Monitoring

For opt-in entry, EENTER, ERESUME, EEXIT, and AEX are all treated as predicted far branches, and any counters that are counting such branches are incremented by 1 as a part of retirement of these instructions. Retirement of these instructions is also counted in any counters configured to count instructions retired.

For opt-out entry, execution inside an enclave is treated as a single predicted branch, and all branch-counting performance monitoring counters are incremented accordingly. Additionally, such execution is also counted as a single instruction, and all performance monitoring counters counting instructions are incremented accordingly.

Enclave entry does not affect any performance monitoring counters shared between cores.
The ability of a processor to support VMX operation and related instructions is indicated by CPUID.1:ECX.VMX[bit 5] = 1. A value 1 in this bit indicates support for VMX features.

Support for specific features detailed in Chapter 27 and other VMX chapters is determined by reading values from a set of capability MSRs. These MSRs are indexed starting at MSR address 480H. VMX capability MSRs are read-only; an attempt to write them (with WRMSR) produces a general-protection exception (#GP(0)). They do not exist on processors that do not support VMX operation; an attempt to read them (with RDMSR) on such processors produces a general-protection exception (#GP(0)).

### A.1 BASIC VMX INFORMATION

The IA32_VMX_BASIC MSR (index 480H) consists of the following fields:

- Bits 30:0 contain the 31-bit VMCS revision identifier used by the processor. Processors that use the same VMCS revision identifier use the same size for VMCS regions (see subsequent item on bits 44:32).\(^1\)
- Bit 31 is always 0.
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are clear).
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor’s physical-address width.\(^2\) If the bit is 1, these addresses are limited to 32 bits. This bit is always 0 for processors that support Intel 64 architecture.
- If bit 49 is read as 1, the logical processor supports the dual-monitor treatment of system-management interrupts and system-management mode. See Section 32.15 for details of this treatment.
- Bits 53:50 report the memory type that should be used for the VMCS, for data structures referenced by pointers in the VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions), and for the MSEG header. If software needs to access these data structures (e.g., to modify the contents of the MSR bitmaps), it can configure the paging structures to map them into the linear-address space. If it does so, it should establish mappings that use the memory type reported bits 53:50 in this MSR.\(^3\)

As of this writing, all processors that support VMX operation indicate the write-back type. The values used are given in Table A-1.

<table>
<thead>
<tr>
<th>Value(s)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Uncacheable (UC)</td>
</tr>
<tr>
<td>1-5</td>
<td>Not used</td>
</tr>
<tr>
<td>6</td>
<td>Write Back (wB)</td>
</tr>
<tr>
<td>7-15</td>
<td>Not used</td>
</tr>
</tbody>
</table>

---

1. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field in bits 31:0 of this MSR. For all processors produced prior to this change, bit 31 of this MSR was read as 0.
2. On processors that support Intel 64 architecture, the pointer must not set bits beyond the processor’s physical address width.
3. Alternatively, software may map any of these regions or structures with the UC memory type. (This may be necessary for the MSEG header.) Doing so is discouraged unless necessary as it will cause the performance of software accesses to those structures to suffer.
• If bit 54 is read as 1, the processor reports information in the VM-exit instruction-information field on VM exits due to execution of the INS and OUTS instructions (see Section 28.2.5). This reporting is done only if this bit is read as 1.

• Bit 55 is read as 1 if any VMX controls that default to 1 may be cleared to 0. See Appendix A.2 for details. It also reports support for the VMX capability MSRs IA32_VMX_TRUE_PINBASED_CTLS, IA32_VMX_TRUE_PROCBASED_CTLS, IA32_VMX_TRUE_EXIT_CTLS, and IA32_VMX_TRUE_ENTRY_CTLS. See Appendix A.3.1, Appendix A.3.2, Appendix A.4, and Appendix A.5 for details.

• If bit 56 is read as 1, software can use VM entry to deliver a hardware exception with or without an error code, regardless of vector (see Section 27.2.1.3).

• The values of bits 47:45 and bits 63:57 are reserved and are read as 0.

A.2 RESERVED CONTROLS AND DEFAULT SETTINGS

As noted in Chapter 27, “VM Entries,” certain VMX controls are reserved and must be set to a specific value (0 or 1) determined by the processor. The specific value to which a reserved control must be set is its default setting. Software can discover the default setting of a reserved control by consulting the appropriate VMX capability MSR (see Appendix A.3 through Appendix A.5).

Future processors may define new functionality for one or more reserved controls. Such processors would allow each newly defined control to be set either to 0 or to 1. Software that does not desire a control's new functionality should set the control to its default setting. For that reason, it is useful for software to know the default settings of the reserved controls.

Default settings partition the various controls into the following classes:
- **Always-flexible.** These have never been reserved.
- **Default0.** These are (or have been) reserved with a default setting of 0.
- **Default1.** They are (or have been) reserved with a default setting of 1.

As noted in Appendix A.1, a logical processor uses bit 55 of the IA32_VMX_BASIC MSR to indicate whether any of the default1 controls may be 0:

• If bit 55 of the IA32_VMX_BASIC MSR is read as 0, all the default1 controls are reserved and must be 1. VM entry will fail if any of these controls are 0 (see Section 27.2.1).

• If bit 55 of the IA32_VMX_BASIC MSR is read as 1, not all the default1 controls are reserved, and some (but not necessarily all) may be 0. The CPU supports four (4) new VMX capability MSRs: IA32_VMX_TRUE_PINBASED_CTLS, IA32_VMX_TRUE_PROCBASED_CTLS, IA32_VMX_TRUE_EXIT_CTLS, and IA32_VMX_TRUE_ENTRY_CTLS. See Appendix A.3 through Appendix A.5 for details. (These MSRs are not supported if bit 55 of the IA32_VMX_BASIC MSR is read as 0.)

A.3 VM-EXECUTION CONTROLS

There are separate capability MSRs for the pin-based VM-execution controls, the primary processor-based VM-execution controls, the secondary processor-based VM-execution controls, and the tertiary processor-based VM-execution controls. These are described in Appendix A.3.1, Appendix A.3.2, Appendix A.3.3, and Appendix A.3.4, respectively.

A.3.1 Pin-Based VM-Execution Controls

The IA32_VMX_PINBASED_CTLS MSR (index 481H) reports on the allowed settings of most of the pin-based VM-execution controls (see Section 25.6.1):

• Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.
Exceptions are made for the pin-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 2, and 4; the corresponding bits of the IA32_VMX_PINBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any pin-based VM-execution control in the default1 class is 0.
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (see below) reports which of the pin-based VM-execution controls in the default1 class can be 0 on VM entry.

• Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (index 48DH) reports on the allowed settings of all of the pin-based VM-execution controls:

• Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the pin-based VM-execution controls:

• If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_PINBASED_CTLS MSR. (The IA32_VMX_TRUE_PINBASED_CTLS MSR is not supported.)
• If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_TRUE_PINBASED_CTLS MSR. Assuming that software knows that the default1 class of pin-based VM-execution controls contains bits 1, 2, and 4, there is no need for software to consult the IA32_VMX_PINBASED_CTLS MSR.

A.3.2 Primary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS MSR (index 482H) reports on the allowed settings of most of the primary processor-based VM-execution controls (see Section 25.6.2):

• Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the primary processor-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

Exceptions are made for the primary processor-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 4–6, 8, 13–16, and 26; the corresponding bits of the IA32_VMX_PROCBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any of the primary processor-based VM-execution controls in the default1 class is 0.
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (see below) reports which of the primary processor-based VM-execution controls in the default1 class can be 0 on VM entry.

• Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (index 48EH) reports on the allowed settings of all of the primary processor-based VM-execution controls:

• Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
• Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the primary processor-based VM-execution controls:
• If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the primary processor-based VM-execution controls is contained in the IA32_VMX_PROCBASED_CTLS MSR. (The IA32_VMX_TRUE_PROCBASED_CTLS MSR is not supported.)

• If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the primary processor-based VM-execution controls is contained in the IA32_VMX_TRUE_PROCBASED_CTLS MSR. Assuming that software knows that the default1 class of primary processor-based VM-execution controls contains bits 1, 4–6, 8, 13–16, and 26, there is no need for software to consult the IA32_VMX_PROCBASED_CTLS MSR.

A.3.3 Secondary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS2 MSR (index 48BH) reports on the allowed settings of the secondary processor-based VM-execution controls (see Section 25.6.2). The following items provide details, including enforcement by VM entry:

• Bits 31:0 indicate the allowed 0-settings of these controls. These bits are always 0. This fact indicates that VM entry allows each bit of the secondary processor-based VM-execution controls to be 0 (reserved bits must be 0)

• Bits 63:32 indicate the allowed 1-settings of these controls; the 1-setting is not allowed for any reserved bit. VM entry allows control X (bit X of the secondary processor-based VM-execution controls) to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X and the "activate secondary controls" primary processor-based VM-execution control are both 1.

The IA32_VMX_PROCBASED_CTLS2 MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1).

A.3.4 Tertiary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS3 MSR (index 492H) reports on the allowed 1-settings of the tertiary processor-based VM-execution controls; the 1-setting is not allowed for any reserved bit.

VM entry allows control X (bit X of the tertiary processor-based VM-execution controls) to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if control X and the "activate tertiary controls" primary processor-based VM-execution control are both 1.

The IA32_VMX_PROCBASED_CTLS3 MSR exists only on processors that support the 1-setting of the "activate tertiary controls" VM-execution control (only if bit 49 of the IA32_VMX_PROCBASED_CTLS MSR is 1).

Notice that the organization of this MSR differs from that of IA32_VMX_PROCBASED_CTLS2 (Appendix A.3.3). This is because there are 64 tertiary processor-based VM-execution controls, while there were only 32 secondary processor-based VM-execution controls.

A.4 VM-EXIT CONTROLS

There are separate capability MSRs for the primary VM-exit controls and the secondary VM-exit controls. These are described in Appendix A.4.1 and Appendix A.4.2, respectively.

A.4.1 Primary VM-Exit Controls

The IA32_VMX_EXIT_CTLS MSR (index 483H) reports on the allowed settings of most of the primary VM-exit controls (see Section 25.7.1):

• Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the primary VM-exit controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.
Exceptions are made for the primary VM-exit controls in the default1 class (see Appendix A.2). These are bits 0–8, 10, 11, 13, 14, 16, and 17; the corresponding bits of the IA32_VMX_EXIT_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any primary VM-exit control in the default1 class is 0.
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (see below) reports which of the primary VM-exit controls in the default1 class can be 0 on VM entry.

- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (index 48FH) reports on the allowed settings of all of the primary VM-exit controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the primary VM-exit controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the primary VM-exit controls is contained in the IA32_VMX_EXIT_CTLS MSR. (The IA32_VMX_TRUE_EXIT_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the primary VM-exit controls is contained in the IA32_VMX_TRUE_EXIT_CTLS MSR. Assuming that software knows that the default1 class of primary VM-exit controls contains bits 0–8, 10, 11, 13, 14, 16, and 17, there is no need for software to consult the IA32_VMX_EXIT_CTLS MSR.

### A.4.2 Secondary VM-Exit Controls

The IA32_VMX_EXIT_CTLS2 MSR (index 493H) reports on the allowed 1-settings of the secondary VM-exit controls (see Section 25.7.1); the 1-setting is not allowed for any reserved bit.

VM entry allows control X (bit X of the secondary VM-exit controls) to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if control X and the “activate secondary controls” primary VM-exit control are both 1.

The IA32_VMX_EXIT_CTLS2 MSR exists only on processors that support the 1-setting of the “activate secondary controls” VM-exit control (only if bit 63 of the IA32_VMX_EXIT_CTLS MSR is 1).

Notice that the organization of this MSR differs from that of IA32_VMX_EXIT_CTLS (Appendix A.4.1). This is because there are 64 secondary VM-exit controls, while there were only 32 primary VM-exit controls.

### A.5 VM-ENTRY CONTROLS

The IA32_VMX_ENTRY_CTLS MSR (index 484H) reports on the allowed settings of most of the VM-entry controls (see Section 25.8.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-entry controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

Exceptions are made for the VM-entry controls in the default1 class (see Appendix A.2). These are bits 0–8 and 12; the corresponding bits of the IA32_VMX_ENTRY_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-entry control in the default1 class is 0.
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (see below) reports which of the VM-entry controls in the default1 class can be 0 on VM entry.
• Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X is 1 in the VM-entry controls and bit 32+X is 0 in this MSR.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (index 490H) reports on the allowed settings of all of the VM-entry controls:
• Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
• Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-entry controls:
• If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_ENTRY_CTLS MSR. (The IA32_VMX_TRUE_ENTRY_CTLS MSR is not supported.)
• If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_TRUE_ENTRY_CTLS MSR. Assuming that software knows that the default1 class of VM-entry controls contains bits 0–8 and 12, there is no need for software to consult the IA32_VMX_ENTRY_CTLS MSR.

A.6 MISCELLANEOUS DATA

The IA32_VMX_MISC MSR (index 485H) consists of the following fields:
• Bits 4:0 report a value X that specifies the relationship between the rate of the VMX-preemption timer and that of the timestamp counter (TSC). Specifically, the VMX-preemption timer (if it is active) counts down by 1 every time bit X in the TSC changes due to a TSC increment.
• If bit 5 is read as 1, VM exits store the value of IA32_EFER.LMA into the “IA-32e mode guest” VM-entry control; see Section 28.2 for more details. This bit is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.
• Bits 8:6 report, as a bitmap, the activity states supported by the implementation:
  — Bit 6 reports (if set) the support for activity state 1 (HLT).
  — Bit 7 reports (if set) the support for activity state 2 (shutdown).
  — Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI).
If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. All implementations support VM entry to activity state 0 (active).
• If bit 14 is read as 1, Intel® Processor Trace (Intel PT) can be used in VMX operation. If the processor supports Intel PT but does not allow it to be used in VMX operation, execution of VMXON clears IA32_RTIT_CTL.TraceEn (see “VMXON—Enter VMX Operation” in Chapter 31); any attempt to write IA32_RTIT_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.
• If bit 15 is read as 1, the RDMSR instruction can be used in system-management mode (SMM) to read the IA32_SMBASE MSR (MSR address 9EH). See Section 32.15.6.3.
• Bits 24:16 indicate the number of CR3-target values supported by the processor. This number is a value between 0 and 256, inclusive (bit 24 is set if and only if bits 23:16 are clear).
• Bits 27:25 is used to compute the recommended maximum number of MSRs that should appear in the VM-exit MSR-store list, the VM-exit MSR-load list, or the VM-entry MSR-load list. Specifically, if the value bits 27:25 of IA32_VMX_MISC is N, then 512 * (N + 1) is the recommended maximum number of MSRs to be included in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).
• If bit 28 is read as 1, bit 2 of the IA32_SMM_MONITOR_CTL can be set to 1. VMXOFF unblocks SMIs unless IA32_SMM_MONITOR_CTL[bit 2] is 1 (see Section 32.14.4).
• If bit 29 is read as 1, software can use VMWRITE to write to any supported field in the VMCS; otherwise, VMWRITE cannot be used to modify VM-exit information fields.
• If bit 30 is read as 1, VM entry allows injection of a software interrupt, software exception, or privileged software exception with an instruction length of 0.
• Bits 63:32 report the 32-bit MSEG revision identifier used by the processor.
• Bits 13:9 and bit 31 are reserved and are read as 0.

A.7 VMX-FIXED BITS IN CR0
The IA32_VMX_CR0_FIXED0 MSR (index 486H) and IA32_VMX_CR0_FIXED1 MSR (index 487H) indicate how bits in CR0 may be set in VMX operation. They report on bits in CR0 that are allowed to be 0 and to be 1, respectively, in VMX operation. If bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit of CR0 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit of CR0 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit is also 1 in IA32_VMX_CR0_FIXED1; if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit is also 0 in IA32_VMX_CR0_FIXED0. Thus, each bit in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR0_FIXED0 and 1 in IA32_VMX_CR0_FIXED1).

A.8 VMX-FIXED BITS IN CR4
The IA32_VMX_CR4_FIXED0 MSR (index 488H) and IA32_VMX_CR4_FIXED1 MSR (index 489H) indicate how bits in CR4 may be set in VMX operation. They report on bits in CR4 that are allowed to be 0 and 1, respectively, in VMX operation. If bit X is 1 in IA32_VMX_CR4_FIXED0, then that bit of CR4 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit of CR4 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR4_FIXED0, then that bit is also 1 in IA32_VMX_CR4_FIXED1; if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit is also 0 in IA32_VMX_CR4_FIXED0. Thus, each bit in CR4 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR4_FIXED0 and 1 in IA32_VMX_CR4_FIXED1).

A.9 VMCS ENUMERATION
The IA32_VMX_VMCS_ENUM MSR (index 48AH) provides information to assist software in enumerating fields in the VMCS.

As noted in Section 25.11.2, each field in the VMCS is associated with a 32-bit encoding which is structured as follows:
• Bits 31:15 are reserved (must be 0).
• Bits 14:13 indicate the field’s width.
• Bit 12 is reserved (must be 0).
• Bits 11:10 indicate the field’s type.
• Bits 9:1 is an index field that distinguishes different fields with the same width and type.
• Bit 0 indicates access type.

IA32_VMX_VMCS_ENUM indicates to software the highest index value used in the encoding of any field supported by the processor:
• Bits 9:1 contain the highest index value used for any VMCS encoding.
• Bit 0 and bits 63:10 are reserved and are read as 0.
A.10 VPID AND EPT CAPABILITIES

The IA32_VMX_EPT_VPID_CAP MSR (index 48CH) reports information about the capabilities of the logical processor with regard to virtual-processor identifiers (VPIDs, Section 29.1) and extended page tables (EPT, Section 29.3):

- If bit 0 is read as 1, the processor supports execute-only translations by EPT. This support allows software to configure EPT paging-structure entries in which bits 1:0 are clear (indicating that data accesses are not allowed) and bit 2 is set (indicating that instruction fetches are allowed).¹
- Bit 6 indicates support for a page-walk length of 4.
- Bit 7 indicates support for a page-walk length of 5.
- If bit 8 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be uncacheable (UC); see Section 25.6.11.
- If bit 14 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be write-back (WB).
- If bit 16 is read as 1, the logical processor allows software to configure a EPT PDE to map a 2-Mbyte page (by setting bit 7 in the EPT PDE).
- If bit 17 is read as 1, the logical processor allows software to configure a EPT PDPTE to map a 1-Gbyte page (by setting bit 7 in the EPT PDPTE).
- Support for the INVEPT instruction (see Chapter 31 and Section 29.4.3.1):
  - If bit 20 is read as 1, the INVEPT instruction is supported.
  - If bit 25 is read as 1, the single-context INVEPT type is supported.
  - If bit 26 is read as 1, the all-context INVEPT type is supported.
- If bit 21 is read as 1, accessed and dirty flags for EPT are supported (see Section 29.3.5).
- If bit 22 is read as 1, the processor reports advanced VM-exit information for EPT violations (see Section 28.2.1). This reporting is done only if this bit is read as 1.
- If bit 23 is read as 1, supervisor shadow-stack control is supported (see Section 29.3.3.2).
- Support for the INVVPID instruction (see Chapter 31 and Section 29.4.3.1):
  - If bit 32 is read as 1, the INVVPID instruction is supported.
  - If bit 40 is read as 1, the individual-address INVVPID type is supported.
  - If bit 41 is read as 1, the single-context INVVPID type is supported.
  - If bit 42 is read as 1, the all-context INVVPID type is supported.
  - If bit 43 is read as 1, the single-context-retaining-globals INVVPID type is supported.
- Bits 53:48 enumerate the maximum HLAT prefix size. It is expected that any processor that supports the 1-setting of the "enable HLAT" VM-execution control will enumerate this value as 1. See Section 4.5.1.
- Bits 5:1, bits 13:9, bit 15, bits 19:18, bit 24, bits 31:27, bits 39:33, bits 47:44, and bits 63:54 are reserved and are read as 0.

The IA32_VMX_EPT_VPID_CAP MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1) and that support either the 1-setting of the "enable EPT" VM-execution control (only if bit 33 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1) or the 1-setting of the "enable VPID" VM-execution control (only if bit 37 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1).

¹. If the "mode-based execute control for EPT" VM-execution control is 1, setting bit 0 indicates also that software may also configure EPT paging-structure entries in which bits 1:0 are both clear and in which bit 10 is set (indicating a translation that can be used to fetch instructions from a supervisor-mode linear address or a user-mode linear address).
A.11 VM FUNCTIONS

The IA32_VMX_VMFUNC MSR (index 491H) reports on the allowed settings of the VM-function controls (see Section 25.6.14). VM entry allows bit X of the VM-function controls to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if bit X of the VM-function controls, the “activate secondary controls” primary processor-based VM-execution control, and the “enable VM functions” secondary processor-based VM-execution control are all 1.

The IA32_VMX_VMFUNC MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1) and the 1-setting of the "enable VM functions" secondary processor-based VM-execution control (only if bit 45 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1).
Every component of the VMCS is encoded by a 32-bit field that can be used by VMREAD and VMWRITE. Section 25.11.2 describes the structure of the encoding space (the meanings of the bits in each 32-bit encoding).

This appendix enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).

B.1 16-BIT FIELDS

A value of 0 in bits 14:13 of an encoding indicates a 16-bit field. Only guest-state areas and the host-state area contain 16-bit fields. As noted in Section 25.11.2, each 16-bit field allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.1.1 16-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-1 enumerates the 16-bit control fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual-processor identifier (VPID) 1</td>
<td>000000000B</td>
<td>00000000H</td>
</tr>
<tr>
<td>Posted-interrupt notification vector 2</td>
<td>000000001B</td>
<td>00000002H</td>
</tr>
<tr>
<td>EPTP index 3</td>
<td>000000010B</td>
<td>00000004H</td>
</tr>
<tr>
<td>HLAT prefix size 4</td>
<td>000000011B</td>
<td>00000006H</td>
</tr>
<tr>
<td>Last PID-pointer index 5</td>
<td>000000100B</td>
<td>00000008H</td>
</tr>
</tbody>
</table>

NOTES:
1. This field exists only on processors that support the 1-setting of the “enable VPID” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “process posted interrupts” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “enable HLAT” VM-execution control.
5. This field exists only on processors that support the 1-setting of the “IPI virtualization” VM-execution control.

B.1.2 16-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-2 enumerates 16-bit guest-state fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest ES selector</td>
<td>000000000B</td>
<td>00000800H</td>
</tr>
<tr>
<td>Guest CS selector</td>
<td>000000001B</td>
<td>00000802H</td>
</tr>
<tr>
<td>Guest SS selector</td>
<td>000000010B</td>
<td>00000804H</td>
</tr>
<tr>
<td>Guest DS selector</td>
<td>000000011B</td>
<td>00000806H</td>
</tr>
<tr>
<td>Guest FS selector</td>
<td>000000100B</td>
<td>00000808H</td>
</tr>
</tbody>
</table>
B.1.3 16-Bit Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-3 enumerates the 16-bit host-state fields.

Table B-3. Encodings for 16-Bit Host-State Fields (0000_11xx_xxxx_xxx0B)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest GS selector</td>
<td>000000101B</td>
<td>0000080AH</td>
</tr>
<tr>
<td>Guest LDTR selector</td>
<td>000000110B</td>
<td>0000080CH</td>
</tr>
<tr>
<td>Guest TR selector</td>
<td>000000111B</td>
<td>0000080EH</td>
</tr>
<tr>
<td>Guest interrupt status¹</td>
<td>000001000B</td>
<td>00000810H</td>
</tr>
<tr>
<td>PML index²</td>
<td>000001001B</td>
<td>00000812H</td>
</tr>
<tr>
<td>Guest UINV³</td>
<td>000001010B</td>
<td>00000814H</td>
</tr>
</tbody>
</table>

NOTES:
1. This field exists only on processors that support the 1-setting of the "virtual-interrupt delivery" VM-execution control.
2. This field exists only on processors that support the 1-setting of the "enable PML" VM-execution control.
3. This field exists only on processors that support the 1-setting of either the "clear UINV" VM-exit control or the "load UINV" VM-entry control.

B.2 64-BIT FIELDS

A value of 1 in bits 14:13 of an encoding indicates a 64-bit field. There are 64-bit fields only for controls and for guest state. As noted in Section 25.11.2, every 64-bit field has two encodings, which differ on bit 0, the access type. Thus, each such field has an even encoding for full access and an odd encoding for high access.

B.2.1 64-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-4 enumerates the 64-bit control fields.

Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxxAb)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of I/O bitmap A (full)</td>
<td>000000000B</td>
<td>00002000H</td>
</tr>
<tr>
<td>Address of I/O bitmap A (high)</td>
<td>000000001B</td>
<td>00002001H</td>
</tr>
<tr>
<td>Address of I/O bitmap B (full)</td>
<td>000000001B</td>
<td>00002002H</td>
</tr>
<tr>
<td>Address of I/O bitmap B (high)</td>
<td>000000001B</td>
<td>00002003H</td>
</tr>
<tr>
<td>Field Name</td>
<td>Index</td>
<td>Encoding</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>Address of MSR bitmaps (full)</td>
<td>00000010B</td>
<td>00002004H</td>
</tr>
<tr>
<td>Address of MSR bitmaps (high)</td>
<td>00000011B</td>
<td>00002005H</td>
</tr>
<tr>
<td>VM-exit MSR-store address (full)</td>
<td>00000010B</td>
<td>00002006H</td>
</tr>
<tr>
<td>VM-exit MSR-store address (high)</td>
<td>00000011B</td>
<td>00002007H</td>
</tr>
<tr>
<td>VM-exit MSR-load address (full)</td>
<td>00000100B</td>
<td>00002008H</td>
</tr>
<tr>
<td>VM-exit MSR-load address (high)</td>
<td>00000101B</td>
<td>00002009H</td>
</tr>
<tr>
<td>VM-entry MSR-load address (full)</td>
<td>00000101B</td>
<td>0000200AH</td>
</tr>
<tr>
<td>VM-entry MSR-load address (high)</td>
<td>00000101B</td>
<td>0000200BH</td>
</tr>
<tr>
<td>Executive-VMCS pointer (full)</td>
<td>00000110B</td>
<td>0000200CH</td>
</tr>
<tr>
<td>Executive-VMCS pointer (high)</td>
<td>00000110B</td>
<td>0000200DH</td>
</tr>
<tr>
<td>PML address (full)</td>
<td>00000111B</td>
<td>0000200EH</td>
</tr>
<tr>
<td>PML address (high)</td>
<td>00000111B</td>
<td>0000200FH</td>
</tr>
<tr>
<td>TSC offset (full)</td>
<td>00001000B</td>
<td>00002010H</td>
</tr>
<tr>
<td>TSC offset (high)</td>
<td>00001000B</td>
<td>00002011H</td>
</tr>
<tr>
<td>Virtual-APIC address (full)</td>
<td>00001001B</td>
<td>00002012H</td>
</tr>
<tr>
<td>Virtual-APIC address (high)</td>
<td>00001001B</td>
<td>00002013H</td>
</tr>
<tr>
<td>APIC-access address (full)</td>
<td>00001010B</td>
<td>00002014H</td>
</tr>
<tr>
<td>APIC-access address (high)</td>
<td>00001010B</td>
<td>00002015H</td>
</tr>
<tr>
<td>Posted-interrupt descriptor address (full)</td>
<td>00001011B</td>
<td>00002016H</td>
</tr>
<tr>
<td>Posted-interrupt descriptor address (high)</td>
<td>00001011B</td>
<td>00002017H</td>
</tr>
<tr>
<td>VM-function controls (full)</td>
<td>00001100B</td>
<td>00002018H</td>
</tr>
<tr>
<td>VM-function controls (high)</td>
<td>00001100B</td>
<td>00002019H</td>
</tr>
<tr>
<td>EPT pointer (EPTP; full)</td>
<td>00001101B</td>
<td>0000201AH</td>
</tr>
<tr>
<td>EPT pointer (EPTP; high)</td>
<td>00001101B</td>
<td>0000201BH</td>
</tr>
<tr>
<td>EOI-exit bitmap 0 (EOI_EXIT0; full)</td>
<td>00001110B</td>
<td>0000201CH</td>
</tr>
<tr>
<td>EOI-exit bitmap 0 (EOI_EXIT0; high)</td>
<td>00001110B</td>
<td>0000201DH</td>
</tr>
<tr>
<td>EOI-exit bitmap 1 (EOI_EXIT1; full)</td>
<td>00001111B</td>
<td>0000201EH</td>
</tr>
<tr>
<td>EOI-exit bitmap 1 (EOI_EXIT1; high)</td>
<td>00001111B</td>
<td>0000201FH</td>
</tr>
<tr>
<td>EOI-exit bitmap 2 (EOI_EXIT2; full)</td>
<td>00001000B</td>
<td>00002020H</td>
</tr>
<tr>
<td>EOI-exit bitmap 2 (EOI_EXIT2; high)</td>
<td>00001000B</td>
<td>00002021H</td>
</tr>
<tr>
<td>EOI-exit bitmap 3 (EOI_EXIT3; full)</td>
<td>00001001B</td>
<td>00002022H</td>
</tr>
<tr>
<td>EOI-exit bitmap 3 (EOI_EXIT3; high)</td>
<td>00001001B</td>
<td>00002023H</td>
</tr>
<tr>
<td>EPTP-list address (full)</td>
<td>00001010B</td>
<td>00002024H</td>
</tr>
<tr>
<td>EPTP-list address (high)</td>
<td>00001010B</td>
<td>00002025H</td>
</tr>
<tr>
<td>VMREAD-bitmap address (full)</td>
<td>00001011B</td>
<td>00002026H</td>
</tr>
<tr>
<td>VMREAD-bitmap address (high)</td>
<td>00001011B</td>
<td>00002027H</td>
</tr>
<tr>
<td>VMWRITE-bitmap address (full)</td>
<td>00001010B</td>
<td>00002028H</td>
</tr>
<tr>
<td>VMWRITE-bitmap address (high)</td>
<td>00001010B</td>
<td>00002029H</td>
</tr>
</tbody>
</table>
### Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb) (Contd.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtualization-exception information address (full)</td>
<td>000010101B</td>
<td>0000202AH</td>
</tr>
<tr>
<td>Virtualization-exception information address (high)</td>
<td>000010110B</td>
<td>0000202BH</td>
</tr>
<tr>
<td>XSS-exiting bitmap (full)</td>
<td>000010110B</td>
<td>0000202CH</td>
</tr>
<tr>
<td>XSS-exiting bitmap (high)</td>
<td>000010110B</td>
<td>0000202DH</td>
</tr>
<tr>
<td>ENCLS-exiting bitmap (full)</td>
<td>000010111B</td>
<td>0000202EH</td>
</tr>
<tr>
<td>ENCLS-exiting bitmap (high)</td>
<td>000010111B</td>
<td>0000202FH</td>
</tr>
<tr>
<td>Sub-page-permission-table pointer (full)</td>
<td>000011000B</td>
<td>00002030H</td>
</tr>
<tr>
<td>Sub-page-permission-table pointer (high)</td>
<td>000011000B</td>
<td>00002031H</td>
</tr>
<tr>
<td>TSC multiplier (full)</td>
<td>000011001B</td>
<td>00002032H</td>
</tr>
<tr>
<td>TSC multiplier (high)</td>
<td>000011001B</td>
<td>00002033H</td>
</tr>
<tr>
<td>Tertiary processor-based VM-execution controls (full)</td>
<td>000011010B</td>
<td>00002034H</td>
</tr>
<tr>
<td>Tertiary processor-based VM-execution controls (high)</td>
<td>000011010B</td>
<td>00002035H</td>
</tr>
<tr>
<td>ENCLV-exiting bitmap (full)</td>
<td>000011011B</td>
<td>00002036H</td>
</tr>
<tr>
<td>ENCLV-exiting bitmap (high)</td>
<td>000011011B</td>
<td>00002037H</td>
</tr>
<tr>
<td>Low PASID directory address (full)</td>
<td>000011100B</td>
<td>00002038H</td>
</tr>
<tr>
<td>Low PASID directory address (high)</td>
<td>000011100B</td>
<td>00002039H</td>
</tr>
<tr>
<td>High PASID directory address (full)</td>
<td>000011101B</td>
<td>0000203AH</td>
</tr>
<tr>
<td>High PASID directory address (high)</td>
<td>000011101B</td>
<td>0000203BH</td>
</tr>
<tr>
<td>Shared EPT pointer (full)</td>
<td>000011110B</td>
<td>0000203CH</td>
</tr>
<tr>
<td>Shared EPT pointer (high)</td>
<td>000011110B</td>
<td>0000203DH</td>
</tr>
<tr>
<td>PCONFIG-exiting bitmap (full)</td>
<td>000011111B</td>
<td>0000203EH</td>
</tr>
<tr>
<td>PCONFIG-exiting bitmap (high)</td>
<td>000011111B</td>
<td>0000203FH</td>
</tr>
<tr>
<td>Hypervisor-managed linear-address translation pointer (HLATP; full)</td>
<td>000100000B</td>
<td>00002040H</td>
</tr>
<tr>
<td>HLATP (high)</td>
<td>000100000B</td>
<td>00002041H</td>
</tr>
<tr>
<td>PID-pointer table address (full)</td>
<td>000100001B</td>
<td>00002042H</td>
</tr>
<tr>
<td>PID-pointer table address (high)</td>
<td>000100001B</td>
<td>00002043H</td>
</tr>
<tr>
<td>Secondary VM-exit controls (full)</td>
<td>000100010B</td>
<td>00002044H</td>
</tr>
<tr>
<td>Secondary VM-exit controls (high)</td>
<td>000100010B</td>
<td>00002045H</td>
</tr>
<tr>
<td>IA32_SPEC_CTRL mask (full)</td>
<td>000100101B</td>
<td>0000204AH</td>
</tr>
<tr>
<td>IA32_SPEC_CTRL mask (high)</td>
<td>000100101B</td>
<td>0000204BH</td>
</tr>
<tr>
<td>IA32_SPEC_CTRL shadow (full)</td>
<td>000100110B</td>
<td>0000204CH</td>
</tr>
<tr>
<td>IA32_SPEC_CTRL shadow (high)</td>
<td>000100110B</td>
<td>0000204DH</td>
</tr>
</tbody>
</table>

**NOTES:**
1. This field exists only on processors that support the 1-setting of the “use MSR bitmaps” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “enable PML” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “virtualize APIC accesses” VM-execution control.
5. This field exists only on processors that support the 1-setting of the “process posted interrupts” VM-execution control.
6. This field exists only on processors that support the 1-setting of the “enable VM functions” VM-execution control.
7. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.
8. This field exists only on processors that support the 1-setting of the “virtual-interrupt delivery” VM-execution control.
9. This field exists only on processors that support the 1-setting of the “EPT switching” VM-function control.
10. This field exists only on processors that support the 1-setting of the “VMCS shadowing” VM-execution control.
11. This field exists only on processors that support the 1-setting of the “EPT-violation #VE” VM-execution control.
12. This field exists only on processors that support the 1-setting of the “enable XSAVES/XRSTORS” VM-execution control.
13. This field exists only on processors that support the 1-setting of the “enable ENCLS exiting” VM-execution control.
14. This field exists only on processors that support the 1-setting of the “sub-page write permissions for EPT” VM-execution control.
15. This field exists only on processors that support the 1-setting of the “use TSC scaling” VM-execution control.
16. This field exists only on processors that support the 1-setting of the “activate tertiary controls” VM-execution control.
17. This field exists only on processors that support the 1-setting of the “enable ENCLV exiting” VM-execution control.
18. This field exists only on processors that support the 1-setting of the “PASID translation” VM-execution control.
19. This field exists only on processors that support the 1-setting of the “shared-EPTP” VM-execution control.
20. This field exists only on processors that support the 1-setting of the “enable PCONFIG” VM-execution control.
21. This field exists only on processors that support the 1-setting of the “enable HLAT” VM-execution control.
22. This field exists only on processors that support the 1-setting of the “IPI virtualization” VM-execution control.
23. This field exists only on processors that support the 1-setting of the “activate secondary controls” VM-exit control.
24. This field exists only on processors that support the 1-setting of the “virtualize IA32_SPEC_CTRL” VM-execution control.

### B.2.2 64-Bit Read-Only Data Field

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. There is only one such 64-bit field as given in Table B-5. (As with other 64-bit fields, this one has two encodings.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest-physical address (full)¹</td>
<td>00000000B</td>
<td>00002400H</td>
</tr>
<tr>
<td>Guest-physical address (high)¹</td>
<td></td>
<td>00002401H</td>
</tr>
</tbody>
</table>

**NOTES:**
1. This field exists only on processors that support the 1-setting of the “enable EPT” VM-execution control.

### B.2.3 64-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-6 enumerates the 64-bit guest-state fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMCS link pointer (full)</td>
<td>00000000B</td>
<td>00002800H</td>
</tr>
<tr>
<td>VMCS link pointer (high)</td>
<td></td>
<td>00002801H</td>
</tr>
<tr>
<td>Guest IA32_DEBUGCTL (full)</td>
<td>00000001B</td>
<td>00002802H</td>
</tr>
<tr>
<td>Guest IA32_DEBUGCTL (high)</td>
<td></td>
<td>00002803H</td>
</tr>
<tr>
<td>Guest IA32_PAT (full)¹</td>
<td>00000001B</td>
<td>00002804H</td>
</tr>
<tr>
<td>Guest IA32_PAT (high)¹</td>
<td></td>
<td>00002805H</td>
</tr>
<tr>
<td>Guest IA32_EFER (full)²</td>
<td>00000001B</td>
<td>00002806H</td>
</tr>
<tr>
<td>Guest IA32_EFER (high)²</td>
<td></td>
<td>00002807H</td>
</tr>
</tbody>
</table>

**NOTES:**
1. This field exists only on processors that support the 1-setting of the “enable ENCLS exiting” VM-execution control.
B.2.4  64-Bit Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-7 enumerates the 64-bit control fields.

### Table B-7. Encodings for 64-Bit Host-State Fields (0010_11xx_xxxx xxxxAb)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host IA32_PAT (full)</td>
<td>000000000B</td>
<td>00002C00H</td>
</tr>
<tr>
<td>Host IA32_PAT (high)</td>
<td>000000000B</td>
<td>00002C01H</td>
</tr>
</tbody>
</table>

### NOTES:
1. This field exists only on processors that support either the 1-setting of the "load IA32_PAT" VM-entry control or that of the "save IA32_PAT" VM-exit control.
2. This field exists only on processors that support either the 1-setting of the "load IA32_EFER" VM-entry control or that of the "save IA32_EFER" VM-exit control.
3. This field exists only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-entry control.
4. This field exists only on processors that support the 1-setting of the "enable EPT" VM-execution control.
5. This field exists only on processors that support either the 1-setting of the "load IA32_BNDCFGS" VM-entry control or that of the "clear IA32_BNDCFGS" VM-exit control.
6. This field exists only on processors that support either the 1-setting of the "load IA32_RTIT_CTL" VM-entry control or that of the "clear IA32_RTIT_CTL" VM-exit control.
7. This field exists only on processors that support either the 1-setting of the "load IA32_LBR_CTL" VM-entry control or that of the "clear IA32_LBR_CTL" VM-exit control.
8. This field exists only on processors that support the 1-setting of the "load PKRS" VM-entry control.

### Table B-7. Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx xxxxAb) (Contd.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest IA32_PERF_GLOBAL_CTRL (full)³</td>
<td>000000100B</td>
<td>00002808H</td>
</tr>
<tr>
<td>Guest IA32_PERF_GLOBAL_CTRL (high)³</td>
<td>000000101B</td>
<td>00002809H</td>
</tr>
<tr>
<td>Guest PDPTE0 (full)⁴</td>
<td>000000011B</td>
<td>0000280AH</td>
</tr>
<tr>
<td>Guest PDPTE0 (high)⁴</td>
<td>000000011B</td>
<td>0000280BH</td>
</tr>
<tr>
<td>Guest PDPTE1 (full)⁴</td>
<td>000000110B</td>
<td>0000280CH</td>
</tr>
<tr>
<td>Guest PDPTE1 (high)⁴</td>
<td>000000110B</td>
<td>0000280DH</td>
</tr>
<tr>
<td>Guest PDPTE2 (full)⁴</td>
<td>000000111B</td>
<td>0000280EH</td>
</tr>
<tr>
<td>Guest PDPTE2 (high)⁴</td>
<td>000000111B</td>
<td>0000280FH</td>
</tr>
<tr>
<td>Guest PDPTE3 (full)⁴</td>
<td>000000100B</td>
<td>00002810H</td>
</tr>
<tr>
<td>Guest PDPTE3 (high)⁴</td>
<td>000000100B</td>
<td>00002811H</td>
</tr>
<tr>
<td>Guest IA32_BNDCFGS (full)⁵</td>
<td>000001000B</td>
<td>00002812H</td>
</tr>
<tr>
<td>Guest IA32_BNDCFGS (high)⁵</td>
<td>000001000B</td>
<td>00002813H</td>
</tr>
<tr>
<td>Guest IA32_RTIT_CTL (full)⁶</td>
<td>000001010B</td>
<td>00002814H</td>
</tr>
<tr>
<td>Guest IA32_RTIT_CTL (high)⁶</td>
<td>000001010B</td>
<td>00002815H</td>
</tr>
<tr>
<td>Guest IA32_LBR_CTL (full)⁷</td>
<td>000001011B</td>
<td>00002816H</td>
</tr>
<tr>
<td>Guest IA32_LBR_CTL (high)⁷</td>
<td>000001011B</td>
<td>00002817H</td>
</tr>
<tr>
<td>Guest IA32_PKRS (full)⁸</td>
<td>000001100B</td>
<td>00002818H</td>
</tr>
<tr>
<td>Guest IA32_PKRS (high)⁸</td>
<td>000001100B</td>
<td>00002819H</td>
</tr>
</tbody>
</table>

NOTES:
1. This field exists only on processors that support either the 1-setting of the "load IA32_PAT" VM-entry control or that of the "save IA32_PAT" VM-exit control.
2. This field exists only on processors that support either the 1-setting of the "load IA32_EFER" VM-entry control or that of the "save IA32_EFER" VM-exit control.
3. This field exists only on processors that support the 1-setting of the "load IA32_PERF_GLOBAL_CTRL" VM-entry control.
4. This field exists only on processors that support the 1-setting of the "enable EPT" VM-execution control.
5. This field exists only on processors that support either the 1-setting of the "load IA32_BNDCFGS" VM-entry control or that of the "clear IA32_BNDCFGS" VM-exit control.
6. This field exists only on processors that support either the 1-setting of the "load IA32_RTIT_CTL" VM-entry control or that of the "clear IA32_RTIT_CTL" VM-exit control.
7. This field exists only on processors that support either the 1-setting of the "load IA32_LBR_CTL" VM-entry control or that of the "clear IA32_LBR_CTL" VM-exit control.
8. This field exists only on processors that support the 1-setting of the "load PKRS" VM-entry control.
B.3 32-BIT FIELDS

A value of 2 in bits 14:13 of an encoding indicates a 32-bit field. As noted in Section 25.11.2, each 32-bit field allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

B.3.1 32-Bit Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-8 enumerates the 32-bit control fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin-based VM-execution controls</td>
<td>000000000B</td>
<td>0000000AB</td>
</tr>
<tr>
<td>Primary processor-based VM-execution controls</td>
<td>000000001B</td>
<td>0000000BB</td>
</tr>
<tr>
<td>Exception bitmap</td>
<td>000000010B</td>
<td>0000000CB</td>
</tr>
<tr>
<td>Page-fault error-code mask</td>
<td>000000011B</td>
<td>0000000DB</td>
</tr>
<tr>
<td>Page-fault error-code match</td>
<td>000000100B</td>
<td>00000010B</td>
</tr>
<tr>
<td>CR3-target count</td>
<td>000000101B</td>
<td>00000011B</td>
</tr>
<tr>
<td>Primary VM-exit controls</td>
<td>000000110B</td>
<td>00000012B</td>
</tr>
<tr>
<td>VM-exit MSR-store count</td>
<td>000000111B</td>
<td>00000013B</td>
</tr>
<tr>
<td>VM-exit MSR-load count</td>
<td>000001000B</td>
<td>00000100B</td>
</tr>
<tr>
<td>VM-entry controls</td>
<td>000001001B</td>
<td>00000101B</td>
</tr>
<tr>
<td>VM-entry MSR-load count</td>
<td>000001010B</td>
<td>00000102B</td>
</tr>
<tr>
<td>VM-entry interruption-information field</td>
<td>000001011B</td>
<td>00000103B</td>
</tr>
<tr>
<td>VM-entry exception error code</td>
<td>000001100B</td>
<td>00000110B</td>
</tr>
<tr>
<td>VM-entry instruction length</td>
<td>000001101B</td>
<td>00000111B</td>
</tr>
<tr>
<td>TPR threshold1</td>
<td>000001110B</td>
<td>00000112B</td>
</tr>
<tr>
<td>Secondary processor-based VM-execution controls2</td>
<td>000001111B</td>
<td>00000113B</td>
</tr>
<tr>
<td>PLE_Gap3</td>
<td>000010000B</td>
<td>00001000B</td>
</tr>
</tbody>
</table>
FIELD ENCODING IN VMCS

Table B-8. Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B) (Contd.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLE_Window</td>
<td>000010001B</td>
<td>00004022H</td>
</tr>
<tr>
<td>Instruction-timeout control</td>
<td>000010010B</td>
<td>00004024H</td>
</tr>
</tbody>
</table>

NOTES:
1. This field exists only on processors that support the 1-setting of the “use TPR shadow” VM-execution control.
2. This field exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control.
3. This field exists only on processors that support the 1-setting of the “PAUSE-loop exiting” VM-execution control.
4. This field exists only on processors that support the 1-setting of the “instruction timeout” VM-execution control.

B.3.2 32-Bit Read-Only Data Fields

A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-9 enumerates the 32-bit read-only data fields.

Table B-9. Encodings for 32-Bit Read-Only Data Fields (0100_01xx_xxxx_xxx0B)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM-instruction error</td>
<td>000000000B</td>
<td>00004400H</td>
</tr>
<tr>
<td>Exit reason</td>
<td>000000001B</td>
<td>00004402H</td>
</tr>
<tr>
<td>VM-exit interruption information</td>
<td>000000010B</td>
<td>00004404H</td>
</tr>
<tr>
<td>VM-exit interruption error code</td>
<td>000000011B</td>
<td>00004406H</td>
</tr>
<tr>
<td>IDT-vectoring information field</td>
<td>000000100B</td>
<td>00004408H</td>
</tr>
<tr>
<td>IDT-vectoring error code</td>
<td>000000101B</td>
<td>0000440AH</td>
</tr>
<tr>
<td>VM-exit instruction length</td>
<td>000000110B</td>
<td>0000440CH</td>
</tr>
<tr>
<td>VM-exit instruction information</td>
<td>000000111B</td>
<td>0000440EH</td>
</tr>
</tbody>
</table>

B.3.3 32-Bit Guest-State Fields

A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-10 enumerates the 32-bit guest-state fields.

Table B-10. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest ES limit</td>
<td>000000000B</td>
<td>00004800H</td>
</tr>
<tr>
<td>Guest CS limit</td>
<td>000000001B</td>
<td>00004802H</td>
</tr>
<tr>
<td>Guest SS limit</td>
<td>000000010B</td>
<td>00004804H</td>
</tr>
<tr>
<td>Guest DS limit</td>
<td>000000011B</td>
<td>00004806H</td>
</tr>
<tr>
<td>Guest FS limit</td>
<td>000000100B</td>
<td>00004808H</td>
</tr>
<tr>
<td>Guest GS limit</td>
<td>000000101B</td>
<td>0000480AH</td>
</tr>
<tr>
<td>Guest LDTR limit</td>
<td>000000110B</td>
<td>0000480CH</td>
</tr>
<tr>
<td>Guest TR limit</td>
<td>000000111B</td>
<td>00004810H</td>
</tr>
<tr>
<td>Guest GDTR limit</td>
<td>00001000B</td>
<td>00004812H</td>
</tr>
<tr>
<td>Guest IDTR limit</td>
<td>00001001B</td>
<td>00004814H</td>
</tr>
<tr>
<td>Guest ES access rights</td>
<td>00001010B</td>
<td>00004816H</td>
</tr>
</tbody>
</table>
The limit fields for GDTR and IDTR are defined to be 32 bits in width even though these fields are only 16-bits wide in the Intel 64 and IA-32 architectures. VM entry ensures that the high 16 bits of both these fields are cleared to 0.

### B.3.4 32-Bit Host-State Field

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. There is only one such 32-bit field as given in Table B-11.

### B.4 NATURAL-WIDTH FIELDS

A value of 3 in bits 14:13 of an encoding indicates a natural-width field. As noted in Section 25.11.2, each of these fields allows only full access, meaning that bit 0 of its encoding is 0. Each such encoding is thus an even number.

#### B.4.1 Natural-Width Control Fields

A value of 0 in bits 11:10 of an encoding indicates a control field. These fields are distinguished by their index value in bits 9:1. Table B-12 enumerates the natural-width control fields.
FIELD ENCODING IN VMCS

B.4.2  Natural-Width Read-Only Data Fields
A value of 1 in bits 11:10 of an encoding indicates a read-only data field. These fields are distinguished by their index value in bits 9:1. Table B-13 enumerates the natural-width read-only data fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit qualification</td>
<td>000000000B</td>
<td>00006400H</td>
</tr>
<tr>
<td>I/O RCX</td>
<td>000000001B</td>
<td>00006402H</td>
</tr>
<tr>
<td>I/O RSI</td>
<td>000000010B</td>
<td>00006404H</td>
</tr>
<tr>
<td>I/O RDI</td>
<td>000000011B</td>
<td>00006406H</td>
</tr>
<tr>
<td>I/O RIP</td>
<td>000000100B</td>
<td>00006408H</td>
</tr>
<tr>
<td>Guest-linear address</td>
<td>000000101B</td>
<td>0000640AH</td>
</tr>
</tbody>
</table>

B.4.3  Natural-Width Guest-State Fields
A value of 2 in bits 11:10 of an encoding indicates a field in the guest-state area. These fields are distinguished by their index value in bits 9:1. Table B-14 enumerates the natural-width guest-state fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest CR0</td>
<td>000000000B</td>
<td>00006800H</td>
</tr>
<tr>
<td>Guest CR3</td>
<td>000000001B</td>
<td>00006802H</td>
</tr>
<tr>
<td>Guest CR4</td>
<td>000000010B</td>
<td>00006804H</td>
</tr>
<tr>
<td>Guest ES base</td>
<td>000000011B</td>
<td>00006806H</td>
</tr>
<tr>
<td>Guest CS base</td>
<td>000000100B</td>
<td>00006808H</td>
</tr>
<tr>
<td>Guest SS base</td>
<td>000000101B</td>
<td>0000680AH</td>
</tr>
<tr>
<td>Guest DS base</td>
<td>00000110B</td>
<td>0000680CH</td>
</tr>
<tr>
<td>Guest FS base</td>
<td>00000111B</td>
<td>0000680EH</td>
</tr>
<tr>
<td>Guest GS base</td>
<td>00001000B</td>
<td>00006810H</td>
</tr>
<tr>
<td>Guest LDTR base</td>
<td>00001001B</td>
<td>00006812H</td>
</tr>
<tr>
<td>Guest TR base</td>
<td>00001010B</td>
<td>00006814H</td>
</tr>
<tr>
<td>Guest GDTR base</td>
<td>00001011B</td>
<td>00006816H</td>
</tr>
</tbody>
</table>

NOTES:
1. If a future implementation supports more than 4 CR3-target values, they will be encoded consecutively following the 4 encodings given here.
The base-address fields for ES, CS, SS, and DS in the guest-state area are defined to be natural-width (with 64 bits on processors supporting Intel 64 architecture) even though these fields are only 32-bits wide in the Intel 64 architecture. VM entry ensures that the high 32 bits of these fields are cleared to 0.

B.4.4 Natural-Width Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-15 enumerates the natural-width host-state fields.

### Table B-15. Encodings for Natural-Width Host-State Fields (0110_10xx_xxxx_xx0B) (Contd.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest IDTR base</td>
<td>000000110B</td>
<td>00006818H</td>
</tr>
<tr>
<td>Guest DR7</td>
<td>000000110B</td>
<td>0000681AH</td>
</tr>
<tr>
<td>Guest RSP</td>
<td>000000110B</td>
<td>0000681CH</td>
</tr>
<tr>
<td>Guest RIP</td>
<td>000000111B</td>
<td>0000681EH</td>
</tr>
<tr>
<td>Guest RFLAGS</td>
<td>000010000B</td>
<td>00006820H</td>
</tr>
<tr>
<td>Guest pending debug exceptions</td>
<td>000010001B</td>
<td>00006822H</td>
</tr>
<tr>
<td>Guest IA32_SYSENTER_ESP</td>
<td>000010010B</td>
<td>00006824H</td>
</tr>
<tr>
<td>Guest IA32_SYSENTER_EIP</td>
<td>000010011B</td>
<td>00006826H</td>
</tr>
<tr>
<td>Guest IA32_S_CET$^1$</td>
<td>000010100B</td>
<td>00006828H</td>
</tr>
<tr>
<td>Guest SSP$^1$</td>
<td>000010101B</td>
<td>0000682AH</td>
</tr>
<tr>
<td>Guest IA32_INTERRUPT_SSP_TABLE_ADDR$^1$</td>
<td>000010110B</td>
<td>0000682CH</td>
</tr>
</tbody>
</table>

NOTES:
1. This field is supported only on processors that support the 1-setting of the “load CET state” VM-entry control.

The base-address fields for ES, CS, SS, and DS in the guest-state area are defined to be natural-width (with 64 bits on processors supporting Intel 64 architecture) even though these fields are only 32-bits wide in the Intel 64 architecture. VM entry ensures that the high 32 bits of these fields are cleared to 0.

B.4.4 Natural-Width Host-State Fields

A value of 3 in bits 11:10 of an encoding indicates a field in the host-state area. These fields are distinguished by their index value in bits 9:1. Table B-15 enumerates the natural-width host-state fields.

### Table B-15. Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xx0B)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host CR0</td>
<td>000000000B</td>
<td>00006C00H</td>
</tr>
<tr>
<td>Host CR3</td>
<td>000000001B</td>
<td>00006C02H</td>
</tr>
<tr>
<td>Host CR4</td>
<td>000000010B</td>
<td>00006C04H</td>
</tr>
<tr>
<td>Host FS base</td>
<td>000000011B</td>
<td>00006C06H</td>
</tr>
<tr>
<td>Host GS base</td>
<td>000000100B</td>
<td>00006C08H</td>
</tr>
<tr>
<td>Host TR base</td>
<td>000000101B</td>
<td>00006C0AH</td>
</tr>
<tr>
<td>Host GDTR base</td>
<td>000000110B</td>
<td>00006C0CH</td>
</tr>
<tr>
<td>Host IDTR base</td>
<td>000000111B</td>
<td>00006C0EH</td>
</tr>
<tr>
<td>Host IA32_SYSENTER_ESP</td>
<td>000001000B</td>
<td>00006C10H</td>
</tr>
<tr>
<td>Host IA32_SYSENTER_EIP</td>
<td>000001001B</td>
<td>00006C12H</td>
</tr>
<tr>
<td>Host RSP</td>
<td>000001010B</td>
<td>00006C14H</td>
</tr>
<tr>
<td>Host RIP</td>
<td>000001011B</td>
<td>00006C16H</td>
</tr>
<tr>
<td>Host IA32_S_CET$^1$</td>
<td>000001100B</td>
<td>00006C18H</td>
</tr>
<tr>
<td>Host SSP$^1$</td>
<td>000001101B</td>
<td>00006C1AH</td>
</tr>
<tr>
<td>Host IA32_INTERRUPT_SSP_TABLE_ADDR$^1$</td>
<td>000001110B</td>
<td>00006C1CH</td>
</tr>
</tbody>
</table>

NOTES:
1. This field is supported only on processors that support the 1-setting of the “load CET state” VM-exit control.
APPENDIX C
VMX BASIC EXIT REASONS

Every VM exit writes a 32-bit exit reason to the VMCS (see Section 25.9.1). Certain VM-entry failures also do this (see Section 27.8). The low 16 bits of the exit-reason field form the basic exit reason which provides basic information about the cause of the VM exit or VM-entry failure.

Table C-1 lists values for basic exit reasons and explains their meaning. Entries apply to VM exits, unless otherwise noted.

<table>
<thead>
<tr>
<th>Basic Exit Reason</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Exception or non-maskable interrupt (NMI). Either:</td>
</tr>
<tr>
<td></td>
<td>1: Guest software caused an exception and the bit in the exception bitmap associated with exception's vector was 1. This case includes executions of BOUND that cause #BR, executions of INT1 (they cause #DB), executions of INT3 (they cause #BP), executions of INT0 that cause #OF, and executions of UD0, UD1, and UD2 (they cause #UD).</td>
</tr>
<tr>
<td></td>
<td>2: An NMI was delivered to the logical processor and the &quot;NMI exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>1</td>
<td>External interrupt. An external interrupt arrived and the &quot;external-interrupt exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>2</td>
<td>Triple fault. The logical processor encountered an exception while attempting to call the double-fault handler and that exception did not itself cause a VM exit due to the exception bitmap.</td>
</tr>
<tr>
<td>3</td>
<td>INIT signal. An INIT signal arrived</td>
</tr>
<tr>
<td>4</td>
<td>Start-up IPI (SIPI). A SIPI arrived while the logical processor was in the &quot;wait-for-SIPI&quot; state.</td>
</tr>
<tr>
<td>5</td>
<td>I/O system-management interrupt (SMI). An SMI arrived immediately after retirement of an I/O instruction and caused an SMM VM exit (see Section 32.15.2).</td>
</tr>
<tr>
<td>6</td>
<td>Other SMI. An SMI arrived and caused an SMM VM exit (see Section 32.15.2) but not immediately after retirement of an I/O instruction.</td>
</tr>
<tr>
<td>7</td>
<td>Interrupt window. At the beginning of an instruction, RFLAGS.IF was 1; events were not blocked by STI or by MOV SS; and the &quot;interrupt-window exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>8</td>
<td>NMI window. At the beginning of an instruction, there was no virtual-NMI blocking; events were not blocked by MOV SS; and the &quot;NMI-window exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>9</td>
<td>Task switch. Guest software attempted a task switch.</td>
</tr>
<tr>
<td>10</td>
<td>CPUID. Guest software attempted to execute CPUID.</td>
</tr>
<tr>
<td>11</td>
<td>GETSEC. Guest software attempted to execute GETSEC.</td>
</tr>
<tr>
<td>12</td>
<td>HLT. Guest software attempted to execute HLT and the &quot;HLT exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>13</td>
<td>INVD. Guest software attempted to execute INVD.</td>
</tr>
<tr>
<td>14</td>
<td>INVLPG. Guest software attempted to execute INVLPG and the &quot;INVLPG exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>15</td>
<td>RDPMC. Guest software attempted to execute RDPMC and the &quot;RDPMC exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>16</td>
<td>RDTSC. Guest software attempted to execute RDTSC and the &quot;RDTSC exiting&quot; VM-execution control was 1.</td>
</tr>
<tr>
<td>17</td>
<td>RSM. Guest software attempted to execute RSM in SMM.</td>
</tr>
<tr>
<td>18</td>
<td>VMCALL. VMCALL was executed either by guest software (causing an ordinary VM exit) or by the executive monitor (causing an SMM VM exit; see Section 32.15.2).</td>
</tr>
<tr>
<td>19</td>
<td>VMCLEAR. Guest software attempted to execute VMCLEAR.</td>
</tr>
<tr>
<td>20</td>
<td>VMLAUNCH. Guest software attempted to execute VMLAUNCH.</td>
</tr>
<tr>
<td>21</td>
<td>VMPTRLD. Guest software attempted to execute VMPTRLD.</td>
</tr>
<tr>
<td>22</td>
<td>VMPTRST. Guest software attempted to execute VMPTRST.</td>
</tr>
<tr>
<td>Basic Exit Reason</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>23</td>
<td>VMREAD. Guest software attempted to execute VMREAD.</td>
</tr>
<tr>
<td>24</td>
<td>VMRESUME. Guest software attempted to execute VMRESUME.</td>
</tr>
<tr>
<td>25</td>
<td>VMWRITE. Guest software attempted to execute VMWRITE.</td>
</tr>
<tr>
<td>26</td>
<td>VMXOFF. Guest software attempted to execute VMXOFF.</td>
</tr>
<tr>
<td>27</td>
<td>VMXON. Guest software attempted to execute VMXON.</td>
</tr>
<tr>
<td>28</td>
<td>Control-register accesses. Guest software attempted to access CR0, CR3, CR4, or CR8 using CLTS, LMSW, or MOV CR and the VM-execution control fields indicate that a VM exit should occur (see Section 26.1 for details). This basic exit reason is not used for trap-like VM exits following executions of the MOV to CR8 instruction when the “use TPR shadow” VM-execution control is 1. Such VM exits instead use basic exit reason 43.</td>
</tr>
<tr>
<td>29</td>
<td>MOV DR. Guest software attempted a MOV to or from a debug register and the “MOV-DR exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>30</td>
<td>I/O instruction. Guest software attempted to execute an I/O instruction and either: 1: The “use I/O bitmaps” VM-execution control was 0 and the “unconditional I/O exiting” VM-execution control was 1. 2: The “use I/O bitmaps” VM-execution control was 1 and a bit in the I/O bitmap associated with one of the ports accessed by the I/O instruction was 1.</td>
</tr>
<tr>
<td>31</td>
<td>RDMSR. Guest software attempted to execute RDMSR and either: 1: The “use MSR bitmaps” VM-execution control was 0. 2: The value of RCX was neither in the range 00000000H – 00001FFFH nor in the range C0000000H – C0001FFFH. 3: The value of RCX was in the range 00000000H – 00001FFFH and the n\textsuperscript{th} bit in read bitmap for low MSRs is 1, where n was the value of RCX. 4: The value of RCX is in the range C0000000H – C0001FFFH and the n\textsuperscript{th} bit in read bitmap for high MSRs is 1, where n is the value of RCX &amp; 00001FFFH.</td>
</tr>
<tr>
<td>32</td>
<td>WRMSR. Guest software attempted to execute WRMSR and either: 1: The “use MSR bitmaps” VM-execution control was 0. 2: The value of RCX was neither in the range 00000000H – 00001FFFH nor in the range C0000000H – C0001FFFH. 3: The value of RCX was in the range 00000000H – 00001FFFH and the n\textsuperscript{th} bit in write bitmap for low MSRs is 1, where n was the value of RCX. 4: The value of RCX is in the range C0000000H – C0001FFFH and the n\textsuperscript{th} bit in write bitmap for high MSRs is 1, where n is the value of RCX &amp; 00001FFFH.</td>
</tr>
<tr>
<td>33</td>
<td>VM-entry failure due to invalid guest state. A VM entry failed one of the checks identified in Section 27.3.1.</td>
</tr>
<tr>
<td>34</td>
<td>VM-entry failure due to MSR loading. A VM entry failed in an attempt to load MSRs. See Section 27.4.</td>
</tr>
<tr>
<td>36</td>
<td>MWAIT. Guest software attempted to execute MWAIT and the “MWAIT exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>37</td>
<td>Monitor trap flag. A VM exit occurred due to the 1-setting of the “monitor trap flag” VM-execution control (see Section 26.5.2) or VM entry injected a pending MTF VM exit as part of VM entry (see Section 27.6.2).</td>
</tr>
<tr>
<td>39</td>
<td>MONITOR. Guest software attempted to execute MONITOR and the “MONITOR exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>40</td>
<td>PAUSE. Either guest software attempted to execute PAUSE and the “PAUSE exiting” VM-execution control was 1 or the “PAUSE-loop exiting” VM-execution control was 1 and guest software executed a PAUSE loop with execution time exceeding PLE_Window (see Section 26.1.3).</td>
</tr>
<tr>
<td>41</td>
<td>VM-entry failure due to machine-check event. A machine-check event occurred during VM entry (see Section 27.9).</td>
</tr>
<tr>
<td>43</td>
<td>TPR below threshold. The logical processor determined that the value of bits 7:4 of the byte at offset 080H on the virtual-APIC page was below that of the TPR threshold VM-execution control field while the “use TPR shadow” VM-execution control was 1 either as part of TPR virtualization (Section 30.1.2) or VM entry (Section 27.7.7).</td>
</tr>
<tr>
<td>44</td>
<td>APIC access. Guest software attempted to access memory at a physical address on the APIC-access page and the “virtualize APIC accesses” VM-execution control was 1 (see Section 30.4).</td>
</tr>
<tr>
<td>45</td>
<td>Virtualized EOI. EOI virtualization was performed for a virtual interrupt whose vector indexed a bit set in the EOI-exit bitmap.</td>
</tr>
<tr>
<td>Basic Exit Reason</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>46</td>
<td><strong>Access to GDTR or IDTR.</strong> Guest software attempted to execute LGDT, LIDT, SGDT, or SIDT and the “descriptor-table exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>47</td>
<td><strong>Access to LDTR or TR.</strong> Guest software attempted to execute LLDT, LTR, SLDT, or STR and the “descriptor-table exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>48</td>
<td><strong>EPT violation.</strong> An attempt to access memory with a guest-physical address was disallowed by the configuration of the EPT paging structures.</td>
</tr>
<tr>
<td>49</td>
<td><strong>EPT misconfiguration.</strong> An attempt to access memory with a guest-physical address encountered a misconfigured EPT paging-structure entry.</td>
</tr>
<tr>
<td>50</td>
<td><strong>INVEPT.</strong> Guest software attempted to execute INVEPT.</td>
</tr>
<tr>
<td>51</td>
<td><strong>RDTSCP.</strong> Guest software attempted to execute RDTSCP and the “enable RDTSCP” and “RDTSC exiting” VM-execution controls were both 1.</td>
</tr>
<tr>
<td>52</td>
<td><strong>VMX-preemption timer expired.</strong> The preemption timer counted down to zero.</td>
</tr>
<tr>
<td>53</td>
<td><strong>INVVPID.</strong> Guest software attempted to execute INVVPID.</td>
</tr>
<tr>
<td>54</td>
<td><strong>WBINVD or WBNOINVD.</strong> Guest software attempted to execute WBINVD or WBNOINVD and the “WBINVD exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>55</td>
<td><strong>XSETBV.</strong> Guest software attempted to execute XSETBV.</td>
</tr>
<tr>
<td>56</td>
<td><strong>APIC write.</strong> Guest software completed a write to the virtual-APIC page that must be virtualized by VMM software (see Section 30.4.3.3).</td>
</tr>
<tr>
<td>57</td>
<td><strong>RDRAND.</strong> Guest software attempted to execute RDRAND and the “RDRAND exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>58</td>
<td><strong>INVP CID.</strong> Guest software attempted to execute INVP CID and the “enable INVP CID” and “INVLPG exiting” VM-execution controls were both 1.</td>
</tr>
<tr>
<td>59</td>
<td><strong>VMFUNC.</strong> Guest software invoked a VM function with the VMFUNC instruction and the VM function either was not enabled or generated a function-specific condition causing a VM exit.</td>
</tr>
<tr>
<td>60</td>
<td><strong>ENCLS.</strong> Guest software attempted to execute ENCLS, “enable ENCLS exiting” VM-execution control was 1, and either (1) EAX &lt; 63 and the corresponding bit in the ENCLS-exiting bitmap is 1; or (2) EAX ≥ 63 and bit 63 in the ENCLS-exiting bitmap is 1.</td>
</tr>
<tr>
<td>61</td>
<td><strong>RDSEED.</strong> Guest software attempted to execute RDSEED and the “RDSEED exiting” VM-execution control was 1.</td>
</tr>
<tr>
<td>62</td>
<td><strong>Page-modification log full.</strong> The processor attempted to create a page-modification log entry and the value of the PML index was not in the range 0–511.</td>
</tr>
<tr>
<td>63</td>
<td><strong>XSAVE S.</strong> Guest software attempted to execute XSAVE S, the “enable XSAVE S/XSRSTORS” was 1, and a bit was set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.</td>
</tr>
<tr>
<td>64</td>
<td><strong>XSR STORS.</strong> Guest software attempted to execute XSR STORS, the “enable XSAVE S/XSRSTORS” was 1, and a bit was set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.</td>
</tr>
<tr>
<td>65</td>
<td><strong>PCONFIG.</strong> Guest software attempted to execute PCONFIG, “enable PCONFIG” VM-execution control was 1, and either (1) EAX &lt; 63 and the corresponding bit in the PCONFIG-exiting bitmap is 1; or (2) EAX ≥ 63 and bit 63 in the PCONFIG-exiting bitmap is 1.</td>
</tr>
<tr>
<td>66</td>
<td><strong>SPP-related event.</strong> The processor attempted to determine an access’s sub-page write permission and encountered an SPP miss or an SPP misconfiguration. See Section 29.3.4.2.</td>
</tr>
<tr>
<td>67</td>
<td><strong>UMWAIT.</strong> Guest software attempted to execute UMWAIT and the “enable user wait and pause” and “RDTSC exiting” VM-execution controls were both 1.</td>
</tr>
<tr>
<td>68</td>
<td><strong>TPAUSE.</strong> Guest software attempted to execute TPAUSE and the “enable user wait and pause” and “RDTSC exiting” VM-execution controls were both 1.</td>
</tr>
<tr>
<td>69</td>
<td><strong>LOADIWKEY.</strong> Guest software attempted to execute LOADI WKEY and the “LOADIWKEY exiting” VM-execution control was 1.</td>
</tr>
</tbody>
</table>
### Table C-1. Basic Exit Reasons (Contd.)

<table>
<thead>
<tr>
<th>Basic Exit Reason</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td><strong>ENCLV.</strong> Guest software attempted to execute ENCLV, the “enable ENCLV exiting” VM-execution control was 1, and either (1) EAX &lt; 63 and the corresponding bit in the ENCLV-exiting bitmap is 1; or (2) EAX ≥ 63 and bit 63 in the ENCLV-exiting bitmap is 1.</td>
</tr>
<tr>
<td>72</td>
<td><strong>ENQCMD PASID translation failure.</strong> A VM exit occurred during PASID translation because the present bit was clear in a PASID-directory entry, the valid bit was clear in a PASID-table entry, or one of the entries set a reserved bit.</td>
</tr>
<tr>
<td>73</td>
<td><strong>ENQCMDs PASID translation failure.</strong> A VM exit occurred during PASID translation because the present bit was clear in a PASID-directory entry, the valid bit was clear in a PASID-table entry, or one of the entries set a reserved bit.</td>
</tr>
<tr>
<td>74</td>
<td><strong>Bus lock.</strong> The processor asserted a bus lock while the “bus-lock detection” VM-execution control was 1. (Such VM exits will also set bit 26 of the exit-reason field.)</td>
</tr>
<tr>
<td>75</td>
<td><strong>Instruction timeout.</strong> The “instruction timeout” VM-execution control was 1 and certain operations prevented the processor from reaching an instruction boundary within the amount of time specified by the instruction-timeout control.</td>
</tr>
<tr>
<td>76</td>
<td><strong>SEAMCALL.</strong> Guest software attempted to execute SEAMCALL.¹</td>
</tr>
<tr>
<td>77</td>
<td><strong>TDCALL.</strong> Guest software attempted to execute TDCALL.¹</td>
</tr>
</tbody>
</table>

**NOTES:**

NOTE: The Intel 64 and IA-32 Architectures Software Developer’s Manual consists of four volumes: Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384; Model-Specific Registers, Order Number 335592. Refer to all four volumes when evaluating your design needs.
Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.
No product or component can be absolutely secure.
Your costs and results may vary.
You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.
All product plans and roadmaps are subject to change without notice.
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.
Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.
Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.
No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.
© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.17.8</td>
<td>MSRs Specific to the 4th and 5th Generation Intel® Xeon® Scalable Processor Families</td>
<td>2-398</td>
</tr>
<tr>
<td>2.17.9</td>
<td>MSRs Introduced in the Intel® Core™ Ultra 7 Processor Supporting Performance Hybrid Architecture</td>
<td>2-409</td>
</tr>
<tr>
<td>2.18</td>
<td>MSRs in the Intel® Xeon® Phi™ Processor 3200/5200/7200 Series and the Intel® Xeon Phi™ Processor 7215/7285/7295 Series</td>
<td>2-432</td>
</tr>
<tr>
<td>2.19</td>
<td>MSRs in the Pentium® 4 and Intel® Xeon® Processors.</td>
<td>2-453</td>
</tr>
<tr>
<td>2.19.1</td>
<td>MSRs Unique to Intel® Xeon® Processor MP with L3 Cache.</td>
<td>2-479</td>
</tr>
<tr>
<td>2.20</td>
<td>MSRs in Intel® Core™ Solo and Intel® Core™ Duo Processors.</td>
<td>2-480</td>
</tr>
<tr>
<td>2.21</td>
<td>MSRs in the Pentium M Processor.</td>
<td>2-491</td>
</tr>
<tr>
<td>2.22</td>
<td>MSRs in the P6 Family Processors.</td>
<td>2-498</td>
</tr>
<tr>
<td>2.23</td>
<td>MSRs in Pentium Processors.</td>
<td>2-508</td>
</tr>
</tbody>
</table>
FIGURES

Figure 1-1. Bit and Byte Order ................................................................. 1-5
Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation ..................... 1-7
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2-1</td>
<td>CPUID Signature Values of DisplayFamily_DisplayModel</td>
<td>2-1</td>
</tr>
<tr>
<td>Table 2-2</td>
<td>IA-32 Architectural MSRs</td>
<td>2-3</td>
</tr>
<tr>
<td>Table 2-3</td>
<td>MSRs in Processors Based on Intel® Core™ Microarchitecture</td>
<td>2-65</td>
</tr>
<tr>
<td>Table 2-4</td>
<td>MSRs in the 45 nm and 32 nm Intel Atom® Processor Family</td>
<td>2-81</td>
</tr>
<tr>
<td>Table 2-5</td>
<td>MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_27H</td>
<td>2-93</td>
</tr>
<tr>
<td>Table 2-6</td>
<td>MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures)</td>
<td>2-94</td>
</tr>
<tr>
<td>Table 2-7</td>
<td>Specific MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_37H, 06_4AH, 06_5AH, or 06_5DH</td>
<td>2-105</td>
</tr>
<tr>
<td>Table 2-8</td>
<td>Specific MSRs Supported by the Intel Atom® Processor E3000 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_37H</td>
<td>2-110</td>
</tr>
<tr>
<td>Table 2-9</td>
<td>Specific MSRs Supported by Intel Atom® Processor C2000 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4DH</td>
<td>2-112</td>
</tr>
<tr>
<td>Table 2-10</td>
<td>Specific MSRs Supported by Intel® Xeon® Processor E7 Family</td>
<td>2-114</td>
</tr>
<tr>
<td>Table 2-11</td>
<td>Specific MSRs Supported by Intel® Xeon® Processor E5 Family</td>
<td>2-120</td>
</tr>
<tr>
<td>Table 2-12</td>
<td>Additional MSRs Supported by Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)</td>
<td>2-225</td>
</tr>
<tr>
<td>Table 2-13</td>
<td>Additional MSRs Supported by the Intel® Xeon® Processor D v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH</td>
<td>2-250</td>
</tr>
<tr>
<td>Table 2-14</td>
<td>Additional MSRs Supported by Intel® Xeon® Processor E5 v2 and E7 v2 Families</td>
<td>2-254</td>
</tr>
<tr>
<td>Table 2-15</td>
<td>Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures</td>
<td>2-259</td>
</tr>
<tr>
<td>Table 2-16</td>
<td>Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH</td>
<td>2-264</td>
</tr>
<tr>
<td>Table 2-17</td>
<td>Additional MSRs Supported by the 4th Generation Intel® Core™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_5EH</td>
<td>2-275</td>
</tr>
<tr>
<td>Table 2-18</td>
<td>Additional MSRs Supported by the 5th Generation Intel® Core™ Processors and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture</td>
<td>2-309</td>
</tr>
<tr>
<td>Table 2-19</td>
<td>Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_5FH</td>
<td>2-320</td>
</tr>
<tr>
<td>Table 2-20</td>
<td>Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH</td>
<td>2-323</td>
</tr>
<tr>
<td>Table 2-21</td>
<td>Uncore PMU MSRs Supported by 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors</td>
<td>2-331</td>
</tr>
<tr>
<td>Table 2-22</td>
<td>Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors</td>
<td>2-352</td>
</tr>
<tr>
<td>Table 2-23</td>
<td>Additional MSRs Supported by the 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture</td>
<td>2-354</td>
</tr>
<tr>
<td>Table 2-24</td>
<td>Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture</td>
<td>2-356</td>
</tr>
<tr>
<td>Table 2-25</td>
<td>Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture</td>
<td>2-366</td>
</tr>
<tr>
<td>Table 2-26</td>
<td>Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture</td>
<td>2-370</td>
</tr>
</tbody>
</table>
Table 2-47. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor P-core ...................................................... 2-374
Table 2-48. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor E-core ...................................................... 2-375
Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors .............................................. 2-376
Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H .................................................................................. 2-379
Table 2-51. MSRs Supported by the 3rd Generation Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_6AH or 06_6CH .................................................................. 2-396
Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) ...................................................... 2-398
Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture 2-409
Table 2-54. MSRs Supported by the Intel® Core™ Ultra 7 Processor P-core ................................................................................. 2-430
Table 2-55. MSRs Supported by the Intel® Core™ Ultra 7 Processor E-core ................................................................................. 2-431
Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H .................................................................................. 2-432
Table 2-57. Additional MSRs Supported by the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_85H .................................................................................. 2-451
Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors ................................................................................................. 2-453
Table 2-59. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache ........................................................ 2-479
Table 2-60. MSRs Unique to Intel® Xeon® Processor 7100 Series ................................................................................................. 2-480
Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV ......................... 2-480
Table 2-62. MSRs in Pentium M Processors .......................................................................................................................... 2-491
Table 2-63. MSRs in the P6 Family Processors ........................................................................................................................ 2-498
Table 2-64. MSRs in the Pentium Processor .......................................................................................................................... 2-509
CHAPTER 1
ABOUT THIS MANUAL

The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4: Model-Specific Registers (order number 335592) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

• Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture (order number 253665).

The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A, 3B, 3C, & 3D, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, and the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, address the programming environment for classes of software that host operating systems. The Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4, describes the model-specific registers of Intel 64 and IA-32 processors.

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

• Pentium® processors
• P6 family processors
• Pentium® 4 processors
• Pentium® M processors
• Intel® Xeon® processors
• Pentium® D processors
• Pentium® processor Extreme Editions
• 64-bit Intel® Xeon® processors
• Intel® Core™ Duo processor
• Intel® Core™ Solo processor
• Dual-Core Intel® Xeon® processor LV
• Intel® Core™ 2 Duo processor
• Intel® Core™ 2 Quad processor Q6000 series
• Intel® Xeon® processor 3000, 3200 series
• Intel® Xeon® processor 5000 series
• Intel® Xeon® processor 5100, 5300 series
• Intel® Core™ 2 Extreme processor X7000 and X6800 series
• Intel® Core™ 2 Extreme QX6000 series
• Intel® Xeon® processor 7100 series
• Intel® Pentium® Dual-Core processor
• Intel® Xeon® processor 7200, 7300 series
• Intel® Core™ 2 Extreme QX9000 series
• Intel® Xeon® processor 5200, 5400, 7400 series
• Intel® Core™ 2 Extreme processor QX9000 and X9000 series
• Intel® Core™ 2 Quad processor Q9000 series
• Intel® Core™ 2 Duo processor E8000, T9000 series
• Intel® Core™ i7 processor
• Intel® Core™ i5 processor
• Intel® Xeon® processor E7-8800/4800/2800 product families
• Intel® Core™ i7-3930K processor
• 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
• Intel® Xeon® processor E3-1200 product family
• Intel® Xeon® processor E5-2400/1400 product family
• Intel® Xeon® processor E5-4600/2600/1600 product family
• 3rd generation Intel® Core™ processors
• Intel® Xeon® processor E3-1200 v2 product family
• Intel® Xeon® processor E5-2400/1400 v2 product families
• Intel® Xeon® processor E5-4600/2600/1600 v2 product families
• Intel® Xeon® processor E7-8800/4800/2800 v2 product families
• 4th generation Intel® Core™ processors
• The Intel® Core™ M processor family
• Intel® Core™ i7-5xxx Processor Extreme Edition
• Intel® Core™ i7-4xxx Processor Extreme Edition
• Intel® Xeon® processor E3-1200 v3 product family
• Intel® Xeon® processor E5-2600/1600 v3 product families
• 5th generation Intel® Core™ processors
• Intel® Xeon® processor D-1500 product family
• Intel® Xeon® processor E5 v4 family
• Intel Atom® processor X7-Z8000 and X5-Z8000 series
• Intel Atom® processor Z3400 series
• Intel Atom® processor Z3500 series
• 6th generation Intel® Core™ processors
• Intel® Xeon® processor E3-1500m v5 product family
• 7th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
• Intel® Xeon® Scalable Processor Family
• 8th generation Intel® Core™ processors
• Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series
• Intel® Xeon® E processors
• 9th generation Intel® Core™ processors
• 2nd generation Intel® Xeon® Scalable Processor Family
• 10th generation Intel® Core™ processors
• 11th generation Intel® Core™ processors
• 3rd generation Intel® Xeon® Scalable Processor Family
• 12th generation Intel® Core™ processors
• 13th generation Intel® Core™ processors
• 4th generation Intel® Xeon® Scalable Processor Family
• 5th generation Intel® Xeon® Scalable Processor Family
• Intel® Core™ Ultra 7 processors

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.


The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™ 2 Duo, Intel® Core™ 2 Quad, and Intel® Core™ 2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™ 2 Quad processor Q9000 series, and Intel® Core™ 2 Extreme processors QX9000, X9000 series, Intel® Core™ 2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.


P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™ 2 Duo, Intel® Core™ 2 Extreme, Intel® Core™ 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Nehalem microarchitecture. Westmere microarchitecture is a 32 nm version of the Nehalem microarchitecture. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on the Westmere microarchitecture. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Sandy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Ivy Bridge-E microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Haswell-E microarchitecture and support Intel 64 architecture.

The Intel Atom® processor Z8000 series is based on the Airmont microarchitecture.
The Intel Atom® processor Z3400 series and the Intel Atom® processor Z3500 series are based on the Silvermont microarchitecture.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Broadwell microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Scalable Processor Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon® Scalable Processor Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Skylake microarchitecture and support Intel 64 architecture.

The Intel® Atom® processor C series, the Intel Atom® processor X series, the Intel® Pentium® processor J series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont microarchitecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Knights Landing microarchitecture and supports Intel 64 architecture.

The Intel® Pentium® Silver processor series, the Intel® Celeron® processor J series, and the Intel® Celeron® processor N series are based on the Goldmont Plus microarchitecture.

The 8th generation Intel® Core™ processors, 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on the Coffee Lake microarchitecture and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series is based on the Knights Mill microarchitecture and supports Intel 64 architecture.

The 2nd generation Intel® Xeon® Scalable Processor Family is based on the Cascade Lake product and supports Intel 64 architecture.

Some 10th generation Intel® Core™ processors are based on the Ice Lake microarchitecture, and some are based on the Comet Lake microarchitecture; both support Intel 64 architecture.

Some 11th generation Intel® Core™ processors are based on the Tiger Lake microarchitecture, and some are based on the Rocket Lake microarchitecture; both support Intel 64 architecture.

Some 3rd generation Intel® Xeon® Scalable Processor Family processors are based on the Cooper Lake product, and some are based on the Ice Lake microarchitecture; both support Intel 64 architecture.

The 12th generation Intel® Core™ processors are based on the Alder Lake performance hybrid architecture and support Intel 64 architecture.

The 13th generation Intel® Core™ processors are based on the Raptor Lake performance hybrid architecture and support Intel 64 architecture.

The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture and supports Intel 64 architecture.

The 5th generation Intel® Xeon® Scalable Processor Family is based on Emerald Rapids microarchitecture and supports Intel 64 architecture.

The Intel® Core™ Ultra 7 processor is based on Meteor Lake hybrid architecture and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

1.2 OVERVIEW OF THE MODEL-SPECIFIC REGISTERS

A description of this manual’s content follows:

Chapter 1 — About This Manual. Gives an overview of all volumes of the Intel® 64 and IA-32 Architectures Software Developer’s Manual. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.
Chapter 2 — Model-Specific Registers (MSRs). Lists the MSRs available in Intel processors, and describes their functions.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as reserved. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

Figure 1-1. Bit and Byte Order
### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:
- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction:

```
CS:EIP
```
1.3.6 Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a single syntax to represent this type of information. See Figure 1-2.

![Syntax for CPUID, CR, and MSR Data Presentation](image)

1.3.7 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```
1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

See also:

- The latest security information on Intel® products:
- Software developer resources, guidance, and insights for security advisories:
  https://software.intel.com/security-software-guidance/
- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
- Intel® Fortran Compiler documentation and online help:
- Intel® Software Development Tools:
- Intel® 64 and IA-32 Architectures Software Developer’s Manual (in one, four, or ten volumes):
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:
- Intel® Software Guard Extensions (Intel® SGX) Information:
- Developing Multi-threaded Applications: A Platform Consistent Approach:
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide:
  http://software.intel.com/file/30388

Literature related to select features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference:

More relevant links are:

- Intel® Developer Zone:
  https://software.intel.com/en-us
- Developer centers:
- Processor support general link:
  http://www.intel.com/support/processors/
- Intel® Hyper-Threading Technology (Intel® HT Technology):
This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions. The scope of an MSR defines the set of processors that access the same MSR with RDMSR and WRMSR. Thread-scope MSRs are unique to every logical processor. Core-scope MSRs are shared by the threads in the same core; similarly for module-scope, die-scope, and package-scope.

When a processor package contains a single die, die-scope and package-scope are synonymous. When a package contains multiple die, they are distinct.

NOTE

For information on hierarchical level types supported, refer to the CPUID Leaf 1FH definition for the actual level type numbers: “V2 Extended Topology Enumeration Leaf” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. Also see Section 9.9.1, “Hierarchical Mapping of Shared Resources,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A). Table 2-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Processor Families/Processor Number Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_85H</td>
<td>Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series based on Knights Mill microarchitecture</td>
</tr>
<tr>
<td>06_57H</td>
<td>Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture</td>
</tr>
<tr>
<td>06_AAH</td>
<td>Intel® Core™ Ultra 7 processors supporting Meteor Lake performance hybrid architecture</td>
</tr>
<tr>
<td>06_CFH</td>
<td>5th generation Intel® Xeon® Scalable Processor Family based on Emerald Rapids microarchitecture</td>
</tr>
<tr>
<td>06_8FH</td>
<td>4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture</td>
</tr>
<tr>
<td>06_BAH, 06_B7H, 06_BFH</td>
<td>13th generation Intel® Core™ processors supporting Raptor Lake performance hybrid architecture</td>
</tr>
<tr>
<td>06_97H, 06_9AH</td>
<td>12th generation Intel® Core™ processors supporting Alder Lake performance hybrid architecture</td>
</tr>
<tr>
<td>06_8CH, 06_8DH</td>
<td>11th generation Intel® Core™ processors based on Tiger Lake microarchitecture</td>
</tr>
<tr>
<td>06_A7H</td>
<td>11th generation Intel® Core™ processors based on Rocket Lake microarchitecture</td>
</tr>
<tr>
<td>06_7DH, 06_7EH</td>
<td>10th generation Intel® Core™ processors based on Ice Lake microarchitecture</td>
</tr>
<tr>
<td>06_A5H, 06_A6H</td>
<td>10th generation Intel® Core™ processors based on Comet Lake microarchitecture</td>
</tr>
<tr>
<td>06_66H</td>
<td>Intel® Core™ processors based on Cannon Lake microarchitecture</td>
</tr>
<tr>
<td>06_8EH, 06_9EH</td>
<td>7th generation Intel® Core™ processors based on Kaby Lake microarchitecture, 8th and 9th generation Intel® Core™ processors based on Coffee Lake microarchitecture, Intel® Xeon® E processors based on Coffee Lake microarchitecture</td>
</tr>
<tr>
<td>06_6AH, 06_6CH</td>
<td>3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture</td>
</tr>
<tr>
<td>06_55H</td>
<td>Intel® Xeon® Scalable Processor Family based on Skylake microarchitecture, 2nd generation Intel® Xeon® Scalable Processor Family based on Cascade Lake product, and 3rd generation Intel® Xeon® Scalable Processor Family based on Cooper Lake product</td>
</tr>
</tbody>
</table>
### Table 2-1: CPUID Signature Values of DisplayFamily_DisplayModel (Contd.)

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel</th>
<th>Processor Families/Processor Number Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>06_4EH, 06_5EH</td>
<td>6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture</td>
</tr>
<tr>
<td>06_56H</td>
<td>Intel Xeon processor D-1500 product family based on Broadwell microarchitecture</td>
</tr>
<tr>
<td>06_4FH</td>
<td>Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition</td>
</tr>
<tr>
<td>06_47H</td>
<td>5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture</td>
</tr>
<tr>
<td>06_3DH</td>
<td>Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture</td>
</tr>
<tr>
<td>06_3FH</td>
<td>Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition</td>
</tr>
<tr>
<td>06_3CH, 06_45H, 06_46H</td>
<td>4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture</td>
</tr>
<tr>
<td>06_3EH</td>
<td>Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture</td>
</tr>
<tr>
<td>06_3EH</td>
<td>Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition</td>
</tr>
<tr>
<td>06_3AH</td>
<td>3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture</td>
</tr>
<tr>
<td>06_2DH</td>
<td>Intel Xeon processor E5 Family based on Sandy Bridge microarchitecture, Intel Core i7-39xx Processor Extreme Edition</td>
</tr>
<tr>
<td>06_2FH</td>
<td>Intel Xeon Processor E7 Family</td>
</tr>
<tr>
<td>06_2AH</td>
<td>Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series</td>
</tr>
<tr>
<td>06_2EH</td>
<td>Intel Xeon processor 7500, 6500 series</td>
</tr>
<tr>
<td>06_25H, 06_2CH</td>
<td>Intel Xeon processors 3600, 5600 series, Intel Core i7, i5, and i3 Processors</td>
</tr>
<tr>
<td>06_1EH, 06_1FH</td>
<td>Intel Core i7 and i5 Processors</td>
</tr>
<tr>
<td>06_1AH</td>
<td>Intel Core i7 Processor, Intel Xeon processor 3400, 3500, 5500 series</td>
</tr>
<tr>
<td>06_1DH</td>
<td>Intel Xeon processor MP 7400 series</td>
</tr>
<tr>
<td>06_17H</td>
<td>Intel Xeon processor 3100, 3300, 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series</td>
</tr>
<tr>
<td>06_0FH</td>
<td>Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors</td>
</tr>
<tr>
<td>06_0EH</td>
<td>Intel Core Duo, Intel Core Solo processors</td>
</tr>
<tr>
<td>06_0DH</td>
<td>Intel Pentium M processor</td>
</tr>
<tr>
<td>06_6EH, 06_96H, 06_9CH</td>
<td>Intel Atom® processors, Intel® Celeron® processors, Intel® Pentium® processors, and Intel® Pentium® Silver processors based on Tremont Microarchitecture</td>
</tr>
<tr>
<td>06_7AH</td>
<td>Intel Atom processors based on Goldmont Plus microarchitecture</td>
</tr>
<tr>
<td>06_5FH</td>
<td>Intel Atom processors based on Goldmont microarchitecture (Denverton)</td>
</tr>
<tr>
<td>06_5CH</td>
<td>Intel Atom processors based on Goldmont microarchitecture</td>
</tr>
<tr>
<td>06_4CH</td>
<td>Intel Atom processor X7-Z8000 and X5-Z8000 series based on Airmont microarchitecture</td>
</tr>
<tr>
<td>06_5DH</td>
<td>Intel Atom processor X3-C3000 based on Silvermont microarchitecture</td>
</tr>
<tr>
<td>06_5AH</td>
<td>Intel Atom processor Z3500 series</td>
</tr>
<tr>
<td>06_4AH</td>
<td>Intel Atom processor Z3400 series</td>
</tr>
</tbody>
</table>
2.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these “architectural MSRs” were given the prefix “IA32_”. Table 2-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 2-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific. Code that accesses a model-specific MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 2-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of “DF_DM” (see Table 2-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as “MAXPHYADDR” in Table 2-2. “MAXPHYADDR” is reported by CPUID.8000_0008H leaf.

MSR address range between 40000000H - 4000FFFFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

### Table 2-2. IA-32 Architectural MSRs

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: OH, 0</td>
<td>IA32_P5_MC_ADDR (P5_MC_ADDR)</td>
<td></td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>IA32_P5_MC_TYPE (P5_MC_TYPE)</td>
<td>Pentium Processor (05_01H)</td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily_DisplayModel = 05_09H and SteppingID = 0
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 6H, 6</td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td>59:0</td>
<td>Reserved.</td>
<td>0F_03H</td>
</tr>
<tr>
<td>See Section 9.10.5, “Monitor/Mwait Address Range Determination.”</td>
<td></td>
<td>49:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER (TSC)</td>
<td>52:50</td>
<td>Platform ID (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Section 18.17, “Time-Stamp Counter.”</td>
<td></td>
<td>49:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>IA32_PLATFORM_ID (MSR_PLATFORM_ID)</td>
<td>63:53</td>
<td>Reserved.</td>
<td>06_01H</td>
</tr>
<tr>
<td>Platform ID (R/O) The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.</td>
<td></td>
<td>49:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>52:50</td>
<td>Platform ID (R/O)</td>
<td>52 51 50</td>
<td>Contains information concerning the intended platform for the processor.</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>Processor Flag 0</td>
<td>0 0 1</td>
<td>Processor Flag 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>Processor Flag 2</td>
<td>0 1 1</td>
<td>Processor Flag 3</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>Processor Flag 4</td>
<td>1 0 1</td>
<td>Processor Flag 5</td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>Processor Flag 6</td>
<td>1 1 1</td>
<td>Processor Flag 7</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>Processor Flag 7</td>
<td>63:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1BH, 27</td>
<td>IA32_APIC_BASE (APIC_BASE)</td>
<td>7:0</td>
<td>Reserved.</td>
<td>06_01H</td>
</tr>
<tr>
<td>This register holds the APIC base address, permitting the relocation of the APIC memory map. See Section 11.4.4, “Local APIC Status and Location,” and Section 11.4.5, “Relocating the Local APIC Registers.”</td>
<td></td>
<td>8</td>
<td>BSP Flag (R/W)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td>10</td>
<td>Enable x2APIC mode.</td>
<td>06_1AH</td>
</tr>
<tr>
<td>11</td>
<td>APIC Global Enable (R/W)</td>
<td>(MAXPHYADDR -1):12</td>
<td>APIC Base (R/W)</td>
<td></td>
</tr>
<tr>
<td>63: MAXPHYADDR</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td>Control Features in Intel 64 Processor (R/W)</td>
<td>If any one enumeration condition for defined bit field holds.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lock bit (R/WO): (1 = locked). When set, locks this MSR from being written; writes to this bit will result in GP(0). Note: Once the Lock bit is set, the contents of this register cannot be modified. Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS. Hence, once the Lock bit is set, the entire IA32_FEATURE_CONTROL contents are preserved across RESET when PWRGOOD is not deasserted.</td>
<td></td>
<td>If any one enumeration condition for defined bit field position greater than bit 0 holds.</td>
</tr>
<tr>
<td>1</td>
<td>Enable VMX inside SMX operation (R/WL) This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).</td>
<td></td>
<td>If CPUID.01H:ECX[5] = 1 &amp;&amp; CPUID.01H:ECX[6] = 1</td>
</tr>
<tr>
<td>2</td>
<td>Enable VMX outside SMX operation (R/WL) This bit enables VMX for a system executive that does not require SMX. BIOS must set this bit only when the CPUID function 1 returns the VMX feature flag set (ECX bit 5).</td>
<td></td>
<td>If CPUID.01H:ECX[5] = 1</td>
</tr>
<tr>
<td>7:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>SENTER Local Function Enables (R/WL) When set, each bit in the field represents an enable control for a corresponding SENTER function. This field is supported only if CPUID.1:ECX.[bit 6] is set.</td>
<td></td>
<td>If CPUID.01H:ECX[6] = 1</td>
</tr>
<tr>
<td>15</td>
<td>SENTER Global Enable (R/WL) This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set.</td>
<td></td>
<td>If CPUID.01H:ECX[6] = 1</td>
</tr>
<tr>
<td>16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>SGX Launch Control Enable (R/WL) This bit must be set to enable runtime re-configuration of SGX Launch Control via the IA32_SGXLEPUKBKEYHASHn MSR.</td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1</td>
</tr>
<tr>
<td>18</td>
<td>SGX Global Enable (R/WL) This bit must be set to enable SGX leaf functions.</td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1</td>
</tr>
<tr>
<td>19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>LMCE On (R/WL) When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.</td>
<td></td>
<td>If IA32_MCG_CAP[27] = 1</td>
</tr>
<tr>
<td>63:21</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 3BH, 59 IA32_TSC_ADJUST</td>
<td>Per Logical Processor TSC Adjust (R/Write to clear)</td>
<td>If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>THREAD_ADJUST</td>
<td>Local offset value of the IA32_TSC for a logical processor. Reset value is zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.</td>
</tr>
</tbody>
</table>
## Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 48H, 72</td>
<td>IA32_SPEC_CTRL</td>
<td>0</td>
<td>Indirect Branch Restricted Speculation (IBRS). Restricts speculation of indirect branch.</td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[26]=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Single Thread Indirect Branch Predictors (STIBP). Prevents indirect branch predictions on all logical processors on the core from being controlled by any sibling logical processor in the same core.</td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[27]=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Speculative Store Bypass Disable (SSBD) delays speculative execution of a load until the addresses for all older stores are known.</td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[31]=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>IPRED_DIS_U</td>
<td>If 1, enables IPRED_DIS control for CPL3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>IPRED_DIS_S</td>
<td>If 1, enables IPRED_DIS control for CPL0/1/2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>RRSBA_DIS_U</td>
<td>If 1, disables RRSBA behavior for CPL3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>RRSBA_DIS_S</td>
<td>If 1, disables RRSBA behavior for CPL0/1/2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>PSFD</td>
<td>If 1, disables Fast Store Forwarding Predictor. Note that setting bit 2 (SSBD) also disables this.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>DDPD_U</td>
<td>If 1, disables the Data Dependent Prefetcher that examines data values in memory while CPL = 3. Note that setting bit 2 (SSBD) also disables this.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>BHI_DIS_S</td>
<td>When '1, enables BHI_DIS_S behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:11</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 49H, 73</td>
<td>IA32_PRED_CMD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction Command (wO)</td>
<td>Gives software a way to issue commands that affect the state of predictors.</td>
<td>If any one of the enumeration conditions for defined bit field positions holds.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Indirect Branch Prediction Barrier (IBPB)</td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[26]=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4EH, 78</td>
<td>IA32_PPIN_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protected Processor Inventory Number Enable Control (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=01H):EBX[0]=1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Model-Specific Registers (MSRs)**

Register Address: 48H, 72

Speculation Control (R/W)

The MSR bits are defined as logical processor scope. On some core implementations, the bits may impact sibling logical processors on the same core.

This MSR has a value of 0 after reset and is unaffected by INIT# or SIPI#.

- **0**: Indirect Branch Restricted Speculation (IBRS). Restricts speculation of indirect branch.
- **1**: Single Thread Indirect Branch Predictors (STIBP). Prevents indirect branch predictions on all logical processors on the core from being controlled by any sibling logical processor in the same core.
- **2**: Speculative Store Bypass Disable (SSBD) delays speculative execution of a load until the addresses for all older stores are known.
- **3**: IPRED_DIS_U
  - If 1, enables IPRED_DIS control for CPL3.
- **4**: IPRED_DIS_S
  - If 1, enables IPRED_DIS control for CPL0/1/2.
- **5**: RRSBA_DIS_U
  - If 1, disables RRSBA behavior for CPL3.
- **6**: RRSBA_DIS_S
  - If 1, disables RRSBA behavior for CPL0/1/2.
- **7**: PSFD
  - If 1, disables Fast Store Forwarding Predictor. Note that setting bit 2 (SSBD) also disables this.
- **8**: DDPD_U
  - If 1, disables the Data Dependent Prefetcher that examines data values in memory while CPL = 3. Note that setting bit 2 (SSBD) also disables this.
- **9**: Reserved.
- **10**: BHI_DIS_S
  - When '1, enables BHI_DIS_S behavior.
- **63:11**: Reserved.
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LockOut (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If 0, indicates that further writes to IA32_PPIN_CTL is allowed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If 1, indicates that further writes to IA32_PPIN_CTL is disallowed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Writing 1 to this bit is only permitted if the Enable_PPIN bit is clear.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The Privileged System Software Inventory Agent should read IA32_PPIN_CTL[bit 1] to determine if IA32_PPIN is accessible.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The Privileged System Software Inventory Agent is not expected to write to this MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable_PPIN (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If 1, indicates that IA32_PPIN is accessible using RDMSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If 0, indicates that IA32_PPIN is inaccessible using RDMSR. Any attempt to read IA32_PPIN will cause #GP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 4FH, 79 IA32_PPIN

Protected Processor Inventory Number (R/O)

Register Address: 79H, 121 IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)

BIOS Update Trigger (W)

Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 10.11.6, "Microcode Update Loader."

A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.

Register Address: 8BH, 139 IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)

BIOS Update Signature (R/W)

Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.

Register Address: 8CH, 140 IA32_SGXLEPUBKEYHASH0
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_SGXLEPUBKEYHASH[127:64] (R/W)</td>
<td>Bits 127:64 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.</td>
<td>Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.</td>
</tr>
<tr>
<td>IA32_SGXLEPUBKEYHASH[191:128] (R/W)</td>
<td>Bits 191:128 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.</td>
<td>Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.</td>
</tr>
<tr>
<td>IA32_SGXLEPUBKEYHASH[255:192] (R/W)</td>
<td>Bits 255:192 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.</td>
<td>Same comment in MSR listing for IA32_SGXLEPUBKEYHASH0 (MSR address 8CH, 140) applies here.</td>
</tr>
<tr>
<td>0</td>
<td>Valid (R/w)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Controls SMI unblocking by VMXOFF (see Section 32.14.4). If IA32_VMX_MISC[28]</td>
<td></td>
</tr>
<tr>
<td>11:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>31:12</td>
<td>MSEG Base (R/w)</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>IA32_SMBASE</td>
<td>Base address of the logical processor’s SMRAM image (R/O, SMM only). If IA32_VMX_MISC[15]</td>
<td></td>
</tr>
<tr>
<td>IA32_MISC_PACKAGE_CTLS</td>
<td>Power Filtering Control (R/W)</td>
<td>If IA32_ARCH_CAPABILITIES [10] = 1</td>
</tr>
<tr>
<td>0</td>
<td>ENERGY_FILTERING_ENABLE (R/W)</td>
<td>If set, RAPL MSRs report filtered processor power consumption data. This bit can be changed from 0 to 1, but cannot be changed from 1 to 0. After setting, all attempts to clear it are ignored until the next processor reset.</td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>IA32_XAPIC_DISABLE_STATUS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 8DH, 141

Register Address: 8EH, 142

Register Address: 8FH, 143

Register Address: 9BH, 155

Register Address: 9EH, 158

Register Address: BCH, 188

Register Address: BDH, 189
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>xAPIC Disable Status (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[29]=1 and IA32_ARCH_CAPABILITIES [21] = 1</td>
</tr>
<tr>
<td>0</td>
<td>LEGACY_XAPIC_DISABLED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the local APIC is in x2APIC mode (IA32_APIC_BASE.EXTD = 1) and that attempts to clear IA32_APIC_BASE.EXTD will fail (e.g., WRMSR will #GP).</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C1H, 193</td>
<td>IA32_PMC0 (PERFCTR0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 0 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 0</td>
</tr>
<tr>
<td>Register Address: C2H, 194</td>
<td>IA32_PMC1 (PERFCTR1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 1 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 1</td>
</tr>
<tr>
<td>Register Address: C3H, 195</td>
<td>IA32_PMC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 2 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 2</td>
</tr>
<tr>
<td>Register Address: C4H, 196</td>
<td>IA32_PMC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 3 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 3</td>
</tr>
<tr>
<td>Register Address: C5H, 197</td>
<td>IA32_PMC4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 4 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 4</td>
</tr>
<tr>
<td>Register Address: C6H, 198</td>
<td>IA32_PMC5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 5 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 5</td>
</tr>
<tr>
<td>Register Address: C7H, 199</td>
<td>IA32_PMC6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 6 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 6</td>
</tr>
<tr>
<td>Register Address: C8H, 200</td>
<td>IA32_PMC7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Performance Counter 7 (R/W)</td>
<td></td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 7</td>
</tr>
<tr>
<td>Register Address: CFH, 207</td>
<td>IA32_CORE_CAPABILITIES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32 Core Capabilities Register</td>
<td></td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[30] = 1</td>
</tr>
<tr>
<td>63:0</td>
<td>Reserved.</td>
<td></td>
<td>No architecturally defined bits.</td>
</tr>
<tr>
<td>Register Address: E1H, 225</td>
<td>IA32_UMWAIT_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UMWAIT Control (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>C0.2 is not allowed by the OS. Value of “1” means all C0.2 requests revert to C0.1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:2</td>
<td>Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7H, 231</td>
<td>IA32_MPERF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSC Frequency Clock Counter (R/Write to clear)</td>
<td></td>
<td></td>
<td>If CPUID.06H: ECX[0] = 1</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>C0_MCNT: C0 TSC Frequency Clock Count</td>
<td>IA32_APERF</td>
<td>Clear upon overflow / wrap-around of IA32_APERF.</td>
</tr>
<tr>
<td></td>
<td>Increments at fixed interval (relative to TSC freq.) when the logical processor is in C0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cleared upon overflow / wrap-around of IA32_APERF.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E8H, 232</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual Performance Clock Counter (R/Write to clear)</td>
<td></td>
<td></td>
<td>If CPUID.06H: ECX[0] = 1</td>
</tr>
<tr>
<td>63:0</td>
<td>C0_ACNT: C0 Actual Frequency Clock Count</td>
<td>IA32_MTRRCAP (MTRRcap)</td>
<td>Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in C0.</td>
</tr>
<tr>
<td></td>
<td>Cleared upon overflow / wrap-around of IA32_MPERF.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRR Capability (R/O)</td>
<td>See Section 12.11.2.1, &quot;IA32_MTRR_DEF_TYPE MSR.&quot;</td>
<td></td>
<td>06_01H</td>
</tr>
<tr>
<td>7:0</td>
<td>VCNT: The number of variable memory type ranges in the processor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Fixed range MTRRs are supported when set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>WC Supported when set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>SMRR Supported when set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>PRMRR supported when set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 10AH, 266</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enumeration of Architectural Features (R/O)</td>
<td>IA32_ARCH_CAPABILITIES</td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[29]=1</td>
</tr>
<tr>
<td>0</td>
<td>RDCL_NO: The processor is not susceptible to Rogue Data Cache Load (RDCL).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>IBRS_ALL: The processor supports enhanced IBRS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RSBA: The processor supports RSB Alternate. Alternative branch predictors may be used by RET instructions when the RSB is empty. SW using retpoline may be affected by this behavior.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SKIP_L1DFL_VMENTRY: A value of 1 indicates the hypervisor need not flush the L1D on VM entry.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SSB_NO: Processor is not susceptible to Speculative Store Bypass.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MDS_NO: Processor is not susceptible to Microarchitectural Data Sampling (MDS).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>IF_PSCCHANGE_MC_NO: The processor is not susceptible to a machine check error due to modifying the size of a code page without TLB invalidation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>TSX_CTRL: If 1, indicates presence of IA32_TSX_CTRL MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>TAA_NO: If 1, processor is not affected by TAA.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MCU_CONTROL: If 1, the processor supports the IA32_MCU_CONTROL MSR.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>MISC_PACKAGE_CTLS: The processor supports IA32_MISC_PACKAGE_CTLS MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ENERGY_FILTERING_CTL: The processor supports setting and reading the IA32_MISC_PACKAGE_CTLS[0] (ENERGY_FILTERING_ENABLE) bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>DOITM: If 1, the processor supports Data Operand Independent Timing Mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SBDR_SSDP_NO: The processor is not affected by either the Shared Buffers Data Read (SBDR) vulnerability or the Sideband Stale Data Propagator (SSDP).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>FBSDP_NO: The processor is not affected by the Fill Buffer Stale Data Propagator (FBSDP).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>PSDP_NO: The processor is not affected by vulnerabilities involving the Primary Stale Data Propagator (PSDP).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>FB_CLEAR: If 1, the processor supports overwrite of fill buffer values as part of MD_CLEAR operations with the VERW instruction.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>FB_CLEAR_CTRL: If 1, the processor supports the IA32_MCU_OPT_CTRL MSR and allows software to set bit 3 of that MSR (FB_CLEAR_DIS).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>RRSBA: A value of 1 indicates the processor may have the RRSBA alternate prediction behavior, if not disabled by RRSBA_DIS_U or RRSBA_DIS_S.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>BHI_NO: A value of 1 indicates BHI_NO branch prediction behavior, regardless of the value of IA32_SPEC_CTRL[BHI_DIS_S] MSR bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>XAPIC_DISABLE_STATUS: Enumerates that the IA32_XAPIC_DISABLE_STATUS MSR exists, and that bit 0 specifies whether the legacy xAPIC is disabled and APIC state is locked to x2APIC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>OVERCLOCKING_STATUS: If set, the IA32_OVERCLOCKING_STATUS MSR exists.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>PBRSB_NO: If 1, the processor is not affected by issues related to Post-Barrier Return Stack Buffer Predictions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 10BH, 267 | IA32_FLUSH_CMD |

Flush Command (WO)
Gives software a way to invalidate structures with finer granularity than other architectural methods.

0 | L1D_FLUSH: Writeback and invalidate the L1 data cache. | If CPUID.(EAX=07H, ECX=0):EDX[28]=1 |

63:1 | Reserved. | |

Register Address: 10FH, 271 | IA32_TSX_FORCE_ABORT |

TSX Force Abort | If CPUID.(EAX=07H, ECX=0):EDX[13]=1 |
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RTM_FORCE_ABORT</td>
<td></td>
<td>R/W, Default: 0</td>
</tr>
<tr>
<td>1</td>
<td>TSX_CPUID_CLEAR</td>
<td></td>
<td>R/W, Default: 0</td>
</tr>
<tr>
<td>2</td>
<td>SDV_ENABLE_RTM</td>
<td></td>
<td>R/W, Default: 0</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>RTM_DISABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TSX_CPUID_CLEAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>RNGDS_MITG_DIS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>RTM_ALLOW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>RTM_LOCKED</td>
<td>When 1, RTM_ALLOW is locked at zero, writes to RTM_ALLOW will be ignored.</td>
<td>Read-Only status bit.</td>
</tr>
<tr>
<td>3</td>
<td>FB_CLEAR_DIS</td>
<td>If 1, prevents the VERW instruction from performing an FB_CLEAR action.</td>
<td>If IA32_ARCH_CAPABILITIES.FB_CLEAR_CTRL=1</td>
</tr>
<tr>
<td>4</td>
<td>GDS_MITG_DIS</td>
<td>If 0, the Gather Data Sampling mitigation is enabled (patch load time default). If 1 on all threads for a given core, the Gather Data Sampling mitigation is disabled.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GDS_MITG_LOCK</td>
<td>If 0, not locked, and GDS_MITG_DIS is under OS control. If 1, locked and GDS_MITG_DIS is forced to 0 (writes are ignored).</td>
<td></td>
</tr>
<tr>
<td>63:6</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSENTER_CS_MSR (R/W)</td>
<td>06_01H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>CS Selector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:16</td>
<td>Not used.</td>
<td>Can be read and written.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Not used.</td>
<td>Writes ignored; reads return zero.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSENTER_ESP_MSR (R/W)</td>
<td>06_01H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSENTER_EIP_MSR (R/W)</td>
<td>06_01H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP (MCG_CAP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Machine Check Capability (R/O)</td>
<td>06_01H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Count: Number of reporting banks.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MCG_CTL_P: IA32_MCG_CTL is present if this bit is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MCG_EXT_P: Extended machine check state registers are present if this bit is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MCP_CMCI_P: Support for corrected MC error event is present.</td>
<td>06_01H</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MCG_TES_P: Threshold-based error status register are present if this bit is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>MCG_EXT_CNT: Number of extended machine check state registers present.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MCG_SER_P: The processor supports software error recovery if this bit is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format “Generic Error Data Entry” that augments the data included in machine check bank registers.</td>
<td>IA32_MCG_STATUS (MCG_STATUS)</td>
<td>06_3EH</td>
</tr>
<tr>
<td>27</td>
<td>MCG_LMCE_P: Indicates that the processor supports extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).</td>
<td></td>
<td>06_3EH</td>
</tr>
<tr>
<td>63:28</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 17AH, 378, 17BH, 379

### Global Machine Check Status (R/W)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RIPV. Restart IP valid.</td>
<td>IA32_MCG_CTL (MCG_CTL)</td>
<td>06_01H</td>
</tr>
<tr>
<td>1</td>
<td>EIPV. Error IP valid.</td>
<td></td>
<td>06_01H</td>
</tr>
<tr>
<td>2</td>
<td>MCIP. Machine check in progress.</td>
<td></td>
<td>06_01H</td>
</tr>
<tr>
<td>3</td>
<td>LMCE_S.</td>
<td>If IA32_MCG_CAP.LMCE_P[27] = 1</td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 180H−185H, 384−389

### Global Machine Check Control (R/W)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>USR: Counts while in privilege level is not ring 0.</td>
<td>IA32_PERFEVTSEL0 (PERFEVTSEL0)</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>OS: Counts while in privilege level is ring 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Edge: Enables edge detection if set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>PC: Enables pin control.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>INT: Enables interrupt on counter overflow.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.</td>
<td>IA32_PERFEVTSEL1 (PERFEVTSEL1)</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>EN: Enables the corresponding performance counter to commence counting when this bit is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>INV: Invert the CMASK.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>CMASK: When CMASK is not zero, the corresponding performance counter increments each cycle if the event count is greater than or equal to the CMASK.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 187H, 391
<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Event Select Register 1 (R/W)</td>
<td>IA32_PERFEVTSEL2</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 1</td>
</tr>
<tr>
<td>Register Address: 188H, 392</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Event Select Register 2 (R/W)</td>
<td>IA32_PERFEVTSEL3</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 2</td>
</tr>
<tr>
<td>Register Address: 189H, 393</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Event Select Register 3 (R/W)</td>
<td>IA32_PERFEVTSEL4</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 3</td>
</tr>
<tr>
<td>Register Address: 18AH, 394</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Event Select Register 4 (R/W)</td>
<td>IA32_PERFEVTSEL5</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 4</td>
</tr>
<tr>
<td>Register Address: 18BH, 395</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Event Select Register 5 (R/W)</td>
<td>IA32_PERFEVTSEL6</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 5</td>
</tr>
<tr>
<td>Register Address: 18CH, 396</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Event Select Register 6 (R/W)</td>
<td>IA32_PERFEVTSEL7</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 6</td>
</tr>
<tr>
<td>Register Address: 18DH, 397</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Event Select Register 7 (R/W)</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Register Address: 18AH−194H, 394−404</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved.</td>
<td></td>
<td>06_0EH3</td>
</tr>
<tr>
<td>Register Address: 195H, 405</td>
<td>IA32_OVERCLOCKING_STATUS</td>
<td></td>
</tr>
<tr>
<td>Overclocking Status (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_ARCH_CAPABILITIES[bit 23] enumerates support for this MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Overclocking Utilized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates if specific forms of overclocking have been enabled on this boot or reset cycle: 0 indicates no, 1 indicates yes.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Undervolt Protection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates if the &quot;Dynamic OC Undervolt Protection&quot; security feature is active: 0 indicates disabled, 1 indicates enabled.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Overclocking Secure Status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates that overclocking capabilities have been unlocked by BIOS, with or without overclocking: 0 indicates Not Secured, 1 indicates Secure.</td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 196H−197H, 406−407</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Reserved.</td>
<td></td>
<td>06_0EH3</td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td></td>
</tr>
<tr>
<td>Current Performance Status (R/O)</td>
<td>IA32_PERF_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 15.1.1, “Software Interface For Initiating Performance State Transitions.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Current Performance State Value.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Control MSR (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software makes a request for a new Performance state (P-State) by writing this MSR. See Section 15.1.1, “Software Interface For Initiating Performance State Transitions.”</td>
<td>0F_03H</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
</tr>
<tr>
<td>15:0</td>
<td>Target performance State Value.</td>
</tr>
<tr>
<td>31:16</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
| 32 | Intel® Dynamic Acceleration Technology Engage (R/W) | 06_0FH (Mobile only)  
When set to 1: Disengages Intel Dynamic Acceleration Technology. |
| 63:33 | Reserved. |  |
| Register Address: 19AH, 410 | IA32_CLOCK_MODULATION |  |
| Clock Modulation Control (R/W) | See Section 15.8.3, “Software Controlled Clock Modulation.” |  |
| 0 | Extended On-Demand Clock Modulation Duty Cycle. | If CPUID.01H:EDX[22] = 1 |
| 3:1 | On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation. | If CPUID.01H:EDX[22] = 1 |
| 4 | On-Demand Clock Modulation Enable: Set 1 to enable modulation. | If CPUID.01H:EDX[22] = 1 |
| 63:5 | Reserved. |  |
| Register Address: 19BH, 411 | IA32_THERM_INTERRUPT |  |
| Thermal Interrupt Control (R/W) | Enables and disables the generation of an interrupt on temperature transitions detected with the processor’s thermal sensors and thermal monitor.  
See Section 15.8.2, “Thermal Monitor.” | If CPUID.01H:EDX[22] = 1 |
| 0 | High-Temperature Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 1 | Low-Temperature Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 2 | PROCHOT# Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 3 | FORCEPR# Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 4 | Critical Temperature Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 7:5 | Reserved. |  |
| 14:8 | Threshold #1 Value | If CPUID.01H:EDX[22] = 1 |
| 15 | Threshold #1 Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 22:16 | Threshold #2 Value | If CPUID.01H:EDX[22] = 1 |
| 23 | Threshold #2 Interrupt Enable | If CPUID.01H:EDX[22] = 1 |
| 24 | Power Limit Notification Enable | If CPUID.06H:EAX[4] = 1 |
| 25 | Hardware Feedback Notification Enable | If CPUID.06H:EAX[24] = 1 |
| 63:26 | Reserved. |  |
| Register Address: 19CH, 412 | IA32_THERM_STATUS |  |
| Thermal Status Information (R/O) | Contains status information about the processor’s thermal sensor and automatic thermal monitoring facilities.  
See Section 15.8.2, “Thermal Monitor.” | If CPUID.01H:EDX[22] = 1 |
| 0 | Thermal Status (R/O) | If CPUID.01H:EDX[22] = 1 |
| 1 | Thermal Status Log (R/W) | If CPUID.01H:EDX[22] = 1 |
| 2 | PROCHOT # or FORCEPR# event (R/O) | If CPUID.01H:EDX[22] = 1 |
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
</tr>
<tr>
<td>3</td>
<td>PROCHOT # or FORCEPR# log (R/wC0)</td>
</tr>
<tr>
<td>4</td>
<td>Critical Temperature Status (R/O)</td>
</tr>
<tr>
<td>5</td>
<td>Critical Temperature Status log (R/wC0)</td>
</tr>
<tr>
<td>6</td>
<td>Thermal Threshold #1 Status (R/O)</td>
</tr>
<tr>
<td>7</td>
<td>Thermal Threshold #1 log (R/wC0)</td>
</tr>
<tr>
<td>8</td>
<td>Thermal Threshold #2 Status (R/O)</td>
</tr>
<tr>
<td>9</td>
<td>Thermal Threshold #2 log (R/wC0)</td>
</tr>
<tr>
<td>10</td>
<td>Power Limitation Status (R/O)</td>
</tr>
<tr>
<td>11</td>
<td>Power Limitation log (R/wC0)</td>
</tr>
<tr>
<td>12</td>
<td>Current Limit Status (R/O)</td>
</tr>
<tr>
<td>13</td>
<td>Current Limit log (R/wC0)</td>
</tr>
<tr>
<td>14</td>
<td>Cross Domain Limit Status (R/O)</td>
</tr>
<tr>
<td>15</td>
<td>Cross Domain Limit log (R/wC0)</td>
</tr>
<tr>
<td>22:16</td>
<td>Digital Readout (R/O)</td>
</tr>
<tr>
<td>26:23</td>
<td>Reserved.</td>
</tr>
<tr>
<td>30:27</td>
<td>Resolution in Degrees Celsius (R/O)</td>
</tr>
<tr>
<td>31</td>
<td>Reading Valid (R/O)</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 1A0H, 416 | IA32_MISC_ENABLE

Enable Misc. Processor Features (R/w)

Allows a variety of processor functions to be enabled and disabled.

0 | Fast-Strings Enable
   When set, the fast-strings feature (for REP MOVBS and REP STOS) is enabled (default). When clear, fast-strings are disabled.
   0F_0H

2:1 | Reserved.

3 | Automatic Thermal Control Circuit Enable (R/w)
   1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation.
   0 = Disabled.
   Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2, and adaptive thermal throttling will still be activated.
   The default value of this field varies with product. See respective tables where default value is listed.
   0F_0H

6:4 | Reserved.

7 | Performance Monitoring Available (R)
   1 = Performance monitoring enabled.
   0 = Performance monitoring disabled.
   0F_0H

10:8 | Reserved.
<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 = Processor doesn't support branch trace storage (BTS).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0 = BTS is supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Processor Event Based Sampling (PEBS) Unavailable (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 = PEBS is not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0 = PEBS is supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Enhanced Intel SpeedStep Technology Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0 = Enhanced Intel SpeedStep Technology disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1 = Enhanced Intel SpeedStep Technology enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>If CPUID.01H: ECX[7] =1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Limit CPUID Maxval (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>BIOS should contain a setup question that allows users to specify when the installed OS does not support CPUID functions greater than 2. Before setting this bit, BIOS must execute the CPUID.0H and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 2, this bit is supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Otherwise, this bit is not supported. Setting this bit when the maximum value is not greater than 2 may generate a #GP exception. Setting this bit may cause unexpected behavior in software that depends on the availability of CPUID leaves greater than 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>If CPUID.01H:ECX[14] = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Note: Some older processors defined one of these bits as a disable for the execute-disable feature of paging. If a processor supports this bit, this information is provided in the model-specific tables. See Table 2-3 for the definition of this bit.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Energy Bias Hint (R/W)</td>
<td>Power Policy Preference: 0 indicates preference to highest performance. 15 indicates preference to maximize energy saving.</td>
<td>If CPUID.6H:ECX[3] = 1</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 1B1H, 433 | IA32_PACKAGE_THERM_STATUS

Package Thermal Status Information (R/O)
Contains status information about the package's thermal sensor.
See Section 15.9, “Package Level Thermal Management.”

<table>
<thead>
<tr>
<th>Bit</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pkg Thermal Status (R/O)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Pkg Thermal Status Log (R/W)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pkg PROCHOT # event. (R/O)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Pkg PROCHOT # log. (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pkg Critical Temperature Status. (R/O)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Pkg Critical Temperature Status Log. (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Pkg Thermal Threshold #1 Status. (R/O)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Pkg Thermal Threshold #1 Log. (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Pkg Thermal Threshold #2 Status. (R/O)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Pkg Thermal Threshold #2 Log. (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Pkg Power Limitation Status. (R/O)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Pkg Power Limitation Log. (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>22:16</td>
<td>Pkg Digital Readout. (R/O)</td>
<td></td>
</tr>
<tr>
<td>25:23</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Hardware Feedback Interface Structure Change Status.</td>
<td>If CPUID.06H:EAX[19] = 1</td>
</tr>
<tr>
<td>63:27</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 1B2H, 434 | IA32_PACKAGE_THERM_INTERRUPT

Pkg Thermal Interrupt Control (R/W)
Enables and disables the generation of an interrupt on temperature transitions detected with the package’s thermal sensor.
See Section 15.9, “Package Level Thermal Management.”

<table>
<thead>
<tr>
<th>Bit</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pkg High-Temperature Interrupt Enable.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Pkg Low-Temperature Interrupt Enable.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pkg PROCHOT# Interrupt Enable.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pkg Overheat Interrupt Enable.</td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>Pkg Threshold #1 Value.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Pkg Threshold #1 Interrupt Enable.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:16</td>
<td>Pkg Threshold #2 Value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Pkg Threshold #2 Interrupt Enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Pkg Power Limit Notification Enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Hardware Feedback Interrupt Enable.</td>
<td></td>
<td>If CPUID.06H:EAX[19] = 1</td>
</tr>
<tr>
<td>63:26</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1C4H, 452</td>
<td></td>
<td>IA32_XFD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extended Feature Disable Control (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Controls which XSAVE-enabled features are temporarily disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CPUID.(EAX=0DH,ECX=1): EAX[4] = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1C5H, 453</td>
<td></td>
<td>IA32_XFD_ERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extended Feature Disable Error Code (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reports which XSAVE-enabled features caused a fault due to being disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CPUID.(EAX=0DH,ECX=1): EAX[4] = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D9H, 473</td>
<td></td>
<td>IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Trace/Profile Resource Control (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.</td>
<td>06_0EH</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.</td>
<td>06_01H</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BLD: Enable OS bus-lock detection. See Section 18.3.1.6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.</td>
<td>If (CPUID.(EAX=07H, ECX=0):ECX[24] = 1)</td>
<td></td>
</tr>
<tr>
<td>5:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TR: Setting this bit to 1 enables branch trace messages to be sent.</td>
<td>06_0EH</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.</td>
<td>06_0EH</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.</td>
<td>06_0EH</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.</td>
<td>06_0FH</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BTS_OFF_USR: When set, BTS or BTM is skipped if CPL &gt; 0.</td>
<td>06_0FH</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.</td>
<td>If CPUID.01H: ECX[15] = 1 &amp;&amp; CPUID.OAH: EAX[7:0] &gt; 1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.</td>
<td>If CPUID.01H: ECX[15] = 1 &amp;&amp; CPUID.OAH: EAX[7:0] &gt; 1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.</td>
<td>06_1AH</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.</td>
<td>If IA32_PERF_CAPABILITIES[12] = 1</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td>IA32_LER_FROM_IP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Last Event Record Source IP Register (R/W)</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>FROM_IP</td>
<td>The source IP of the recorded branch or event, in canonical form.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 1DDH, 477</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_LER_TO_IP</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>TO_IP</td>
<td>The destination IP of the recorded branch or event, in canonical form.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 1DEH, 478</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_LER_INFO</td>
<td></td>
</tr>
<tr>
<td>55:0</td>
<td>Undefined, may be zero or non-zero. Writes of non-zero values do not fault, but reads may return a different value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59:56</td>
<td>BR_TYPE</td>
<td>The branch type recorded by this LBR. Encodings match those of IA32_LBR_x_INFO.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>60</td>
<td>Undefined, may be zero or non-zero. Writes of non-zero values do not fault, but reads may return a different value.</td>
<td>Reset Value: 0</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>TSX_ABORT</td>
<td>This LBR record is a TSX abort. On processors that do not support Intel® TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>62</td>
<td>IN_TSX</td>
<td>This LBR record records a branch that retired during a TSX transaction. On processors that do not support Intel® TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>63</td>
<td>MISPRED</td>
<td>The recorded branch taken/not-taken resolution (for conditional branches) or target (for any indirect branch, including RETs) was mispredicted.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 1F2H, 498</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_SMRR_PHYSBASE</td>
<td>If IA32_MTRRCAP.SMRR[11] = 1</td>
</tr>
<tr>
<td></td>
<td>SMRR Base Address (Writable only in SMM)</td>
<td>Base address of SMM memory range.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Type. Specifies memory type of the range.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>Reserved.</td>
<td>If IA32_MTRRCAP.SMR[11] = 1</td>
<td></td>
</tr>
<tr>
<td>31:12</td>
<td>PhysBase</td>
<td>SMRR physical Base Address.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 1F3H, 499</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_SMRR_PHYSMASK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMRR Range Mask (Writable only in SMM)</td>
<td>Range Mask of SMM memory range.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Range Mask of SMM memory range.</td>
<td>If IA32_MTRRCAP.SMRR[11] = 1</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enable range mask.</td>
<td></td>
</tr>
<tr>
<td>31:12</td>
<td>PhysMask</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMRR address range mask.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 1F8H, 504  
IA32_PLATFORM_DCA_CAP

DCA Capability (R)  
If CPUID.01H: ECX[18] = 1

Register Address: 1F9H, 505  
IA32_CPU_DCA_CAP

If set, CPU supports Prefetch-Hint type.  
If CPUID.01H: ECX[18] = 1

Register Address: 1FAH, 506  
IA32_DCA_0_CAP

DCA type 0 Status and Control register.  
If CPUID.01H: ECX[18] = 1

| 0          | DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set. |         |
| 2:1        | TRANSACTION |         |
| 6:3        | DCA_TYPE    |         |
| 10:7       | DCA_QUEUE_SIZE |         |
| 12:11      | Reserved.   |         |
| 16:13      | DCA_DELAY: Writes will update the register but have no HW side-effect. |         |
| 23:17      | Reserved.   |         |
| 24         | SW_BLOCK: SW can request DCA block by setting this bit. |         |
| 25         | Reserved.   |         |
| 26         | HW_BLOCK: Set when DCA is blocked by HW (e.g., CR0.CD = 1). |         |
| 31:27      | Reserved.   |         |

Register Address: 200H, 512  
IA32_MTRR_PHYSBASE0 (MTRRphysBase0)

See Section 12.11.2.3, “Variable Range MTRRs.”  
If IA32_MTRRCAP[7:0] > 0

Register Address: 201H, 513  
IA32_MTRR_PHYSMASK0

MTRRphysMask0  
If IA32_MTRRCAP[7:0] > 0

Register Address: 202H, 514  
IA32_MTRR_PHYSBASE1

MTRRphysBase1  
If IA32_MTRRCAP[7:0] > 1

Register Address: 203H, 515  
IA32_MTRR_PHYSMASK1

MTRRphysMask1  
If IA32_MTRRCAP[7:0] > 1

Register Address: 204H, 516  
IA32_MTRR_PHYSBASE2

MTRRphysBase2  
If IA32_MTRRCAP[7:0] > 2

Register Address: 205H, 517  
IA32_MTRR_PHYSMASK2

MTRRphysMask2  
If IA32_MTRRCAP[7:0] > 2

Register Address: 206H, 518  
IA32_MTRR_PHYSBASE3

MTRRphysBase3  
If IA32_MTRRCAP[7:0] > 3
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 207H, 519</td>
<td>IA32_MTRR_PHYSMASK3</td>
<td></td>
</tr>
<tr>
<td>MTRRphysMask3</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td>If IA32_MTRRCAP[7:0] &gt; 3</td>
</tr>
<tr>
<td>Register Address: 208H, 520</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td></td>
</tr>
<tr>
<td>MTRRphysBase4</td>
<td>IA32_MTRR_PHYSMASK4</td>
<td>If IA32_MTRRCAP[7:0] &gt; 4</td>
</tr>
<tr>
<td>Register Address: 209H, 521</td>
<td>IA32_MTRR_PHYSMASK4</td>
<td></td>
</tr>
<tr>
<td>MTRRphysMask4</td>
<td>IA32_MTRR_PHYSBASE5</td>
<td>If IA32_MTRRCAP[7:0] &gt; 4</td>
</tr>
<tr>
<td>Register Address: 20AH, 522</td>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
</tr>
<tr>
<td>MTRRphysBase5</td>
<td>IA32_MTRR_PHYSMASK5</td>
<td>If IA32_MTRRCAP[7:0] &gt; 5</td>
</tr>
<tr>
<td>Register Address: 20BH, 523</td>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
</tr>
<tr>
<td>MTRRphysMask5</td>
<td>IA32_MTRR_PHYSBASE6</td>
<td>If IA32_MTRRCAP[7:0] &gt; 5</td>
</tr>
<tr>
<td>Register Address: 20CH, 524</td>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
</tr>
<tr>
<td>MTRRphysBase6</td>
<td>IA32_MTRR_PHYSMASK6</td>
<td>If IA32_MTRRCAP[7:0] &gt; 6</td>
</tr>
<tr>
<td>Register Address: 20DH, 525</td>
<td>IA32_MTRR_PHYSMASK6</td>
<td></td>
</tr>
<tr>
<td>MTRRphysMask6</td>
<td>IA32_MTRR_PHYSBASE7</td>
<td>If IA32_MTRRCAP[7:0] &gt; 6</td>
</tr>
<tr>
<td>Register Address: 20EH, 526</td>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
</tr>
<tr>
<td>MTRRphysBase7</td>
<td>IA32_MTRR_PHYSMASK7</td>
<td>If IA32_MTRRCAP[7:0] &gt; 7</td>
</tr>
<tr>
<td>Register Address: 210H, 528</td>
<td>IA32_MTRR_PHYSBASE8</td>
<td></td>
</tr>
<tr>
<td>MTRRphysBase8</td>
<td>IA32_MTRR_PHYSMASK8</td>
<td>If IA32_MTRRCAP[7:0] &gt; 7</td>
</tr>
<tr>
<td>Register Address: 211H, 529</td>
<td>IA32_MTRR_PHYSMASK8</td>
<td></td>
</tr>
<tr>
<td>MTRRphysMask8</td>
<td>IA32_MTRR_PHYSBASE9</td>
<td>If IA32_MTRRCAP[7:0] &gt; 8</td>
</tr>
<tr>
<td>Register Address: 212H, 530</td>
<td>IA32_MTRR_PHYSBASE9</td>
<td></td>
</tr>
<tr>
<td>MTRRphysBase9</td>
<td>IA32_MTRR_PHYSMASK9</td>
<td>If IA32_MTRRCAP[7:0] &gt; 8</td>
</tr>
<tr>
<td>Register Address: 213H, 531</td>
<td>IA32_MTRR_PHYSMASK9</td>
<td></td>
</tr>
<tr>
<td>MTRRphysMask9</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td>If CPUID.01H: EDX.MTRR[12] =1</td>
</tr>
<tr>
<td>Register Address: 250H, 592</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
</tr>
<tr>
<td>MTRRfix64K_00000</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td>If CPUID.01H: EDX.MTRR[12] =1</td>
</tr>
<tr>
<td>Register Address: 258H, 600</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
</tr>
<tr>
<td>MTRRfix16K_80000</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td>If CPUID.01H: EDX.MTRR[12] =1</td>
</tr>
<tr>
<td>Register Address: 259H, 601</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
</tr>
<tr>
<td>MTRRfix16K_A0000</td>
<td>IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)</td>
<td>If CPUID.01H: EDX.MTRR[12] =1</td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)</td>
<td></td>
</tr>
<tr>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td>If CPUID.01H: EDX.MTRR[12] =1</td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
</tr>
<tr>
<td>MTRRfix4K_C8000</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td>If CPUID.01H: EDX.MTRR[12] =1</td>
</tr>
<tr>
<td>Register Address: 26AH, 618</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTRRfix4K_D0000</td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td></td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRRfix4K_D8000</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td></td>
<td></td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRRfix4K_E0000</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td></td>
<td></td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRRfix4K_E8000</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td></td>
<td></td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRRfix4K_F0000</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td></td>
<td></td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRRfix4K_F8000</td>
<td>IA32_PAT</td>
<td></td>
<td></td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_PAT (R/W)</td>
<td></td>
<td>2:0</td>
<td>PA0</td>
<td>If CPUID.01H: EDX.MTRR[16] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10:8</td>
<td>PA1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>15:11</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>18:16</td>
<td>PA2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>23:19</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>26:24</td>
<td>PA3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>31:27</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>34:32</td>
<td>PA4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>39:35</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>42:40</td>
<td>PA5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>47:43</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50:48</td>
<td>PA6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>55:51</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>58:56</td>
<td>PA7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:59</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 280H, 640</td>
<td>IA32_MC0_CTL2</td>
<td></td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14:0</td>
<td>Corrected error count threshold.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>29:15</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td>CMCI_EN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:31</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 281H, 641</td>
<td>IA32_MC1_CTL2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC0_CTL2</td>
</tr>
<tr>
<td>Register Address: 282H, 642</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC1_CTL2</td>
</tr>
<tr>
<td>Register Address: 283H, 643</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC2_CTL2</td>
</tr>
<tr>
<td>Register Address: 284H, 644</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC3_CTL2</td>
</tr>
<tr>
<td>Register Address: 285H, 645</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC4_CTL2</td>
</tr>
<tr>
<td>Register Address: 286H, 646</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC5_CTL2</td>
</tr>
<tr>
<td>Register Address: 287H, 647</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC6_CTL2</td>
</tr>
<tr>
<td>Register Address: 288H, 648</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC7_CTL2</td>
</tr>
<tr>
<td>Register Address: 289H, 649</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC8_CTL2</td>
</tr>
<tr>
<td>Register Address: 28AH, 650</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC9_CTL2</td>
</tr>
<tr>
<td>Register Address: 28BH, 651</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC10_CTL2</td>
</tr>
<tr>
<td>Register Address: 28CH, 652</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC11_CTL2</td>
</tr>
<tr>
<td>Register Address: 28DH, 653</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC12_CTL2</td>
</tr>
<tr>
<td>Register Address: 28EH, 654</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>IA32_MC13_CTL2</td>
</tr>
<tr>
<td>Register Address: 28FH, 655</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 15</td>
</tr>
<tr>
<td>Register Address: 290H, 656</td>
<td>IA32_MC16_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 16</td>
</tr>
<tr>
<td>Register Address: 291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 17</td>
</tr>
<tr>
<td>Register Address: 292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 18</td>
</tr>
<tr>
<td>Register Address: 293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 19</td>
</tr>
<tr>
<td>Register Address: 294H, 660</td>
<td>IA32_MC20_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 20</td>
</tr>
<tr>
<td>Register Address: 295H, 661</td>
<td>IA32_MC21_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 21</td>
</tr>
<tr>
<td>Register Address: 296H, 662</td>
<td>IA32_MC22_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 22</td>
</tr>
<tr>
<td>Register Address: 297H, 663</td>
<td>IA32_MC23_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 23</td>
</tr>
<tr>
<td>Register Address: 298H, 664</td>
<td>IA32_MC24_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 24</td>
</tr>
<tr>
<td>Register Address: 299H, 665</td>
<td>IA32_MC25_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 25</td>
</tr>
<tr>
<td>Register Address: 29AH, 666</td>
<td>IA32_MC26_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 26</td>
</tr>
<tr>
<td>Register Address: 29BH, 667</td>
<td>IA32_MC27_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 27</td>
</tr>
<tr>
<td>Register Address: 29CH, 668</td>
<td>IA32_MC28_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td></td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 28</td>
</tr>
<tr>
<td>Register Address: 29DH, 669</td>
<td>IA32_MC29_CTL2</td>
<td></td>
</tr>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 29</td>
<td></td>
</tr>
<tr>
<td>Register Address: 29EH, 670</td>
<td>IA32_MC30_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 30</td>
<td></td>
</tr>
<tr>
<td>Register Address: 29FH, 671</td>
<td>IA32_MC31_CTL2</td>
<td></td>
</tr>
<tr>
<td>Same fields as IA32_MC0_CTL2. (R/W)</td>
<td>If IA32_MCG_CAP[10] = 1 &amp;&amp; IA32_MCG_CAP[7:0] &gt; 31</td>
<td></td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td></td>
</tr>
<tr>
<td>MTRRdefType (R/W)</td>
<td>If CPUID.01H: EDX.MTRR[12] = 1</td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Default Memory Type</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Fixed Range MTRR Enable</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MTRR Enable</td>
<td></td>
</tr>
<tr>
<td>63:12</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 0</td>
<td></td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 30BH, 779</td>
<td>IA32_FIXED_CTR2</td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 2</td>
<td></td>
</tr>
<tr>
<td>Register Address: 345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td></td>
</tr>
<tr>
<td>Read Only MSR that enumerates the existence of performance monitoring features. (R/O)</td>
<td>If CPUID.01H: ECX[15] = 1</td>
<td></td>
</tr>
<tr>
<td>5:0</td>
<td>LBR format</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PEBS Trap</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PEBSSaveArchRegs</td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>PEBS Record Format</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1: Freeze while SMM is supported.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1: Full width of counter writable via IA32_A_PMCx.</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>PEBS_BASELINE</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1: Performance metrics available.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1: PEBS output will be written into the Intel PT trace stream. If CPUID.0x7.0.EBX[25]=1</td>
<td></td>
</tr>
<tr>
<td>63:17</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 38DH, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Control (R/W)</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 1</td>
<td></td>
</tr>
<tr>
<td>Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>EN0.OS: Enable Fixed Counter 0 to count while CPL = 0.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>EN0.Usr: Enable Fixed Counter 0 to count while CPL &gt; 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>AnyThr0: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.</td>
<td></td>
<td>If CPUID.0AH:EAX[7:0] &gt; 2 &amp;&amp; CPUID.0AH:EDX[15]=0</td>
</tr>
<tr>
<td>3</td>
<td>EN0_PMI: Enable PMI when fixed counter 0 overflows.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EN1_OS: Enable Fixed Counter 1 to count while CPL = 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>EN1_Usr: Enable Fixed Counter 1 to count while CPL &gt; 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>AnyThr1: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.</td>
<td></td>
<td>If CPUID.0AH:EAX[7:0] &gt; 2 &amp;&amp; CPUID.0AH:EDX[15]=0</td>
</tr>
<tr>
<td>7</td>
<td>EN1_PMI: Enable PMI when fixed counter 1 overflows.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>EN2_Usr: Enable Fixed Counter 2 to count while CPL &gt; 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>AnyThr2: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.</td>
<td></td>
<td>If CPUID.0AH:EAX[7:0] &gt; 2 &amp;&amp; CPUID.0AH:EDX[15]=0</td>
</tr>
<tr>
<td>11</td>
<td>EN2_PMI: Enable PMI when fixed counter 2 overflows.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>EN3_OS: Enable Fixed Counter 3 to count while CPL = 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>EN3_Usr: Enable Fixed Counter 3 to count while CPL &gt; 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>EN3_PMI: Enable PMI when fixed counter 3 overflows.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 38EH, 910

**Global Performance Counter Status (R/O)**

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ovf_PMC0: Overflow status of IA32_PMC0.</td>
<td></td>
<td>If CPUID.0AH: EAX[15:8] &gt; 0</td>
</tr>
<tr>
<td>1</td>
<td>Ovf_PMC1: Overflow status of IA32_PMC1.</td>
<td></td>
<td>If CPUID.0AH: EAX[15:8] &gt; 1</td>
</tr>
<tr>
<td>2</td>
<td>Ovf_PMC2: Overflow status of IA32_PMC2.</td>
<td></td>
<td>If CPUID.0AH: EAX[15:8] &gt; 2</td>
</tr>
<tr>
<td>3</td>
<td>Ovf_PMC3: Overflow status of IA32_PMC3.</td>
<td></td>
<td>If CPUID.0AH: EAX[15:8] &gt; 3</td>
</tr>
<tr>
<td>n</td>
<td>Ovf_PMCn: Overflow status of IA32_PMCn.</td>
<td></td>
<td>If CPUID.0AH: EAX[15:8] &gt; n</td>
</tr>
<tr>
<td>31:n+1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.</td>
<td></td>
<td>If CPUID.0AH: EAX[7:0] &gt; 1</td>
</tr>
<tr>
<td>33</td>
<td>Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.</td>
<td></td>
<td>If CPUID.0AH: EAX[7:0] &gt; 1</td>
</tr>
<tr>
<td>34</td>
<td>Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.</td>
<td></td>
<td>If CPUID.0AH: EAX[7:0] &gt; 1</td>
</tr>
<tr>
<td>47:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>OVF_PERF_METRICS: If this bit is set, it indicates that PERF_METRIC counter has overflowed and a PMI is triggered; however, an overflow of fixed counter 3 should normally happen first. If this bit is clear no overflow occurred.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54:49</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer that was completely filled.</td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0):EBX[25] = 1 &amp;&amp; CPUID.(EAX=014H, ECX=0):ECX[0] = 1</td>
</tr>
<tr>
<td>57:56</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 58                             | LBR_Frz: LBRs are frozen due to:  
  • IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1.  
  • The LBR stack overflowed. |                     | If CPUID.OAH: EAX[7:0] > 3 |
| 59                             | CTR_Frz: Performance counters in the core PMU are frozen due to:  
  • IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1.  
  • One or more core PMU counters overflowed. |                     | If CPUID.OAH: EAX[7:0] > 3 |
| 60                             | ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation Intel SGX to protect an enclave. |                     | If CPUID.(EAX=07H, ECX=0):EBX[2] = 1 |
| 61                             | Ovf_Uncore: Uncore counter overflow status. |                     | If CPUID.OAH: EAX[7:0] > 2 |
| 62                             | OvfBuf: DS SAVE area Buffer overflow status. |                     | If CPUID.OAH: EAX[7:0] > 0 |
| 63                             | CondChgd: Status bits of this register have changed. |                     | If CPUID.OAH: EAX[7:0] > 0 |

<table>
<thead>
<tr>
<th>Register Address: 38FH, 911</th>
<th>IA32_PERF_GLOBAL_CTRL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EN_PMC0</td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 0</td>
</tr>
<tr>
<td>1</td>
<td>EN_PMC1</td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 1</td>
</tr>
<tr>
<td>2</td>
<td>EN_PMC2</td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 2</td>
</tr>
<tr>
<td>n</td>
<td>EN_PMCn</td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; n</td>
</tr>
<tr>
<td>31:n+1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>EN_FIXED_CTRL0</td>
<td></td>
<td>If CPUID.OAH: EDX[4:0] &gt; 0</td>
</tr>
<tr>
<td>33</td>
<td>EN_FIXED_CTRL1</td>
<td></td>
<td>If CPUID.OAH: EDX[4:0] &gt; 1</td>
</tr>
<tr>
<td>34</td>
<td>EN_FIXED_CTRL2</td>
<td></td>
<td>If CPUID.OAH: EDX[4:0] &gt; 2</td>
</tr>
<tr>
<td>47:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>EN_PERF_METRICS: If this bit is set and fixed counter 3 is effectively enabled, built-in performance metrics are enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:49</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 390H, 912</th>
<th>IA32_PERF_GLOBAL_OVF_CTRL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set 1 to Clear Ovf_PMC0 bit.</td>
<td></td>
<td>If CPUID.OAH: EAX[15:8] &gt; 0</td>
</tr>
</tbody>
</table>

Global Performance Counter Control (R/W)  
Counter increments while the result of ANDing the respective enable bit in this MSR with the corresponding OS or USR bits in the general-purpose or fixed counter control MSR is true.  
If CPUID.OAH: EAX[7:0] > 0
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Set 1 to Clear Ovf_PMC1 bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 1</td>
</tr>
<tr>
<td>2</td>
<td>Set 1 to Clear Ovf_PMC2 bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 2</td>
</tr>
<tr>
<td>n</td>
<td>Set 1 to Clear Ovf_PMCn bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; n</td>
</tr>
<tr>
<td>31:n</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to Clear Ovf_FIXED_CTR0 bit.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 0</td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to Clear Ovf_FIXED_CTR1 bit.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 1</td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to Clear Ovf_FIXED_CTR2 bit.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 2</td>
</tr>
<tr>
<td>54:35</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Set 1 to Clear Trace_ToPA_PMI bit.</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) &amp;&amp; IA32_RTIT_CTL.ToPA = 1</td>
</tr>
<tr>
<td>60:56</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Set 1 to Clear Ovf_Uncore bit.</td>
<td>06_2EH</td>
</tr>
<tr>
<td>62</td>
<td>Set 1 to Clear OvfBuf bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 0</td>
</tr>
<tr>
<td>63</td>
<td>Set 1 to clear CondChgd bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 0</td>
</tr>
</tbody>
</table>

Register Address: 390H, 912, IA32_PERF_GLOBAL_STATUS_RESET

Global Performance Counter Overflow Reset Control (R/W)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set 1 to Clear Ovf_PMC0 bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 0</td>
</tr>
<tr>
<td>1</td>
<td>Set 1 to Clear Ovf_PMC1 bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 1</td>
</tr>
<tr>
<td>2</td>
<td>Set 1 to Clear Ovf_PMC2 bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 2</td>
</tr>
<tr>
<td>n</td>
<td>Set 1 to Clear Ovf_PMCn bit.</td>
<td>If CPUID.0AH: EAX[15:8] &gt; n</td>
</tr>
<tr>
<td>31:n</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to Clear Ovf_FIXED_CTR0 bit.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 0</td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to Clear Ovf_FIXED_CTR1 bit.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 1</td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to Clear Ovf_FIXED_CTR2 bit.</td>
<td>If CPUID.0AH: EDX[4:0] &gt; 2</td>
</tr>
<tr>
<td>47:35</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Reset_OVF_PERF_METRICS: If this bit is set, it will clear the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.</td>
<td></td>
</tr>
<tr>
<td>54:49</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Set 1 to Clear Trace_ToPA_PMI bit.</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) &amp;&amp; CPUID.(EAX=014H, ECX=0):ECX[0] = 1</td>
</tr>
<tr>
<td>57:56</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Set 1 to Clear LBR_Frz bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 3</td>
</tr>
<tr>
<td>59</td>
<td>Set 1 to Clear CTR_Frz bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 3</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>Set 1 to Clear ASCI bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 3</td>
</tr>
<tr>
<td>61</td>
<td>Set 1 to Clear Ovf_Uncore bit.</td>
<td>06_2EH</td>
</tr>
<tr>
<td>62</td>
<td>Set 1 to Clear OvfBuf bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 0</td>
</tr>
<tr>
<td>63</td>
<td>Set 1 to clear CondChgd bit.</td>
<td>If CPUID.0AH: EAX[7:0] &gt; 0</td>
</tr>
</tbody>
</table>

Register Address: 391H, 913  | IA32_PERF_GLOBAL_STATUS_SET

<table>
<thead>
<tr>
<th>Global Performance Counter Overflow Set Control (R/W)</th>
<th>If CPUID.0AH: EAX[7:0] &gt; 3 II (CPUID.(EAX=07H, ECX=0):EBX[25] = 1 &amp;&amp; CPUID.(EAX=014H, ECX=0):ECX[0] = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Set 1 to cause Ovf_PMC0 = 1.</td>
</tr>
<tr>
<td>1</td>
<td>Set 1 to cause Ovf_PMC1 = 1.</td>
</tr>
<tr>
<td>2</td>
<td>Set 1 to cause Ovf_PMC2 = 1.</td>
</tr>
<tr>
<td>n</td>
<td>Set 1 to cause Ovf_PMCn = 1.</td>
</tr>
<tr>
<td>31:n</td>
<td>Reserved.</td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to cause Ovf_FIXED_CTR0 = 1.</td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to cause Ovf_FIXED_CTR1 = 1.</td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to cause Ovf_FIXED_CTR2 = 1.</td>
</tr>
<tr>
<td>47:35</td>
<td>Reserved.</td>
</tr>
<tr>
<td>48</td>
<td>SET_OVF_PERF_METRICS: If this bit is set, it will set the status bit in the IA32_PERF_GLOBAL_STATUS register for the PERF_METRICS counters.</td>
</tr>
<tr>
<td>54:49</td>
<td>Reserved.</td>
</tr>
<tr>
<td>55</td>
<td>Set 1 to cause Trace_ToPA_PMI = 1.</td>
</tr>
<tr>
<td>57:56</td>
<td>Reserved.</td>
</tr>
<tr>
<td>58</td>
<td>Set 1 to cause LBR_Frz = 1.</td>
</tr>
<tr>
<td>59</td>
<td>Set 1 to cause CTR_Frz = 1.</td>
</tr>
<tr>
<td>58</td>
<td>Set 1 to cause ASCI = 1.</td>
</tr>
<tr>
<td>61</td>
<td>Set 1 to cause Ovf_Uncore = 1.</td>
</tr>
<tr>
<td>62</td>
<td>Set 1 to cause OvfBuf = 1.</td>
</tr>
<tr>
<td>63</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 392H, 914  | IA32_PERF_GLOBAL_INUSE

<table>
<thead>
<tr>
<th>Indicator that core perfmon interface is in use. (R/O)</th>
<th>If CPUID.0AH: EAX[7:0] &gt; 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IA32_PERFEVTSEL0 in use.</td>
</tr>
<tr>
<td>1</td>
<td>IA32_PERFEVTSEL1 in use.</td>
</tr>
<tr>
<td>2</td>
<td>IA32_PERFEVTSEL2 in use.</td>
</tr>
<tr>
<td>n</td>
<td>IA32_PERFEVTSELn in use.</td>
</tr>
<tr>
<td>31:n+1</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
## Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
</tr>
<tr>
<td>32</td>
<td>IA32_FIXED_CTR0 in use.</td>
</tr>
<tr>
<td>33</td>
<td>IA32_FIXED_CTR1 in use.</td>
</tr>
<tr>
<td>34</td>
<td>IA32_FIXED_CTR2 in use.</td>
</tr>
<tr>
<td>62:35</td>
<td>Reserved or model specific.</td>
</tr>
<tr>
<td>63</td>
<td>PMI in use.</td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBS_ENABLE</td>
</tr>
<tr>
<td>PEBS Control (R/w)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Enable PEBS on IA32_PMC0.</td>
</tr>
<tr>
<td>3:1</td>
<td>Reserved or model specific.</td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>35:32</td>
<td>Reserved or model specific.</td>
</tr>
<tr>
<td>63:36</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
</tr>
<tr>
<td>MC0_CTL</td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
</tr>
<tr>
<td>MC0_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;0</td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
</tr>
<tr>
<td>MC0_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;0</td>
</tr>
<tr>
<td>Register Address: 403H, 1027</td>
<td>IA32_MC0_MISC</td>
</tr>
<tr>
<td>MC0_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;0</td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MC1_CTL</td>
</tr>
<tr>
<td>MC1_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;1</td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MC1_STATUS</td>
</tr>
<tr>
<td>MC1_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;1</td>
</tr>
<tr>
<td>Register Address: 406H, 1030</td>
<td>IA32_MC1_ADDR</td>
</tr>
<tr>
<td>MC1_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;1</td>
</tr>
<tr>
<td>Register Address: 407H, 1031</td>
<td>IA32_MC1_MISC</td>
</tr>
<tr>
<td>MC1_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;1</td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
</tr>
<tr>
<td>MC2_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;2</td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_STATUS</td>
</tr>
<tr>
<td>MC2_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;2</td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
</tr>
<tr>
<td>MC2_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;2</td>
</tr>
<tr>
<td>Register Address: 40BH, 1035</td>
<td>IA32_MC2_MISC</td>
</tr>
<tr>
<td>MC2_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;2</td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
<td></td>
</tr>
<tr>
<td>MC3_CTL</td>
<td>Register Address: 40DH, 1037</td>
<td>IA32_MC3_STATUS</td>
</tr>
<tr>
<td>MC3_STATUS</td>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MC3_ADDR</td>
</tr>
<tr>
<td>MC3_ADDR</td>
<td>Register Address: 40FH, 1039</td>
<td>IA32_MC3_MISC</td>
</tr>
<tr>
<td>MC3_MISC</td>
<td>Register Address: 410H, 1040</td>
<td>IA32_MC4_CTL</td>
</tr>
<tr>
<td>MC4_CTL</td>
<td>Register Address: 411H, 1041</td>
<td>IA32_MC4_STATUS</td>
</tr>
<tr>
<td>MC4_STATUS</td>
<td>Register Address: 412H, 1042</td>
<td>IA32_MC4_ADDR</td>
</tr>
<tr>
<td>MC4_ADDR</td>
<td>Register Address: 413H, 1043</td>
<td>IA32_MC4_MISC</td>
</tr>
<tr>
<td>MC4_MISC</td>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
</tr>
<tr>
<td>MC5_CTL</td>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
</tr>
<tr>
<td>MC5_STATUS</td>
<td>Register Address: 416H, 1046</td>
<td>IA32_MC5_ADDR</td>
</tr>
<tr>
<td>MC5_ADDR</td>
<td>Register Address: 417H, 1047</td>
<td>IA32_MC5_MISC</td>
</tr>
<tr>
<td>MC5_MISC</td>
<td>Register Address: 418H, 1048</td>
<td>IA32_MC6_CTL</td>
</tr>
<tr>
<td>MC6_CTL</td>
<td>Register Address: 419H, 1049</td>
<td>IA32_MC6_STATUS</td>
</tr>
<tr>
<td>MC6_STATUS</td>
<td>Register Address: 41AH, 1050</td>
<td>IA32_MC6_ADDR</td>
</tr>
<tr>
<td>MC6_ADDR</td>
<td>Register Address: 41BH, 1051</td>
<td>IA32_MC6_MISC</td>
</tr>
<tr>
<td>MC6_MISC</td>
<td>Register Address: 41CH, 1052</td>
<td>IA32_MC7_CTL</td>
</tr>
<tr>
<td>MC7_CTL</td>
<td>Register Address: 41DH, 1053</td>
<td>IA32_MC7_STATUS</td>
</tr>
<tr>
<td>MC7_STATUS</td>
<td>Register Address: 41EH, 1054</td>
<td>IA32_MC7_ADDR</td>
</tr>
<tr>
<td>MC7_ADDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 41FH, 1055</td>
<td>IA32_MC7_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;7</td>
<td></td>
</tr>
<tr>
<td>Register Address: 420H, 1056</td>
<td>IA32_MC8_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;8</td>
<td></td>
</tr>
<tr>
<td>Register Address: 421H, 1057</td>
<td>IA32_MC8_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;8</td>
<td></td>
</tr>
<tr>
<td>Register Address: 422H, 1058</td>
<td>IA32_MC8_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;8</td>
<td></td>
</tr>
<tr>
<td>Register Address: 423H, 1059</td>
<td>IA32_MC8_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;8</td>
<td></td>
</tr>
<tr>
<td>Register Address: 424H, 1060</td>
<td>IA32_MC9_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;9</td>
<td></td>
</tr>
<tr>
<td>Register Address: 425H, 1061</td>
<td>IA32_MC9_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;9</td>
<td></td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td>IA32_MC9_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;9</td>
<td></td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td>IA32_MC9_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;9</td>
<td></td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td>IA32_MC10_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;10</td>
<td></td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td>IA32_MC10_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;10</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td>IA32_MC10_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;10</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42BH, 1067</td>
<td>IA32_MC10_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;10</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42CH, 1068</td>
<td>IA32_MC11_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;11</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42DH, 1069</td>
<td>IA32_MC11_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;11</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42EH, 1070</td>
<td>IA32_MC11_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;11</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42FH, 1071</td>
<td>IA32_MC11_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;11</td>
<td></td>
</tr>
<tr>
<td>Register Address: 430H, 1072</td>
<td>IA32_MC12_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;12</td>
<td></td>
</tr>
<tr>
<td>Register Address: 431H, 1073</td>
<td>IA32_MC12_STATUS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-34 Vol. 4
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC12_STATUS</td>
<td>IA32_MCG_CAP.CNT &gt;12</td>
<td></td>
</tr>
<tr>
<td>Register Address: 432H, 1074</td>
<td>IA32_MC12_ADDR</td>
<td></td>
</tr>
<tr>
<td>MC12_ADDR</td>
<td>IA32_MCG_CAP.CNT &gt;12</td>
<td></td>
</tr>
<tr>
<td>Register Address: 433H, 1075</td>
<td>IA32_MC12_MISC</td>
<td></td>
</tr>
<tr>
<td>MC12_MISC</td>
<td>IA32_MCG_CAP.CNT &gt;12</td>
<td></td>
</tr>
<tr>
<td>Register Address: 434H, 1076</td>
<td>IA32_MC13_CTL</td>
<td></td>
</tr>
<tr>
<td>MC13_CTL</td>
<td>IA32_MCG_CAP.CNT &gt;13</td>
<td></td>
</tr>
<tr>
<td>Register Address: 435H, 1077</td>
<td>IA32_MC13_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC13_STATUS</td>
<td>IA32_MCG_CAP.CNT &gt;13</td>
<td></td>
</tr>
<tr>
<td>Register Address: 436H, 1078</td>
<td>IA32_MC13_ADDR</td>
<td></td>
</tr>
<tr>
<td>MC13_ADDR</td>
<td>IA32_MCG_CAP.CNT &gt;13</td>
<td></td>
</tr>
<tr>
<td>Register Address: 437H, 1079</td>
<td>IA32_MC13_MISC</td>
<td></td>
</tr>
<tr>
<td>MC13_MISC</td>
<td>IA32_MCG_CAP.CNT &gt;13</td>
<td></td>
</tr>
<tr>
<td>Register Address: 438H, 1080</td>
<td>IA32_MC14_CTL</td>
<td></td>
</tr>
<tr>
<td>MC14_CTL</td>
<td>IA32_MCG_CAP.CNT &gt;14</td>
<td></td>
</tr>
<tr>
<td>Register Address: 439H, 1081</td>
<td>IA32_MC14_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC14_STATUS</td>
<td>IA32_MCG_CAP.CNT &gt;14</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43AH, 1082</td>
<td>IA32_MC14_ADDR</td>
<td></td>
</tr>
<tr>
<td>MC14_ADDR</td>
<td>IA32_MCG_CAP.CNT &gt;14</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43BH, 1083</td>
<td>IA32_MC14_MISC</td>
<td></td>
</tr>
<tr>
<td>MC14_MISC</td>
<td>IA32_MCG_CAP.CNT &gt;14</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43CH, 1084</td>
<td>IA32_MC15_CTL</td>
<td></td>
</tr>
<tr>
<td>MC15_CTL</td>
<td>IA32_MCG_CAP.CNT &gt;15</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43DH, 1085</td>
<td>IA32_MC15_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC15_STATUS</td>
<td>IA32_MCG_CAP.CNT &gt;15</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43EH, 1086</td>
<td>IA32_MC15_ADDR</td>
<td></td>
</tr>
<tr>
<td>MC15_ADDR</td>
<td>IA32_MCG_CAP.CNT &gt;15</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43FH, 1087</td>
<td>IA32_MC15_MISC</td>
<td></td>
</tr>
<tr>
<td>MC15_MISC</td>
<td>IA32_MCG_CAP.CNT &gt;15</td>
<td></td>
</tr>
<tr>
<td>Register Address: 440H, 1088</td>
<td>IA32_MC16_CTL</td>
<td></td>
</tr>
<tr>
<td>MC16_CTL</td>
<td>IA32_MCG_CAP.CNT &gt;16</td>
<td></td>
</tr>
<tr>
<td>Register Address: 441H, 1089</td>
<td>IA32_MC16_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC16_STATUS</td>
<td>IA32_MCG_CAP.CNT &gt;16</td>
<td></td>
</tr>
<tr>
<td>Register Address: 442H, 1090</td>
<td>IA32_MC16_ADDR</td>
<td></td>
</tr>
<tr>
<td>MC16_ADDR</td>
<td>IA32_MCG_CAP.CNT &gt;16</td>
<td></td>
</tr>
<tr>
<td>Register Address: 443H, 1091</td>
<td>IA32_MC16_MISC</td>
<td></td>
</tr>
<tr>
<td>MC16_MISC</td>
<td>IA32_MCG_CAP.CNT &gt;16</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 444H, 1092</td>
<td>IA32_MC17_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;17</td>
</tr>
<tr>
<td>MC17_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 445H, 1093</td>
<td>IA32_MC17_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC17_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 446H, 1094</td>
<td>IA32_MC17_ADDR¹</td>
<td></td>
</tr>
<tr>
<td>MC17_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 447H, 1095</td>
<td>IA32_MC17_MISC</td>
<td></td>
</tr>
<tr>
<td>MC17_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 448H, 1096</td>
<td>IA32_MC18_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;18</td>
</tr>
<tr>
<td>MC18_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 449H, 1097</td>
<td>IA32_MC18_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC18_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44AH, 1098</td>
<td>IA32_MC18_ADDR¹</td>
<td></td>
</tr>
<tr>
<td>MC18_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44BH, 1099</td>
<td>IA32_MC18_MISC</td>
<td></td>
</tr>
<tr>
<td>MC18_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;19</td>
</tr>
<tr>
<td>MC19_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44DH, 1101</td>
<td>IA32_MC19_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC19_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44EH, 1102</td>
<td>IA32_MC19_ADDR¹</td>
<td></td>
</tr>
<tr>
<td>MC19_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44FH, 1103</td>
<td>IA32_MC19_MISC</td>
<td></td>
</tr>
<tr>
<td>MC19_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 450H, 1104</td>
<td>IA32_MC20_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;20</td>
</tr>
<tr>
<td>MC20_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 451H, 1105</td>
<td>IA32_MC20_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC20_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 452H, 11061106</td>
<td>IA32_MC20_ADDR¹</td>
<td></td>
</tr>
<tr>
<td>MC20_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 453H, 1107</td>
<td>IA32_MC20_MISC</td>
<td></td>
</tr>
<tr>
<td>MC20_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 454H, 1108</td>
<td>IA32_MC21_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;21</td>
</tr>
<tr>
<td>MC21_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 455H, 1109</td>
<td>IA32_MC21_STATUS</td>
<td></td>
</tr>
<tr>
<td>MC21_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 456H, 1110</td>
<td>IA32_MC21_ADDR¹</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC21_ADDR</td>
<td></td>
<td>IA32_MC21_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;21</td>
</tr>
<tr>
<td>MC21_MISC</td>
<td></td>
<td>IA32_MC22_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;21</td>
</tr>
<tr>
<td>MC22_CTL</td>
<td></td>
<td>IA32_MC22_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;22</td>
</tr>
<tr>
<td>MC22_STATUS</td>
<td></td>
<td>IA32_MC22_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;22</td>
</tr>
<tr>
<td>MC22_ADDR</td>
<td></td>
<td>IA32_MC22_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;22</td>
</tr>
<tr>
<td>MC22_MISC</td>
<td></td>
<td>IA32_MC23_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;22</td>
</tr>
<tr>
<td>MC23_CTL</td>
<td></td>
<td>IA32_MC23_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;23</td>
</tr>
<tr>
<td>MC23_STATUS</td>
<td></td>
<td>IA32_MC23_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;23</td>
</tr>
<tr>
<td>MC23_ADDR</td>
<td></td>
<td>IA32_MC23_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;23</td>
</tr>
<tr>
<td>MC23_MISC</td>
<td></td>
<td>IA32_MC24_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;23</td>
</tr>
<tr>
<td>MC24_CTL</td>
<td></td>
<td>IA32_MC24_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;24</td>
</tr>
<tr>
<td>MC24_STATUS</td>
<td></td>
<td>IA32_MC24_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;24</td>
</tr>
<tr>
<td>MC24_ADDR</td>
<td></td>
<td>IA32_MC24_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;24</td>
</tr>
<tr>
<td>MC24_MISC</td>
<td></td>
<td>IA32_MC25_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;24</td>
</tr>
<tr>
<td>MC25_CTL</td>
<td></td>
<td>IA32_MC25_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;25</td>
</tr>
<tr>
<td>MC25_STATUS</td>
<td></td>
<td>IA32_MC25_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;25</td>
</tr>
<tr>
<td>MC25_ADDR</td>
<td></td>
<td>IA32_MC25_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;25</td>
</tr>
<tr>
<td>MC25_MISC</td>
<td></td>
<td>IA32_MC26_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;25</td>
</tr>
<tr>
<td>MC26_CTL</td>
<td></td>
<td>IA32_MC26_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;26</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 469H, 1129</td>
<td>IA32_MC26_STATUS</td>
<td>MC26_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;26</td>
</tr>
<tr>
<td>Register Address: 46AH, 1130</td>
<td>IA32_MC26_ADDR¹</td>
<td>MC26_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;26</td>
</tr>
<tr>
<td>Register Address: 46BH, 1131</td>
<td>IA32_MC26_MISC</td>
<td>MC26_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;26</td>
</tr>
<tr>
<td>Register Address: 46CH, 1132</td>
<td>IA32_MC27_CTL</td>
<td>MC27_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;27</td>
</tr>
<tr>
<td>Register Address: 46DH, 1133</td>
<td>IA32_MC27_STATUS</td>
<td>MC27_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;27</td>
</tr>
<tr>
<td>Register Address: 46EH, 1134</td>
<td>IA32_MC27_ADDR¹</td>
<td>MC27_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;27</td>
</tr>
<tr>
<td>Register Address: 46FH, 1135</td>
<td>IA32_MC27_MISC</td>
<td>MC27_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;27</td>
</tr>
<tr>
<td>Register Address: 470H, 1136</td>
<td>IA32_MC28_CTL</td>
<td>MC28_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;28</td>
</tr>
<tr>
<td>Register Address: 471H, 1137</td>
<td>IA32_MC28_STATUS</td>
<td>MC28_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;28</td>
</tr>
<tr>
<td>Register Address: 472H, 1138</td>
<td>IA32_MC28_ADDR¹</td>
<td>MC28_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;28</td>
</tr>
<tr>
<td>Register Address: 473H, 1139</td>
<td>IA32_MC28_MISC</td>
<td>MC28_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;28</td>
</tr>
<tr>
<td>Register Address: 474H, 1140</td>
<td>IA32_MC29_CTL</td>
<td>MC29_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;29</td>
</tr>
<tr>
<td>Register Address: 475H, 1141</td>
<td>IA32_MC29_STATUS</td>
<td>MC29_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;29</td>
</tr>
<tr>
<td>Register Address: 476H, 1142</td>
<td>IA32_MC29_ADDR</td>
<td>MC29_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;29</td>
</tr>
<tr>
<td>Register Address: 477H, 1143</td>
<td>IA32_MC29_MISC</td>
<td>MC29_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;29</td>
</tr>
<tr>
<td>Register Address: 478H, 1144</td>
<td>IA32_MC30_CTL</td>
<td>MC30_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;30</td>
</tr>
<tr>
<td>Register Address: 479H, 1145</td>
<td>IA32_MC30_STATUS</td>
<td>MC30_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;30</td>
</tr>
<tr>
<td>Register Address: 47AH, 1146</td>
<td>IA32_MC30_ADDR</td>
<td>MC30_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;30</td>
</tr>
<tr>
<td>Register Address: 47BH, 1147</td>
<td>IA32_MC30_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Fields</td>
<td>MSR/Bit Description</td>
<td>Architectural MSR Name (Former MSR Name)</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>-----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>MC30_MISC</td>
<td></td>
<td>IA32_MC31_CTL</td>
<td>If IA32_MCG_CAP.CNT &gt;30</td>
</tr>
<tr>
<td>Register Address: 47CH, 1148</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC31_CTL</td>
<td></td>
<td>IA32_MC31_STATUS</td>
<td>If IA32_MCG_CAP.CNT &gt;31</td>
</tr>
<tr>
<td>Register Address: 47DH, 1149</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC31_STATUS</td>
<td></td>
<td>IA32_MC31_ADDR</td>
<td>If IA32_MCG_CAP.CNT &gt;31</td>
</tr>
<tr>
<td>Register Address: 47EH, 1150</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC31_ADDR</td>
<td></td>
<td>IA32_MC31_MISC</td>
<td>If IA32_MCG_CAP.CNT &gt;31</td>
</tr>
<tr>
<td>Register Address: 47FH, 1151</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC31_MISC</td>
<td></td>
<td>IA32_VMX_BASIC</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>Register Address: 480H, 1152</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting Register of Basic VMXCapabilities (R/O)</td>
<td></td>
<td>IA32_VMX_PINBASED_CTLS</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.1, “Basic VMX Information.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 481H, 1153</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Pin-Based VM-Execution Controls (R/O)</td>
<td></td>
<td>IA32_VMX_PROCBASED_CTLS</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.3.1, “Pin-Based VM-Execution Controls.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 482H, 1154</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)</td>
<td></td>
<td>IA32_VMX_EXIT_CTLS</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.3.2, “Primary Processor-Based VM-Execution Controls.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 483H, 1155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Primary VM-Exit Controls (R/O)</td>
<td></td>
<td>IA32_VMX_ENTRY_CTLS</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.4.1, “Primary VM-Exit Controls.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 484H, 1156</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Entry Controls (R/O)</td>
<td></td>
<td>IA32_VMX_MISC</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.5, “VM-Entry Controls.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 485H, 1157</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
<td></td>
<td>IA32_VMX_CR0_FIXED0</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.6, “Miscellaneous Data.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 486H, 1158</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
<td></td>
<td>IA32_VMX_CR0_FIXED1</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.7, “VMX-Fixed Bits in CR0.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 487H, 1159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
<td></td>
<td>IA32_VMX_CR4_FIXED0</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.7, “VMX-Fixed Bits in CR0.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 488H, 1160</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
<td></td>
<td>IA32_VMX_CR4_FIXED1</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td>See Appendix A.8, “VMX-Fixed Bits in CR4.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 489H, 1161</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>48AH, 1162</td>
<td>IA32_VMX_VMCS_ENUM</td>
<td></td>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O)</td>
<td>If CPUID.01H:ECX.[5] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.9, &quot;VMCS Enumeration.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.3.3, &quot;Secondary Processor-Based VM-Execution Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.10, &quot;VPID and EPT Capabilities.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.3.1, &quot;Pin-Based VM-Execution Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.3.2, &quot;Primary Processor-Based VM-Execution Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.4, &quot;VM-Exit Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.5, &quot;VM-Entry Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.3.4, &quot;Tertiary Processor-Based VM-Execution Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.3.4, &quot;Tertiary Processor-Based VM-Execution Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Appendix A.4.2, &quot;Secondary VM-Exit Controls.&quot;</td>
<td></td>
</tr>
<tr>
<td>4C1H, 1217</td>
<td>IA32_A_PMC0</td>
<td></td>
<td>Full Width Writable IA32_PMC0 Alias (R/W)</td>
<td>If CPUID.0AH: EAX[15:8] &gt; 0 &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
</tr>
</tbody>
</table>

2-40 Vol. 4
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 4C2H, 1218</td>
<td>IA32_A_PMC1</td>
<td></td>
</tr>
<tr>
<td>Full Width Writable IA32_PMC1 Alias (R/W)</td>
<td>(If CPUID.0AH: EAX[15:8] &gt; 1) &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C3H, 1219</td>
<td>IA32_A_PMC2</td>
<td></td>
</tr>
<tr>
<td>Full Width Writable IA32_PMC2 Alias (R/W)</td>
<td>(If CPUID.0AH: EAX[15:8] &gt; 2) &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C4H, 1220</td>
<td>IA32_A_PMC3</td>
<td></td>
</tr>
<tr>
<td>Full Width Writable IA32_PMC3 Alias (R/W)</td>
<td>(If CPUID.0AH: EAX[15:8] &gt; 3) &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C5H, 1221</td>
<td>IA32_A_PMC4</td>
<td></td>
</tr>
<tr>
<td>Full Width Writable IA32_PMC4 Alias (R/W)</td>
<td>(If CPUID.0AH: EAX[15:8] &gt; 4) &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C6H, 1222</td>
<td>IA32_A_PMC5</td>
<td></td>
</tr>
<tr>
<td>Full Width Writable IA32_PMC5 Alias (R/W)</td>
<td>(If CPUID.0AH: EAX[15:8] &gt; 5) &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C7H, 1223</td>
<td>IA32_A_PMC6</td>
<td></td>
</tr>
<tr>
<td>Full Width Writable IA32_PMC6 Alias (R/W)</td>
<td>(If CPUID.0AH: EAX[15:8] &gt; 6) &amp;&amp; IA32_PERF_CAPABILITIES[13] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C8H, 1224</td>
<td>IA32_A_PMC7</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4D0H, 1232</td>
<td>IA32_MCG_EXT_CTL</td>
<td></td>
</tr>
<tr>
<td>Allows software to signal some MCEs to only a single logical processor in the system. (R/W)</td>
<td>If IA32_MCG_CAP.LMCE_P =1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LMCE_EN</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 500H, 1280</td>
<td>IA32_SGX_SVN_STATUS</td>
<td></td>
</tr>
<tr>
<td>Status and SVN Threshold of SGX Support for ACM (R/O)</td>
<td>If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Lock.</td>
<td>See Section 39.11.3, “Interactions with Authenticated Code Modules (ACMs).”</td>
</tr>
<tr>
<td>15:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>SGX_SVN_SINIT</td>
<td>See Section 39.11.3, “Interactions with Authenticated Code Modules (ACMs).”</td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>6:0 Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:MAXPHYADDR</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 561H, 1377</td>
<td>IA32_RTIT_OUTPUT_MASK_PTRS</td>
<td></td>
</tr>
<tr>
<td>6:0 Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:7 MaskOrTableOffset.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32 Output Offset.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 570H, 1392</td>
<td>IA32_RTIT_CTL</td>
<td></td>
</tr>
<tr>
<td>Trace Control Register (R/W)</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)</td>
</tr>
<tr>
<td>0 TraceEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 CYCEn</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)</td>
</tr>
<tr>
<td>2 OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 User</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 PwrEvtEn</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=1):EBX[5] = 1)</td>
</tr>
<tr>
<td>5 FUPonPTW</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=1):EBX[4] = 1)</td>
</tr>
<tr>
<td>6 FabricEn</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=0):ECX[3] = 1)</td>
</tr>
<tr>
<td>7 CR3Filter</td>
<td></td>
<td>If (CPUID.(EAX=14H,ECX=0):EBX[0] = 1)</td>
</tr>
<tr>
<td>8 ToPA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 MTCEn</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)</td>
</tr>
<tr>
<td>10 TSCEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 DisRETC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 PTWEn</td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=1):EBX[4] = 1)</td>
</tr>
<tr>
<td>13 BranchEn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>17:14</td>
<td>MTCFreq.</td>
<td>18</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>22:19</td>
<td>CycThresh</td>
<td>23</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>PSBFreq</td>
<td>30:28</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>EventEn</td>
<td>35:32</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>39:36</td>
<td>ADDR0_CFG</td>
<td>43:40</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>47:44</td>
<td>ADDR3_CFG</td>
<td>54:48</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>DisTNT</td>
<td>56</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>54:48</td>
<td>PacketByteCnt</td>
<td>63:57</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 571H, 1393</td>
<td>IA32_RTIT_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing Status Register (R/W)</td>
<td></td>
<td>0</td>
<td>FilterEn (writes ignored).</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>ContexEn (writes ignored).</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>TriggerEn (writes ignored).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Stopped</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>PendPSB</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>PendToPAPMI</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[6] = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31:8</td>
<td>Reserved, must be zero.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>48:32</td>
<td>PacketByteCnt</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[1] &gt; 3)</td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:49</td>
<td>Reserved.</td>
<td>IA32_RTIT_CR3_MATCH</td>
<td></td>
</tr>
<tr>
<td>Register Address: 572H, 1394</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace Filter CR3 Match Register (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 580H, 1408</td>
<td>IA32_RTIT_ADDR0_A</td>
<td>Region 0 Start Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 0)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>SignExt_VA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 581H, 1409</td>
<td>IA32_RTIT_ADDR0_B</td>
<td>Region 0 End Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 0)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>SignExt_VA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 582H, 1410</td>
<td>IA32_RTIT_ADDR1_A</td>
<td>Region 1 Start Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 1)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>SignExt_VA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 583H, 1411</td>
<td>IA32_RTIT_ADDR1_B</td>
<td>Region 1 End Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 1)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>SignExt_VA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 584H, 1412</td>
<td>IA32_RTIT_ADDR2_A</td>
<td>Region 2 Start Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 2)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>SignExt_VA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 585H, 1413</td>
<td>IA32_RTIT_ADDR2_B</td>
<td>Region 2 End Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 2)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>SignExt_VA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 586H, 1414</td>
<td>IA32_RTIT_ADDR3_A</td>
<td>Region 3 Start Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 3)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:48 SignExt_VA</td>
<td>IA32_RTIT_ADDR3_B</td>
<td>Register Address: 587H, 1415</td>
<td>Region 3 End Address (R/W)</td>
<td>If (CPUID.(EAX=07H, ECX=1):EAX[2:0] &gt; 3)</td>
</tr>
<tr>
<td>47:0</td>
<td>Virtual Address.</td>
<td>63:48 SignExt_VA</td>
<td>Register Address: 600H, 1536</td>
<td>IA32_DS_AREA</td>
</tr>
<tr>
<td></td>
<td>DS Save Area (R/W)</td>
<td>Points to the linear address of the first byte of the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 20.6.3.4, &quot;Debug Store (DS) Mechanism.&quot;</td>
<td>If(CPUID.01H:EDX.DS[21] = 1</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>The linear address of the first byte of the DS buffer management area, if IA-32e mode is active.</td>
<td>31:0 The linear address of the first byte of the DS buffer management area, if not in IA-32e mode.</td>
<td>63:32 Reserved if not in IA-32e mode.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SH_STK_EN: When set to 1, enable shadow stacks at CPL3.</td>
<td>1</td>
<td>WR_SHSTK_EN: When set to 1, enables the WRSSD/WRSSQ instructions.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ENDBR_EN: When set to 1, enables indirect branch tracking.</td>
<td>3</td>
<td>LEG_IW_EN: Enable legacy compatibility treatment for indirect branch tracking.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NO_TRACK_EN: When set to 1, enables use of no-track prefix for indirect branch tracking.</td>
<td>5</td>
<td>SUPPRESS_DIS: When set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SUPPRESS: When set to 1, indirect branch tracking is suppressed. This bit can be written to 1 only if TRACKER is written as IDLE.</td>
<td>11</td>
<td>TRACKER: Value of the indirect branch tracking state machine. Values: IDLE (0), WAIT_FOR_ENDBRANCH(1).</td>
<td></td>
</tr>
<tr>
<td>63:12</td>
<td>EB_LEG_BITMAP_BASE: Linear address bits 63:12 of a legacy code page bitmap used for legacy compatibility when indirect branch tracking is enabled. If the processor does not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are reserved. On processors that support Intel 64 architecture this value cannot represent a non-canonical address. In protected mode, only 31:0 are used.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 6A2H, 1698</td>
<td>IA32_S_CET</td>
<td></td>
<td>Configure Supervisor Mode CET (R/W)</td>
<td>See IA32_U_CET (6A0H) for reference; similar format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A4H, 1700</td>
<td>IA32_PL0_SSP</td>
<td></td>
<td>Linear address to be loaded into SSP on transition to privilege level 0. (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A5H, 1701</td>
<td>IA32_PL1_SSP</td>
<td></td>
<td>Linear address to be loaded into SSP on transition to privilege level 1. (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A6H, 1702</td>
<td>IA32_PL2_SSP</td>
<td></td>
<td>Linear address to be loaded into SSP on transition to privilege level 2. (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A7H, 1703</td>
<td>IA32_PL3_SSP</td>
<td></td>
<td>Linear address to be loaded into SSP on transition to privilege level 3. (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A8H, 1704</td>
<td>IA32_INTERRUPT_SSP_TABLE_ADDR</td>
<td></td>
<td>Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=0H):ECX.CET_SS[07] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This MSR is not present on processors that do not support Intel 64 architecture. This field cannot represent a non-canonical address.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6E0H, 1760</td>
<td>IA32_TSC_DEADLINE</td>
<td></td>
<td>TSC Target of Local APIC's TSC Deadline Mode (R/W)</td>
<td>If CPUID.01H:ECX.[24] = 1</td>
</tr>
<tr>
<td>Register Address: 6E1H, 1761</td>
<td>IA32_PKRS</td>
<td></td>
<td>Specifies the PK permissions associated with each protection domain for supervisor pages (R/W)</td>
<td>If CPUID.(EAX=07H, ECX=0H):ECX.PKS [31] = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 770H, 1904</td>
<td>IA32_PM_ENABLE</td>
<td></td>
<td>Enable/disable HWP (R/W)</td>
<td>If CPUID.06H:EAX[7] = 1</td>
</tr>
</tbody>
</table>

31:0 For domain i (i between 0 and 15), bits 2i and 2i+1 contain the AD and WD permissions, respectively.

63:32 Reserved.
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>HWP_ENABLE (R/W1-Once)</td>
<td>See Section 15.4.2, “Enabling HWP.”</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>63:1</strong></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 771H, 1905</td>
<td>IA32_HWP_CAPABILITIES</td>
<td>HWP Performance Range Enumeration (R/O)</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>7:0</strong></td>
<td>Highest_Performance</td>
<td>See Section 15.4.3, “HWP Performance Range and Dynamic Capabilities.”</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>15:8</strong></td>
<td>Guaranteed_Performance</td>
<td>See Section 15.4.3, “HWP Performance Range and Dynamic Capabilities.”</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>23:16</strong></td>
<td>Most_Efficient_Performance</td>
<td>See Section 15.4.3, “HWP Performance Range and Dynamic Capabilities.”</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>31:24</strong></td>
<td>Lowest_Performance</td>
<td>See Section 15.4.3, “HWP Performance Range and Dynamic Capabilities.”</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>63:32</strong></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 772H, 1906</td>
<td>IA32_HWP_REQUEST_PKG</td>
<td>Power Management Control Hints for All Logical Processors in a Package (R/W)</td>
<td>If CPUID.06H:EAX.[11] = 1</td>
</tr>
<tr>
<td><strong>7:0</strong></td>
<td>Minimum_Performance</td>
<td>See Section 15.4.4, “Managing HWP.”</td>
<td>If CPUID.06H:EAX.[11] = 1</td>
</tr>
<tr>
<td><strong>15:8</strong></td>
<td>Maximum_Performance</td>
<td>See Section 15.4.4, “Managing HWP.”</td>
<td>If CPUID.06H:EAX.[11] = 1</td>
</tr>
<tr>
<td><strong>63:42</strong></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 773H, 1907</td>
<td>IA32_HWP_INTERRUPT</td>
<td>Control HWP Native Interrupts (R/W)</td>
<td>If CPUID.06H:EAX.[8] = 1</td>
</tr>
<tr>
<td><strong>0</strong></td>
<td>EN_Guaranteed_Performance_Change</td>
<td>See Section 15.4.6, “HWP Notifications.”</td>
<td>If CPUID.06H:EAX.[8] = 1</td>
</tr>
<tr>
<td><strong>1</strong></td>
<td>EN_Excursion_Minimum</td>
<td>See Section 15.4.6, “HWP Notifications.”</td>
<td>If CPUID.06H:EAX.[8] = 1</td>
</tr>
<tr>
<td><strong>63:2</strong></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 774H, 1908</td>
<td>IA32_HWP_REQUEST</td>
<td>Power Management Control Hints to a Logical Processor (R/W)</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
<tr>
<td><strong>7:0</strong></td>
<td>Minimum_Performance</td>
<td>See Section 15.4.4, “Managing HWP.”</td>
<td>If CPUID.06H:EAX.[7] = 1</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:8</td>
<td>Maximum_Performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.4.4, “Managing HwP.”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CPUID.06H:EAX.[7] = 1</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Desired_Performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.4.4, “Managing HwP.”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CPUID.06H:EAX.[7] = 1</td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Energy_Performance_Preference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.4.4, “Managing HwP.”</td>
<td></td>
</tr>
<tr>
<td>41:32</td>
<td>Activity_Window</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.4.4, “Managing HwP.”</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Package_Control</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.4.4, “Managing HwP.”</td>
<td></td>
</tr>
<tr>
<td>63:43</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 775H, 1909

IA32_PECI_Hwp_REQUEST_INFO

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Minimum_Performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(MINIMUM_PERFORMANCE): Used by OS to read</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the latest value of PECI minimum performance input.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default value is 0.</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum_Performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(MAXIMUM_PERFORMANCE): Used by OS to read</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the latest value of PECI maximum performance input.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default value is 0.</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Energy_Performance_Preference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ENERGY_PERFORMANCE_PREFERENCE): Used by OS to read</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the latest value of PECI Energy Performance Preference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>input. Default value is 0.</td>
<td></td>
</tr>
<tr>
<td>59:32</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>EPP PECI Override (EPP_PECI_OVERRIDE):</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates whether PECI is currently overriding the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy Performance Preference input. If set to '1',</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PECI is overriding the Energy Performance Preference</td>
<td></td>
</tr>
<tr>
<td></td>
<td>input. If clear (0), OS has control over Energy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Performance Preference input. Default value is 0.</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Max PECI Override (MAX_PECI_OVERRIDE):</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates whether PECI is currently overriding the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum Performance input. If set to '1', PECI is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>overriding the Maximum Performance input. If clear (0),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OS has control over Maximum Performance input. Default</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value is 0.</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Min PECI Override (MIN_PECI_OVERRIDE):</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates whether PECI is currently overriding the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minimum Performance input. If set to '1', PECI is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>overriding the Minimum Performance input. If clear (0),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OS has control over Minimum Performance input. Default</td>
<td></td>
</tr>
<tr>
<td></td>
<td>value is 0.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 776H, 1910

IA32_HWP_CTL
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKG_CTL_POLARITY</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines which HWP Request MSR is used whether Thread level or package level. When package MSR is used, the thread MSR valid bits define which thread MSR fields override the package. Default value is 0.</td>
<td>If CPUID.06H:EAX[22] = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>IA32_HWP_STATUS</td>
<td></td>
<td></td>
<td>Log bits indicating changes to Guaranteed &amp; excursions to Minimum (R/w)</td>
<td>If CPUID.06H:EAX[7] = 1</td>
</tr>
<tr>
<td>x2APIC_ID Register (R/O)</td>
<td></td>
<td>0</td>
<td>Guaranteed_Performance_Change (R/WC0)</td>
<td>If CPUID.06H:EAX[7] = 1</td>
</tr>
<tr>
<td>See Section 15.4.5, “HWP Feedback.”</td>
<td></td>
<td>1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>x2APIC Version Register (R/O)</td>
<td></td>
<td>2</td>
<td>Excursion_To_Minimum (R/WC0)</td>
<td>If CPUID.06H:EAX[7] = 1</td>
</tr>
<tr>
<td>See Section 15.4.5, “HWP Feedback.”</td>
<td></td>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_APICID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 802H, 2050</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Task Priority Register (R/W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 803H, 2051</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Processor Priority Register (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 808H, 2056</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC EOI Register (w/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 809H, 2059</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Logical Destination Register (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80FH, 2063</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Spurious Interrupt Vector Register (R/W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 810H, 2064</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits 31:0 (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 811H, 2065</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits 63:32 (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>812H, 2066</td>
<td>IA32_X2APIC_ISR2</td>
<td>x2APIC In-Service Register Bits 95:64 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>813H, 2067</td>
<td>IA32_X2APIC_ISR3</td>
<td>x2APIC In-Service Register Bits 127:96 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>814H, 2068</td>
<td>IA32_X2APIC_ISR4</td>
<td>x2APIC In-Service Register Bits 159:128 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>815H, 2069</td>
<td>IA32_X2APIC_ISR5</td>
<td>x2APIC In-Service Register Bits 191:160 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>816H, 2070</td>
<td>IA32_X2APIC_ISR6</td>
<td>x2APIC In-Service Register Bits 223:192 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>817H, 2071</td>
<td>IA32_X2APIC_ISR7</td>
<td>x2APIC In-Service Register Bits 255:224 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>818H, 2072</td>
<td>IA32_X2APIC_TMR0</td>
<td>x2APIC Trigger Mode Register Bits 31:0 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>81AH, 2074</td>
<td>IA32_X2APIC_TMR2</td>
<td>x2APIC Trigger Mode Register Bits 95:64 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>81BH, 2075</td>
<td>IA32_X2APIC_TMR3</td>
<td>x2APIC Trigger Mode Register Bits 127:96 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>81CH, 2076</td>
<td>IA32_X2APIC_TMR4</td>
<td>x2APIC Trigger Mode Register Bits 159:128 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>81DH, 2077</td>
<td>IA32_X2APIC_TMR5</td>
<td>x2APIC Trigger Mode Register Bits 191:160 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 820H, 2080</td>
<td>IA32_X2APIC_IRR0</td>
<td>x2APIC Interrupt Request Register Bits 31:0 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 821H, 2081</td>
<td>IA32_X2APIC_IRR1</td>
<td>x2APIC Interrupt Request Register Bits 63:32 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 822H, 2082</td>
<td>IA32_X2APIC_IRR2</td>
<td>x2APIC Interrupt Request Register Bits 95:64 (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 828H, 2088</td>
<td>IA32_X2APIC_ESR</td>
<td>x2APIC Error Status Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 82FH, 2095</td>
<td>IA32_X2APIC_LVT_CMCI</td>
<td>x2APIC LVT Corrected Machine Check Interrupt Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 830H, 2096</td>
<td>IA32_X2APIC_ICR</td>
<td>x2APIC Interrupt Command Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 832H, 2098</td>
<td>IA32_X2APIC_LVT_TIMER</td>
<td>x2APIC LVT Timer Interrupt Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 833H, 2099</td>
<td>IA32_X2APIC_LVT_THERMAL</td>
<td>x2APIC LVT Thermal Sensor Interrupt Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
<tr>
<td>Register Address: 834H, 2100</td>
<td>IA32_X2APIC_LVT_PMI</td>
<td>x2APIC LVT Performance Monitor Interrupt Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 835H, 2101</td>
<td>IA32_X2APIC_LVT_LINT0</td>
<td>x2APIC LVT LINT0 Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 836H, 2102</td>
<td>IA32_X2APIC_LVT_LINT1</td>
<td>x2APIC LVT LINT1 Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 837H, 2103</td>
<td>IA32_X2APIC_LVT_ERROR</td>
<td>x2APIC LVT Error Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 838H, 2104</td>
<td>IA32_X2APIC_INIT_COUNT</td>
<td>x2APIC Initial Count Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 839H, 2105</td>
<td>IA32_X2APIC_CUR_COUNT</td>
<td>x2APIC Current Count Register (R/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 83EH, 2110</td>
<td>IA32_X2APIC_DIV_CONF</td>
<td>x2APIC Divide Configuration Register (R/w)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 83FH, 2111</td>
<td>IA32_X2APIC_SELF_IPI</td>
<td>x2APIC Self IPI Register (W/O)</td>
<td>If CPUID.01H:ECX.[21] = 1 &amp;&amp; IA32_APIC_BASE.[10] = 1</td>
<td></td>
</tr>
<tr>
<td>Register Address: 981H, 2433</td>
<td>IA32_TME_CAPABILITY</td>
<td>Memory Encryption Capability MSR</td>
<td>If CPUID.07H:ECX.[13] = 1</td>
<td></td>
</tr>
</tbody>
</table>

#### Memory Encryption Capability MSR

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Support for AES-XTS 128-bit encryption algorithm. (NIST standard)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Support for AES-XTS 128-bit encryption with integrity algorithm.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Support for AES-XTS 256-bit encryption algorithm.</td>
<td></td>
</tr>
<tr>
<td>29:3</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>SUPPORT_I32_TME_CLEAR_SAVED_KEY Support for the IA32_TME_CLEAR_SAVED_KEY MSR.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>TME encryption bypass supported.</td>
<td></td>
</tr>
<tr>
<td>35:32</td>
<td>MK_TME_MAX_KEYID_BITS Number of bits which can be allocated for usage as key identifiers for multi-key memory encryption. 4 bits allow for a maximum value of 15, which could address 32K keys. Zero if TME-MK is not supported.</td>
<td></td>
</tr>
<tr>
<td>50:36</td>
<td>MK_TME_MAX_KEYS Indicates the maximum number of keys which are available for usage. This value may not be a power of 2. KeyId 0 is specially reserved and is not accounted for in this field.</td>
<td></td>
</tr>
<tr>
<td>63:51</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>982H, 2434</td>
<td>IA32_TME_ACTIVATE</td>
<td></td>
<td>Memory Encryption Activation MSR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This MSR is used to lock the MSRs listed below. Any write to the following MSRs will be ignored after they are locked. The lock is reset when CPU is reset.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IA32_TME_ACTIVATE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IA32_TME_EXCLUDE_MASK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IA32_TME_EXCLUDE_BASE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note that IA32_TME_EXCLUDE_MASK and IA32_TME_EXCLUDE_BASE must be configured before IA32_TME_ACTIVATE.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If CPUID.07H:ECX.[13] = 1</td>
</tr>
<tr>
<td>0</td>
<td>Lock R/O – Will be set upon successful WRMSR (or first SMI); written value ignored.</td>
<td></td>
<td>Hardware Encryption Enable</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Hardware Encryption Enable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This bit also enables TME-MK; TME-MK cannot be enabled without enabling encryption hardware.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Key Select</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0: Create a new TME key (expected cold/warm boot). 1: Restore the TME key from storage (Expected when resume from standby).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Save TME Key for Standby</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save key into storage to be used when resume from standby. Note: This may not be supported in all processors.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td>TME Policy/Encryption Algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Only algorithms enumerated in IA32_TME_CAPABILITY are allowed. For example: 0000 – AES-XTS-128. 0001 – AES-XTS-128 with integrity. 0010 – AES-XTS-256. Other values are invalid.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>TME Encryption Bypass Enable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When encryption hardware is enabled:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Total Memory Encryption is enabled using a CPU generated ephemeral key based on a hardware random number generator when this bit is set to 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Total Memory Encryption is bypassed (no encryption/decryption for KeyID0) when this bit is set to 1. Software must inspect Hardware Encryption Enable (bit 1) and TME encryption bypass Enable (bit 31) to determine if TME encryption is enabled.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>35:32</strong></td>
<td>MK_TME_KEYID_BITS</td>
<td>Reserved if TME-MK is not enumerated, otherwise:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The number of key identifier bits to allocate to TME-MK usage. Similar to enumeration, this is an encoded value.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Writing a value greater than MK_TME_MAX_KEYID_BITS will result in #GP.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Writing a non-zero value to this field will #GP if bit 1 of EAX (Hardware Encryption Enable) is not also set to ‘1, as encryption hardware must be enabled to use TME-MK.</td>
<td>Example: To support 255 keys, this field would be set to a value of 8.</td>
</tr>
<tr>
<td><strong>47:36</strong></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>63:48</strong></td>
<td>MK_TME_CRYPTO_ALGS</td>
<td>Reserved if TME-MK is not enumerated, otherwise:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 48: AES-XTS 128.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 49: AES-XTS 128 with integrity.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 50: AES-XTS 256.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 63:51: Reserved (#GP)</td>
<td>Bitmask for BIOS to set which encryption algorithms are allowed for TME-MK, would be later enforced by the key loading ISA (‘1 = allowed).</td>
</tr>
</tbody>
</table>

**Register Address:** 983H, 2435 | **IA32_TME_EXCLUDE_MASK** |
---|---|
Memory Encryption Exclude Mask | If CPUID.07H:ECX.[13] = 1 |

| **10:0**  | Reserved | | |
| **11**    | Reserved | Enable: When set to ‘1’, then TME_EXCLUDE_BASE and TME_EXCLUDE_MASK are used to define an exclusion region for TME/TME-MK (for KeyID=0). | |

**Register Address:** 984H, 2436 | **IA32_TME_EXCLUDE_BASE** |
---|---|
Memory Encryption Exclude Base | If CPUID.07H:ECX.[13] = 1 |

| **MAXPHYSADDR-1:12** | TMEEMASK: This field indicates the bits that must match TMEBASE in order to qualify as a TME/TME-MK (for KeyID=0) exclusion memory range access. | |
| **63:MAXPHYSADDR**  | Reserved; must be zero. | |

**Register Address:** 985H, 2437 | **IA32_UINTR_RR** |
---|---|
User Interrupt Request Register (R/W) | If CPUID.07H:EDX[13]=1 |

| **63:0**  | UIRR | Bitmap of requested user interrupt vectors. | |

**Register Address:** 986H, 2438 | **IA32_UINTR_HANDLER** |
---|---|
User Interrupt Handler Address (R/W) | If CPUID.07H:EDX[13]=1 |
<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>UIHANDLER</td>
<td>User interrupt handler linear address.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 987H, 2439</td>
<td>IA32_UINTR_STACKADJUST</td>
</tr>
<tr>
<td></td>
<td>User Interrupt Stack Adjustment (R/w)</td>
<td>IF CPUID.07H.01H:EDX[13]=1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LOAD_RSP</td>
<td>User interrupt stack mode.</td>
<td></td>
</tr>
<tr>
<td>2:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>STACK_ADJUST</td>
<td>Stack adjust value.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 988H, 2440</td>
<td>IA32_UINTR_MISC</td>
</tr>
<tr>
<td></td>
<td>User-Interrupt Target-Table Size and Notification Vector (R/w)</td>
<td>IF CPUID.07H.01H:EDX[13]=1</td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>UITTSZ</td>
<td>The highest index of a valid entry in the user-interrupt target table. Valid entries are indices 0..UITTSZ (inclusive).</td>
<td></td>
</tr>
<tr>
<td>39:32</td>
<td>UINV</td>
<td>User-interrupt notification vector.</td>
<td></td>
</tr>
<tr>
<td>63:40</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 989H, 2441</td>
<td>IA32_UINTR_PD</td>
</tr>
<tr>
<td></td>
<td>User Interrupt PID Address (R/w)</td>
<td>IF CPUID.07H.01H:EDX[13]=1</td>
<td></td>
</tr>
<tr>
<td>5:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:6</td>
<td>UPIDADDR</td>
<td>User-interrupt notification processing accesses a UPID at this linear address.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 98AH, 2442</td>
<td>IA32_UINTR_TT</td>
</tr>
<tr>
<td></td>
<td>User-Interrupt Target Table (R/w)</td>
<td>IF CPUID.07H.01H:EDX[13]=1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SENDUIPI_ENABLE</td>
<td>User-interrupt target table is valid.</td>
<td></td>
</tr>
<tr>
<td>3:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>UITTADDR</td>
<td>User-interrupt target table base linear address.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 990H, 2448</td>
<td>IA32_COPY_STATUS5</td>
</tr>
<tr>
<td></td>
<td>Status of Most Recent Platform to Local or Local to Platform Copies (R/O)</td>
<td>IF ((CPUID.19H:EBX[4] = 1) &amp;&amp; (CPUID.(07H,0).ECX[23] = 1))</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>IwKEY_COPY_SUCCESSFUL</td>
<td>Status of most recent copy to or from IwKeyBackup.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 991H, 2449</td>
<td>IA32_IWKEYBACKUP_STATUS5</td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cleared when a write to IWKeyBackup is initiated, and then set when the latest write of IWKeyBackup has been written to storage that persists across S3/S4 sleep state. If S3/S4 is entered between when an IWKeyBackup write occurs and when this bit is set, then IWKeyBackup may not be recovered after S3/S4 exit. During S3/S4 sleep state exit (system wake up), this bit is cleared. It is set again when IWKeyBackup is restored from persistent storage and thus available to be copied to IWKey using IA32_COPY_PLATFORM_TO_LOCAL MSR. Another write to IWKeyBackup (via IA32_COPY_LOCAL_TO_PLATFORM MSR) may fail if a previous write has not yet set this bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Updated prior to backup/restore valid being set. Set when an error is encountered while backing up or restoring a key to persistent storage.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set after the previous backup operation has been consumed by the platform. This does not indicate that the system is ready for a second IWKeyBackup write as the previous IWKeyBackup write may still need to set Backup/restore valid.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 9FBH, 2555</td>
<td>IA32_TME_CLEAR_SAVED_KEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_TME_CLEAR_SAVED_KEY (W/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>TME_CLEAR_SAVED_KEY</td>
<td>Clear saved TME keys.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C80H, 3200</td>
<td>IA32_DEBUG_INTERFACE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Silicon Debug Feature Control (R/w)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Enable (R/W)</td>
<td>BIOS set 1 to enable Silicon debug features. Default is 0.</td>
<td>IF CPUID.01H:ECX[11] = 1</td>
</tr>
<tr>
<td>29:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Lock (R/W): If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.</td>
<td>IF CPUID.01H:ECX[11] = 1</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Debug Occurred (R/O): This “sticky bit” is set by hardware to indicate the status of bit 0. Default is 0.</td>
<td>IF CPUID.01H:ECX[11] = 1</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C81H, 3201</td>
<td>IA32_L3_QOS_CFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 QOS Configuration (R/w)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Enable (R/W)</td>
<td>Set 1 to enable L3 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode.</td>
<td>IF (CPUID.(EAX=10H, ECX=1):ECX[2] = 1)</td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved. Attempts to write to reserved bits result in a #GP(0).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C82H, 3202</td>
<td>IA32_L2_QOS_CFG</td>
<td>L2 QOS Configuration (R/w)</td>
<td>0</td>
<td>Enable (R/w)</td>
</tr>
<tr>
<td>63:1 Reserved. Attempts to write to reserved bits result in a #GP(0).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C83H, 3203</td>
<td>IA32_L3_IO_QOS_CFG</td>
<td>L3 I/O QOS Configuration (R/w)</td>
<td>0</td>
<td>L3 I/O RDT Allocation Enable.</td>
</tr>
<tr>
<td>1</td>
<td>L3 I/O RDT Monitoring Enable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C8DH, 3213</td>
<td>IA32_QM_EVTSEL</td>
<td>Monitoring Event Select Register (R/w)</td>
<td>7:0</td>
<td>Event ID: ID of a supported monitoring event to report via IA32_QM_CTR.</td>
</tr>
<tr>
<td>31:8 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N+31:32</td>
<td>Resource Monitoring ID: ID for monitoring hardware to report monitored data via IA32_QM_CTR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:N+32 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C8EH, 3214</td>
<td>IA32_QM_CTR</td>
<td>Monitoring Counter Register (R/O)</td>
<td>61:0</td>
<td>Resource Monitored Data.</td>
</tr>
<tr>
<td>62</td>
<td>Unavailable: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Error: If 1, indicates an unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C90H−D8FH, 3216−3471</td>
<td>Reserved MSR Address Space for CAT Mask Registers</td>
<td>Resource Association Register (R/W)</td>
<td>N-1:0</td>
<td>Resource Monitoring ID (R/W): ID for monitoring hardware to track internal operation, e.g., memory access.</td>
</tr>
<tr>
<td>31:N Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>COS (R/W): The class of service (COS) to enforce (on writes); returns the current COS when read.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: C90H, 3216</td>
<td>IA32_L3_MASK_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 CAT Mask for COS0 (R/w)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>Capacity Bit Mask (R/w)</td>
<td></td>
<td>If (CPUID.(EAX=10H, ECX=0H):EBX[1] != 0)</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C90H+n, 3216+n</td>
<td>IA32_L3_MASK_n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 CAT Mask for COSn (R/w)</td>
<td></td>
<td></td>
<td>n = CPUID.(EAX=10H, ECX=1H):EDX[15:0]</td>
</tr>
<tr>
<td>31:0</td>
<td>Capacity Bit Mask (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D10H, 3344</td>
<td>IA32_L2_MASK_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 CAT Mask for COS0 (R/w)</td>
<td></td>
<td></td>
<td>If (CPUID.(EAX=10H, ECX=0H):EBX[2] != 0)</td>
</tr>
<tr>
<td>31:0</td>
<td>Capacity Bit Mask (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D10H+n, 3344+n</td>
<td>IA32_L2_MASK_n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 CAT Mask for COSn (R/w)</td>
<td></td>
<td></td>
<td>n = CPUID.(EAX=10H, ECX=2H):EDX[15:0]</td>
</tr>
<tr>
<td>31:0</td>
<td>Capacity Bit Mask (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D90H, 3472</td>
<td>IA32_BNDCFGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supervisor State of MPX Configuration (R/W)</td>
<td></td>
<td></td>
<td>If (CPUID.(EAX=07H, ECX=0H):EBX[14] = 1)</td>
</tr>
<tr>
<td>0</td>
<td>EN: Enable Intel MPX in supervisor mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BNDPRESERVE: Preserve the bounds registers for near branch instructions in the absence of the BND prefix.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:2</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:12</td>
<td>Base Address of Bound Directory.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D91H, 3473</td>
<td>IA32_COPY_LOCAL_TO_PLATFORM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>IwKeyBackup</td>
<td>Copy IwKey to IwKeyBackup.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D92H, 3474</td>
<td>IA32_COPY_PLATFORM_TO_LOCAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>IwKeyBackup</td>
<td>Copy IwKey to IwKeyBackup.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit Fields</strong></td>
<td><strong>MSR/Bit Description</strong></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: D93H, 3475 | IA32_PASID |

Process Address Space Identifier. (R/W)

| 19:0 | Process address space identifier (PASID). Specifies the PASID of the currently running software thread. |
| 30:20 | Reserved. |
| 31 | Valid. Execution of ENQCMD causes a #GP if this bit is clear. |
| 63:32 | Reserved. |

Register Address: DA0H, 3488 | IA32_XSS |

Extended Supervisor State Mask (R/w)

| 7:0 | Reserved. |
| 8 | PT State (R/w) |
| 9 | Reserved. |
| 10 | PASID State (R/w) |
| 11 | CET_U State (R/w) |
| 12 | CET_S State (R/w) |
| 13 | HDC State (R/w) |
| 14 | UNTR State (R/w) |
| 15 | LBR State (R/w) |
| 16 | Hwp State (R/w) |
| 63:17 | Reserved. |

Register Address: DB0H, 3504 | IA32_PKG_HDC_CTL |

Package Level Enable/disable HDC (R/W)

| 0 | HDC_Pkg_Enable (R/w) |
| 63:1 | Reserved. |

Register Address: DB1H, 3505 | IA32_PM_CTL1 |

Enable/disable HWP (R/W)

| 0 | HDC_Allow_Block (R/W) |
| 63:1 | Reserved. |

Register Address: DB2H, 3506 | IA32_THREADSTALL |
### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-Logical_Processor_ID HDC Idle Residency (R/O)</td>
<td>If CPUID.06H:EAX.[13] = 1</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Stall_Cycle_Cnt (R/W)</td>
<td>If CPUID.06H:EAX.[13] = 1</td>
</tr>
<tr>
<td></td>
<td>Stalled cycles due to HDC forced idle on this logical processor. See Section 15.5.4.1.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1200H–121FH, 4608–4639</td>
<td>IA32_LBR_x_INFO</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record Entry X Info Register (R/W)</td>
<td>An attempt to read or write IA32_LBR_x_INFO such that x ≥ IA32_LBR_DEPTH.DEPTH will #GP.</td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>CYC_CNT</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td>The elapsed CPU cycles (saturating) since the last LBR was recorded. See Section 18.1.3.3.</td>
<td></td>
</tr>
<tr>
<td>55:16</td>
<td>Undefined, may be zero or non-zero. Writes of non-zero values do not fault, but reads may return a different value.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>59:56</td>
<td>BR_TYPE</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td>The branch type recorded by this LBR. Encodings:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000B: COND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001B: JMP Indirect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010B: JMP Direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011B: CALL Indirect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100B: CALL Direct</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101B: RET</td>
<td></td>
</tr>
<tr>
<td></td>
<td>011xB: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1xxxB: Other Branch</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>CYC_CNT_VALID</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td>CYC_CNT value is valid. See Section 19.1.3.3.</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>TSX_ABORT</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td>This LBR record is a TSX abort. On processors that do not support Intel TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>IN_TSX</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td>This LBR record records a branch that retired during a TSX transaction. On processors that do not support Intel TSX (CPUID.07H.EBX.HLE[bit 4]=0 and CPUID.07H.EBX.RTM[bit 11]=0), this bit is undefined.</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>MISPRED</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td></td>
<td>The recorded branch direction (conditional branch) or target (indirect branch) was mispredicted.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1406H, 5126</td>
<td>IA32_MCU_CONTROL</td>
<td></td>
</tr>
<tr>
<td>MCU Control (R/W)</td>
<td>Controls the behavior of the Microcode Update Trigger MSR, IA32_BIOS_UPDT_TRIG.</td>
<td>If CPUID.07H.OH:EDX[29]=1 &amp;&amp; MSR.IA32_ARCH_CAPABILITIES.MCU_CONTROL=1</td>
</tr>
<tr>
<td>0</td>
<td>LOCK</td>
<td>Once set, further writes to this MSR will cause a #GP(0) fault. Bypassed during SMM if EN_SMM_BYPASS (bit 2) is set.</td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIS_MCU_LOAD</td>
<td>If this bit is set on a given logical processor, then any subsequent attempts to load a microcode update by that logical processor will be silently dropped (WRMSR 0x79 has no effect).</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EN_SMM_BYPASS</td>
<td>If set, then writes to IA32_MCUMCUCONTROL are allowed during SMM regardless of the LOCK bit. This enables BIOS to Opt-In to the SMM Bypass functionality.</td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 14CEH, 5326 | IA32_LBR_CTL

Last Branch Record Enabling and Configuration Register (R/W)

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LBREn</td>
<td>When set, enables LBR recording. Reset Value: 0</td>
</tr>
<tr>
<td>1</td>
<td>OS</td>
<td>When set, allows LBR recording when CPL == 0. Reset Value: 0</td>
</tr>
<tr>
<td>2</td>
<td>USR</td>
<td>When set, allows LBR recording when CPL != 0. Reset Value: 0</td>
</tr>
<tr>
<td>3</td>
<td>CALL_STACK</td>
<td>When set, records branches in call-stack mode. See Section 19.1.2.4. Reset Value: 0</td>
</tr>
<tr>
<td>15:4</td>
<td>Reserved.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>16</td>
<td>COND</td>
<td>When set, records taken conditional branches. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>17</td>
<td>NEAR_REL_JMP</td>
<td>When set, records near relative JMPs. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>18</td>
<td>NEAR_IND_JMP</td>
<td>When set, records near indirect JMPs. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>19</td>
<td>NEAR_REL_CALL</td>
<td>When set, records near relative CALLs. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>20</td>
<td>NEAR_IND_CALL</td>
<td>When set, records near indirect CALLs. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>21</td>
<td>NEAR_RET</td>
<td>When set, records near RETs. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>22</td>
<td>OTHER_BRANCH</td>
<td>When set, records other branches. See Section 19.1.2.3.</td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 14CFH, 5327 | IA32_LBR_DEPTH

Last Branch Record Maximum Stack Depth Register (R/W)
### Model-Specific Registers (MSRs)

#### Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>N:0</td>
<td>DEPTH</td>
<td>The number of LBRs to be used for recording. Supported values are indicated by the bitmap in CPUID.(EAX=01CH,ECX=0):EAX[7:0]. The reset value will match the maximum supported by the CPU. Writes of unsupported values will #GP fault.</td>
<td>Reset Value: Varies</td>
</tr>
<tr>
<td>63:N+1</td>
<td>Reserved.</td>
<td></td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>Register Address: 1500H–151FH, 5376–5407</td>
<td>IA32_LBR_x_FROM_IP</td>
<td>Last Branch Record entry X source IP register (R/W). An attempt to read or write IA32_LBR_x_FROM_IP such that x ≥ IA32_LBR_DEPTH.DEPTH will #GP.</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>FROM_IP</td>
<td>The source IP of the recorded branch or event, in canonical form. Writes to bits above MAXLINADDR-1 are ignored.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>Register Address: 1600H–161FH, 5632–5663</td>
<td>IA32_LBR_x_TO_IP</td>
<td>Last Branch Record Entry X Destination IP Register (R/W) An attempt to read or write IA32_LBR_x_TO_IP such that x ≥ IA32_LBR_DEPTH.DEPTH will #GP.</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>TO_IP</td>
<td>The destination IP of the recorded branch or event, in canonical form. Writes to bits above MAXLINADDR-1 are ignored.</td>
<td>Reset Value: 0</td>
</tr>
<tr>
<td>Register Address: 17D0H, 6096</td>
<td>IA32_HW_FEEDBACK_PTR</td>
<td>Hardware Feedback Interface Pointer</td>
<td>If CPUID.06H:EAX.[19] = 1</td>
</tr>
<tr>
<td>0</td>
<td>Valid (R/W)</td>
<td>When set to 1, indicates a valid pointer is programmed into the ADDR field of the MSR.</td>
<td></td>
</tr>
<tr>
<td>11:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(MAXPHYADDR-1):12</td>
<td>ADDR (R/W)</td>
<td>Physical address of the page frame of the first page of the hardware feedback interface structure.</td>
<td></td>
</tr>
<tr>
<td>63:MAXPHYADDR</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17D1H, 6097</td>
<td>IA32_HW_FEEDBACK_CONFIG</td>
<td>Hardware Feedback Interface Configuration</td>
<td>If CPUID.06H:EAX.[19] = 1</td>
</tr>
<tr>
<td>0</td>
<td>Enable (R/w)</td>
<td>When set to 1, enables the hardware feedback interface.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17D2H, 6098</td>
<td>IA32_THREAD_FEEDBACK_CHAR</td>
<td>Thread Feedback Characteristics (R/O)</td>
<td>If CPUID.06H:EAX.[23] = 1</td>
</tr>
<tr>
<td>7:0</td>
<td>Application Class ID, pointing into the Intel Thread Director structure.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 Valid bit. When set to 1 the OS Scheduler can use the Class ID (in bits 7:0) for its scheduling decisions. If this bit is 0, the Class ID field should be ignored. It is recommended that the OS uses the last known Class ID of the software thread for its scheduling decisions.</td>
<td></td>
<td>IA32_HW_FEEDBACK_THREAD_CONFIG</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware Feedback Thread Configuration (R/W)</td>
<td></td>
</tr>
<tr>
<td>0 Enables Intel Thread Director. When set to 1, logical processor scope Intel Thread Director is enabled. Default is 0 (disabled).</td>
<td></td>
<td>IA32_HRESET_ENABLE</td>
<td>History Reset Enable (R/w)</td>
</tr>
<tr>
<td>63:1 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 17DAH, 6106</td>
<td></td>
</tr>
<tr>
<td>0 Enable reset of the Intel Thread Director history.</td>
<td></td>
<td>IA32_UARCH_MISC_CTL</td>
<td>IA32_UARCH_MISC_CTL If IA32_ARCH_CAPABILITIES[12]=1</td>
</tr>
<tr>
<td>31:1 Reserved for other capabilities that can be reset by the HRESET instruction.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 1B01H, 6913</td>
<td></td>
</tr>
<tr>
<td>If IA32_ARCH_CAPABILITIES[12]=1</td>
<td></td>
<td>IA32_EFER</td>
<td>Extended Feature Enables</td>
</tr>
<tr>
<td>0 SYSCALL Enable: IA32_EFER.SCE (R/w) Enables SYSCALL/SYSRET instructions in 64-bit mode.</td>
<td></td>
<td></td>
<td>If ( CPUID.80000001H:EDX.[20]</td>
</tr>
<tr>
<td>7:1 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 IA-32e Mode Enable: IA32_EFER.LME (R/w) Enables IA-32e mode operation.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 IA-32e Mode Active: IA32_EFER.LMA (R) Indicates IA-32e mode is active when set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 Execute Disable Bit Enable: IA32_EFER.NXE (R/w)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:12 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: C000_0081H</td>
<td></td>
</tr>
<tr>
<td>If CPUID.80000001H:EDX.[29] = 1</td>
<td></td>
<td>IA32_STAR</td>
<td>System Call Target Address (R/w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: C000_0082H</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_LSTAR</td>
<td></td>
</tr>
</tbody>
</table>
MODEL-SPECIFIC REGISTERS (MSRS)

Table 2-2. IA-32 Architectural MSRs (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Architectural MSR Name (Former MSR Name)</th>
<th>Bit Fields</th>
<th>MSR/Bit Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA-32e Mode System Call Target Address (R/W)</td>
<td>IA32_CSTAR</td>
<td>IA32_0083H</td>
<td>Target RIP for the called procedure when SYSCALL is executed in 64-bit mode.</td>
<td>If CPUID.80000001:EDX.[29] = 1</td>
</tr>
<tr>
<td>IA-32e Mode System Call Target Address (R/W)</td>
<td>IA32_FMASK</td>
<td>IA32_0084H</td>
<td>Not used, as the SYSCALL instruction is not recognized in compatibility mode.</td>
<td>If CPUID.80000001:EDX.[29] = 1</td>
</tr>
<tr>
<td>System Call Flag Mask (R/W)</td>
<td>IA32_FS_BASE</td>
<td>IA32_0100H</td>
<td>Map of BASE Address of FS (R/W)</td>
<td>If CPUID.80000001:EDX.[29] = 1</td>
</tr>
<tr>
<td>Map of BASE Address of GS (R/W)</td>
<td>IA32_KERNEL_BASE</td>
<td>IA32_0101H</td>
<td>Map of BASE Address of GS (R/W)</td>
<td>If CPUID.80000001:EDX.[29] = 1</td>
</tr>
<tr>
<td>Swap Target of BASE Address of GS (R/W)</td>
<td>IA32_TSC_AUX</td>
<td>IA32_0102H</td>
<td>Swap Target of BASE Address of GS (R/W)</td>
<td>If CPUID.80000001:EDX.[29] = 1</td>
</tr>
<tr>
<td>Auxiliary TSC (R/W)</td>
<td></td>
<td>IA32_0103H</td>
<td>Auxiliary TSC (R/W)</td>
<td>If CPUID.80000001H:EDX[27] = 1 or CPUID.(EAX=7,ECX=0):ECX[bit 22] = 1</td>
</tr>
<tr>
<td>31:0 AUX: Auxiliary signature of TSC.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. Some older processors may have supported this MSR as model-specific and do not enumerate it with CPUID.
2. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.
3. The _ADDR MSRs may or may not be present; this depends on flag settings in IA32_MCi_STATUS. See Section 16.3.2.3 and Section 16.3.2.4 for more information.
4. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].
5. Further details on Key Locker and usage of this MSR can be found here: https://software.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

2.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 2-3 lists model-specific registers (MSRs) for the Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 2-3. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_0FH, see Table 2-1.

MSRs listed in Table 2-2 and Table 2-3 are also supported by processors based on the Enhanced Intel Core microarchitecture. Processors based on the Enhanced Intel Core microarchitecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_17H.
The column “Shared/Unique” applies to multi-core processors based on Intel Core microarchitecture. “Unique” means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. “Shared” means the MSR or the bit field in an MSR address governs the operation of both processor cores.

### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 0H, 0</td>
<td>IA32_P5_MC_ADDR</td>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>IA32_P5_MC_TYPE</td>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6H, 6</td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td>See Section 9.10.5, “Monitor/Mwait Address Range Determination,” and Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
<td>See Section 18.17, “Time-Stamp Counter,” and Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform ID (R)</td>
<td>IA32_PLATFORM_ID</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>MSR_PLATFORM_ID</td>
<td>Model Specific Platform ID (R)</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:8 Maximum Qualified Ratio (R)</td>
<td>The maximum allowed bus ratio.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49:13 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52:50 See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:53 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1BH, 27</td>
<td>IA32_APIC_BASE</td>
<td>See Section 11.4.4, “Local APIC Status and Location,” and Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2AH, 42</td>
<td>MSR_EBL_CR_POWERON</td>
<td>Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Data Error Checking Enable (R/w)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processors implement R/w.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Response Error Checking Enable (R/w)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processor implements R/w.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 MCERR# Drive Enable (R/w)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processors implement R/w.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/ Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Address Parity Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td>Note: Not all processors implement R/W.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BINIT# Driver Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td>Note: Not all processors implement R/W.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Output Tri-state Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Execute BIST (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MCERR# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Intel TXT Capable Chipset. (R/O)</td>
<td>1 = Present; 0 = Not Present.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BINIT# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 MByte Power on Reset Vector (R/O)</td>
<td>1 = 1 MByte; 0 = 4 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:16</td>
<td>APIC Cluster ID (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>N/2 Non-Integer Bus Ratio (R/O)</td>
<td>0 = Integer ratio; 1 = Non-integer ratio.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Symmetric Arbitration ID (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26:22</td>
<td>Integer Bus Frequency Ratio (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Register Address: 3AH, 58</strong></td>
<td><strong>MSR_FEATURE_CONTROL</strong></td>
<td>Control Features in Intel 64 Processor (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SMRR Enable (R/WL)</td>
<td>When this bit is set and the lock bit is set, this makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.</td>
<td>Unique</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 40H, 64 **MSR_LASTBRANCH_0_FROM_IP**
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 0 From IP (R/W)</td>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td>One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also:</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>• Section 18.5.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41H, 65</td>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td>Last Branch Record 1 From IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42H, 66</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td>Last Branch Record 2 From IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43H, 67</td>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td>Last Branch Record 3 From IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 60H, 96</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td>Last Branch Record 0 To IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 61H, 97</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td>Last Branch Record 1 To IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 62H, 98</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td>Last Branch Record 2 To IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 63H, 99</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td>Last Branch Record 3 To IP (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 79H, 121</td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td>BIOS Update Trigger Register (W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 8BH, 139</td>
<td>IA32_BIOS_SIGN_ID</td>
<td>BIOS Update Signature ID (R/W)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: A0H, 160</td>
<td>MSR_SMRR_PHYSBASE</td>
<td>System Management Mode Base Address register (WO in SMM)</td>
<td>Unique</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>Model-specific implementation of SMRR-like interface, read visible and write only in SMM.</td>
<td></td>
</tr>
<tr>
<td>11:0</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>31:12</td>
<td></td>
<td>PhysBase: SMRR physical Base Address.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: A1H, 161</td>
<td>MSR_SMRR_PHYSMASK</td>
<td>System Management Mode Physical Address Mask register (WO in SMM)</td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Model-specific implementation of SMRR-like interface, read visible and write only in SMM.</td>
<td></td>
</tr>
<tr>
<td>10:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Valid: Physical address base and range mask are valid.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:12</td>
<td>PhysMask: SMRR physical address range mask.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C1H, 193</td>
<td>IA32_PMC0</td>
<td>Performance Counter Register</td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C2H, 194</td>
<td>IA32_PMC1</td>
<td>Performance Counter Register</td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: CDH, 205</td>
<td>MSR_FSB_FREQ</td>
<td>Scaleable Bus Speed (R/O)</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field indicates the intended scalable bus clock speed for processors based on Intel Core microarchitecture.</td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>101B: 100 MHz (FSB 400)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>001B: 133 MHz (FSB 533)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>011B: 167 MHz (FSB 667)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>010B: 200 MHz (FSB 800)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000B: 267 MHz (FSB 1067)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100B: 333 MHz (FSB 1333)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: CDH, 205</td>
<td>MSR_FSB_FREQ</td>
<td>Scaleable Bus Speed (R/O)</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This field indicates the intended scalable bus clock speed for processors based on Enhanced Intel Core microarchitecture.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/ Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td></td>
<td>101B: 100 MHz (FSB 400)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>001B: 133 MHz (FSB 533)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>011B: 167 MHz (FSB 667)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>010B: 200 MHz (FSB 800)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000B: 267 MHz (FSB 1067)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100B: 333 MHz (FSB 1333)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>110B: 400 MHz (FSB 1600)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7H, 231</td>
<td>IA32_MPERF</td>
<td>Maximum Performance Frequency Clock Count (R/W)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E8H, 232</td>
<td>IA32_APERF</td>
<td>Actual Performance Frequency Clock Count (R/W)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td>IA32_MTRRCAP</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>SMRR Capability Using MSR 0A0H and 0A1H (R)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>0</td>
<td>RIPV</td>
<td>R I P V</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.

When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.

Reserved.

Current performance status. See Section 15.1.1, "Software Interface For Initiating Performance State Transitions.”

Reserved.

If set, XE operation is enabled. Default is cleared.

Reserved.

Indicates maximum bus ratio configured for the processor.

Reserved.

Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture.

Reserved.

See Table 2-2.

See Table 2-2.

See Table 2-2.

See Table 2-2.

See Table 2-2.

IA32_PERF_CTL

Unique

IA32_CLOCK_MODULATION

Unique

IA32_THERM_INTERRUPT MSR was originally named IA32_THERM_CONTROL MSR.

| Register Address: 199H, 411 | IA32_THERM_INTERRUPT | Unique |

| Register Address: 198H, 408 | MSR_PERF_STATUS | Shared |

| Current performance status. See Section 15.1.1, “Software Interface For Initiating Performance State Transitions.” | Shared |

| 15:0 | Current Performance State Value |

| 30:16 | Reserved. |

| 31 | XE Operation (R/O). If set, XE operation is enabled. Default is cleared. |

| 39:32 | Reserved. |

| 44:40 | Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor. |

| 45 | Reserved. |

| 46 | Non-Integer Bus Ratio (R/O) Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture. |

| 63:47 | Reserved. |

Register Address: 199H, 409 IA32_PERF_CTL

See Table 2-2. Unique

Register Address: 19AH, 410 IA32 CLOCK_MODULATION

Clock Modulation (R/W) See Table 2-2. Unique

IA32 CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

Register Address: 198H, 411 IA32_THERM_INTERRUPT |
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/ Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thermal Interrupt Control (R/W)</strong></td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td><strong>Register Address: 19CH, 412</strong></td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thermal Monitor Status (R/W)</strong></td>
<td>See Table 2-2.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td><strong>Register Address: 19DH, 413</strong></td>
<td>MSR_THERM2_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thermal Monitor 2 Control</strong></td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>15:0  </td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16  </td>
<td>TM_SELECT (R/W)</td>
<td>Mode of automatic thermal monitor:</td>
<td></td>
</tr>
<tr>
<td> </td>
<td>0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle).</td>
<td></td>
<td></td>
</tr>
<tr>
<td> </td>
<td>1 = Thermal Monitor 2 (thermally-initiated frequency transitions).</td>
<td></td>
<td></td>
</tr>
<tr>
<td> </td>
<td>If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16  </td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1A0H, 416</strong></td>
<td>IA32_MISC_ENABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Enable Misc. Processor Features (R/W)</strong></td>
<td></td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
<td></td>
</tr>
<tr>
<td>0  </td>
<td>Fast-Strings Enable</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>2:1  </td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3  </td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>6:4  </td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7  </td>
<td>Performance Monitoring Available (R)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>8  </td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9  </td>
<td>Hardware Prefetcher Disable (R/W)</td>
<td>When set, disables the hardware prefetcher operation on streams of data.</td>
<td></td>
</tr>
<tr>
<td> </td>
<td>When clear (default), enables the prefetch queue.</td>
<td></td>
<td></td>
</tr>
<tr>
<td> </td>
<td>Disabling of the hardware prefetcher may impact processor performance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10  </td>
<td>FERR# Multiplexing Enable (R/W)</td>
<td>1 = FERR# asserted by the processor to indicate a pending break event within the processor.</td>
<td>Shared</td>
</tr>
<tr>
<td> </td>
<td>0 = Indicates compatible FERR# signaling behavior.</td>
<td></td>
<td></td>
</tr>
<tr>
<td> </td>
<td>This bit must be set to 1 to support XAPIC interrupt model usage.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11  </td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
<td>Shared/ Unique</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------</td>
<td>-----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>12</td>
<td>Processor Event Based Sampling Unavailable (R/O)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>13</td>
<td>TM2 Enable (R/W)</td>
<td>When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.</td>
<td>Shared</td>
</tr>
<tr>
<td>15:14</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Enhanced Intel SpeedStep Technology Enable (R/W)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM (R/W)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>19</td>
<td>Adjacent Cache Line Prefetch Disable (R/W)</td>
<td>When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes). Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.</td>
<td>Shared</td>
</tr>
<tr>
<td>20</td>
<td>Enhanced Intel SpeedStep Technology Select Lock (R/WO)</td>
<td>When set, this bit causes the following bits to become read-only: • Enhanced Intel SpeedStep Technology Select Lock (this bit). • Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.</td>
<td>Shared</td>
</tr>
<tr>
<td>21</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Limit CPUID Maxval (R/W)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>33:24</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>XD Bit Disable (R/W)</td>
<td>When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0). When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages. BIOS must not alter the contents of this bit location if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.</td>
<td>Unique</td>
</tr>
<tr>
<td>36:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>DCU Prefetcher Disable (R/W)</td>
<td>When set to 1, the DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write ‘1’ to disable this feature. The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.</td>
<td>Unique</td>
</tr>
<tr>
<td>38</td>
<td>IDA Disable (R/W)</td>
<td>When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be cleared (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled. Note: The power-on default value is used by BIOS to detect hardware support of IDA. If the power-on default value is 1, IDA is available in the processor. If the power-on default value is 0, IDA is not available.</td>
<td>Shared</td>
</tr>
<tr>
<td>39</td>
<td>IP Prefetcher Disable (R/W)</td>
<td>When set to 1, the IP prefetcher is disabled. The default value after reset is 0. BIOS may write ‘1’ to disable this feature. The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.</td>
<td>Unique</td>
</tr>
<tr>
<td>63:40</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1C9H, 457</strong></td>
<td>MSR_LASTBRANCH_TOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record Stack TOS (R/W)</strong></td>
<td></td>
<td>Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).</td>
<td>Unique</td>
</tr>
<tr>
<td><strong>Register Address: 1D9H, 473</strong></td>
<td>IA32_DEBUGCTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Debug Control (R/W)</strong></td>
<td></td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td><strong>Register Address: 1DDH, 477</strong></td>
<td>MSR_LER_FROM_LIP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Last Exception Record From Linear IP (R/W)</td>
<td>MSR_LER_TO_LIP</td>
<td>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td>Unique</td>
</tr>
<tr>
<td>1DEH, 478</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200H, 512</td>
<td>IA32_MTRR_PHYSBASE0</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>201H, 513</td>
<td>IA32_MTRR_PHYSMASK0</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202H, 514</td>
<td>IA32_MTRR_PHYSBASE1</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203H, 515</td>
<td>IA32_MTRR_PHYSMASK1</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204H, 516</td>
<td>IA32_MTRR_PHYSBASE2</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>205H, 517</td>
<td>IA32_MTRR_PHYSMASK2</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>206H, 518</td>
<td>IA32_MTRR_PHYSBASE3</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>207H, 519</td>
<td>IA32_MTRR_PHYSMASK3</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>208H, 520</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209H, 521</td>
<td>IA32_MTRR_PHYSMASK4</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20AH, 522</td>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20BH, 523</td>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20CH, 524</td>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20DH, 525</td>
<td>IA32_MTRR_PHYSMASK6</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20EH, 526</td>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
</tbody>
</table>

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 20FH, 527</td>
<td>IA32_MTRR_PHYSMASK7</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 250H, 592</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 258H, 600</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 259H, 601</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26AH, 618</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Default Memory Types (R/W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 0 (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 1 (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>308H, 779</td>
<td>IA32_FIXED_CTR2</td>
<td>Fixed-Function Performance Counter Register 2 (R/W)</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td>See Table 2-2. See Section 18.4.1, “IA32_DEBUGCTL MSR.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>345H, 837</td>
<td>MSR_PERF_CAPABILITIES</td>
<td>R/O. This applies to processors that do not support architectural perfmon version 2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>38DH, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td>Fixed-Function-Counter Control Register (R/W)</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>38EH, 910</td>
<td>MSR_PERF_GLOBAL_STATUS</td>
<td>See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>38FH, 911</td>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>38FH, 911</td>
<td>MSR_PERF_GLOBAL_CTRL</td>
<td>See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>390H, 912</td>
<td>IA32_PERF_GLOBAL_OVF_CTRL</td>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>390H, 912</td>
<td>MSR_PERF_GLOBAL_OVF_CTRL</td>
<td>See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>3F1H, 1009</td>
<td>IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)</td>
<td>See Table 2-2. See Section 20.6.2.4, “Processor Event Based Sampling (PEBS).”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td>Enable PEBS on IA32_PMC0. (R/W)</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs.”</td>
<td>Unique</td>
</tr>
<tr>
<td>401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td>See Section 16.3.2.2, “IA32_MCI_STATUS MSRS.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td>See Section 16.3.2.2, “IA32_MCI_STATUS MSRS.”</td>
<td>Unique</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MC1_ADDR</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC1_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MC1_CTL MSRs.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC1_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MC1_STATUS MSRS.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC1_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MC1_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MC2_ADDR</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC2_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MC2_CTL MSRs.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC2_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MC2_STATUS MSRS.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC2_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MC2_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MC3_ADDR</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC3_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MC3_CTL MSRs.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC3_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MC3_STATUS MSRS.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC3_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MC3_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear.

When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”
The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear.
When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MC3_MISC</td>
<td>Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_CTL</td>
<td>Machine Check Error Reporting Register: Controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_STATUS</td>
<td>Machine Check Error Reporting Register: Contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_ADDR</td>
<td>Machine Check Error Reporting Register: Contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_MISC</td>
<td>Machine Check Error Reporting Register: Contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MCI_STATUS</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 16.3.2.2, “IA32_MCI_STATUS MSRs,” and Chapter 24.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_BASIC</td>
<td>Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2. See Appendix A.1, “Basic VMX Information.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_PINBASED_CTLS</td>
<td>Capability Reporting Register of Pin-Based VM-Execution Controls (R/O) See Table 2-2. See Appendix A.3, “VM-Execution Controls.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_PROCBASED_CTLS</td>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O) See Appendix A.3, “VM-Execution Controls.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_EXIT_CTLS</td>
<td>Capability Reporting Register of VM-Exit Controls (R/O) See Table 2-2. See Appendix A.4, “VM-Exit Controls.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_ENTRY_CTLS</td>
<td>Capability Reporting Register of VM-Entry Controls (R/O) See Table 2-2. See Appendix A.5, “VM-Entry Controls.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
<td>See Table 2-2. See Appendix A.6, “Miscellaneous Data.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 486H, 1158</td>
<td>IA32_VMX_CR0_FIXED0</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
<td>See Table 2-2. See Appendix A.7, “VMX-Fixed Bits in CR0.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 487H, 1159</td>
<td>IA32_VMX_CR0_FIXED1</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
<td>See Table 2-2. See Appendix A.7, “VMX-Fixed Bits in CR0.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 488H, 1160</td>
<td>IA32_VMX_CR4_FIXED0</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
<td>See Table 2-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 489H, 1161</td>
<td>IA32_VMX_CR4_FIXED1</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</td>
<td>See Table 2-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 48AH, 1162</td>
<td>IA32_VMX_VMCS_ENUM</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O)</td>
<td>See Table 2-2. See Appendix A.9, “VMCS Enumeration.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 48BH, 1163</td>
<td>IA32_VMX_PROCBASED_CTLS2</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)</td>
<td>See Appendix A.3, “VM-Execution Controls.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 600H, 1536</td>
<td>IA32_DS_AREA</td>
<td></td>
</tr>
<tr>
<td>DS Save Area (R/W)</td>
<td>See Table 2-2. See Section 20.6.3.4, “Debug Store (DS) Mechanism.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107CCH, 67532</td>
<td>MSR_EMON_L3_CTR_CTL0</td>
<td></td>
</tr>
<tr>
<td>GBUSQ Event Control/Counter Register (R/W)</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107CDH, 67533</td>
<td>MSR_EMON_L3_CTR_CTL1</td>
<td></td>
</tr>
<tr>
<td>GBUSQ Event Control/Counter Register (R/W)</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107CEH, 67534</td>
<td>MSR_EMON_L3_CTR_CTL2</td>
<td></td>
</tr>
<tr>
<td>GSNPQ Event Control/Counter Register (R/W)</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107CFH, 67535</td>
<td>MSR_EMON_L3_CTR_CTL3</td>
<td></td>
</tr>
<tr>
<td>GSNPQ Event Control/Counter Register (R/W)</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107DOH, 67536</td>
<td>MSR_EMON_L3_CTR_CTL4</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSB Event Control/Counter Register (R/W)</td>
<td>MSR_EMON_L3_CTR_CTL5</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107D1H, 67537</td>
<td>MSR_EMON_L3_CTR_CTL6</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107D2H, 67538</td>
<td>MSR_EMON_L3_CTR_CTL7</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 107D3H, 67539</td>
<td>MSR_EMON_L3_GL_CTL</td>
<td>Applies to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 18.2.2.</td>
<td>Unique</td>
</tr>
<tr>
<td>L3/FSB Common Control Register (R/W)</td>
<td>IA32_EFER</td>
<td>Extended Feature Enables See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_STAR</td>
<td>System Call Target Address (R/W) See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_LSTAR</td>
<td>IA-32e Mode System Call Target Address (R/W) See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_FMASK</td>
<td>System Call Flag Mask (R/W) See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FS_BASE</td>
<td>Map of BASE Address of FS (R/W) See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_GS_BASE</td>
<td>Map of BASE Address of GS (R/W) See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0101H</td>
<td>IA32_KERNEL_GS_BASE</td>
<td>Swap Target of BASE Address of GS (R/W) See Table 2-2.</td>
<td>Unique</td>
</tr>
</tbody>
</table>
2.3 MSRS IN THE 45 NM AND 32 NM INTEL ATOM® PROCESSOR FAMILY

Table 2-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 2-4. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_1CH, 06_26H, 06_27H, 06_35H, or 06_36H; see Table 2-1.

The column "Shared/Unique" applies to logical processors sharing the same core in processors based on the Intel Atom microarchitecture. "Unique" means each logical processor has a separate MSR, or a bit field in an MSR governs only a logical processor. "Shared" means the MSR or the bit field in an MSR address governs the operation of both logical processors in the same core.

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_P5_MC_ADDR</td>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>IA32_P5_MC_TYPE</td>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td>See Section 9.10.5, “Monitor/Mwait Address Range Determination,” and Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_TIME_STAMP_COUNTER</td>
<td>See Section 18.17, “Time-Stamp Counter,” and see Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_PLATFORM_ID</td>
<td>Platform ID (R) See Table 2-2.</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>MSR_PLATFORM_ID</td>
<td>Model Specific Platform ID (R)</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>IA32_APIC_BASE</td>
<td>See Section 11.4.4, “Local APIC Status and Location,” and Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>MSR_EBL_CR_POWERON</td>
<td>Processor Hard Power-On Configuration (R/W) Enables and disables processor features; (R) indicates current processor configuration.</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Response Error Checking Enable (R/W) 1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>AERR# Drive Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BERR# Enable for initiator bus requests (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BINIT# Driver Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Execute BIST (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>AERR# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BINIT# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 MByte Power on Reset Vector (R/O)</td>
<td>1 = 1 MByte; 0 = 4 GBytes.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:16</td>
<td>APIC Cluster ID (R/O)</td>
<td>1 = Enabled; 0 = Disabled. Always 0.</td>
<td></td>
</tr>
<tr>
<td>19:18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Symmetric Arbitration ID (R/O)</td>
<td>Always 00B.</td>
<td></td>
</tr>
<tr>
<td>26:22</td>
<td>Integer Bus Frequency Ratio (R/O)</td>
<td>Always 00B.</td>
<td></td>
</tr>
</tbody>
</table>

**Register Address: 3AH, 58**

**IA32_FEATURE_CONTROL**

Control Features in Intel 64Processor (R/W)  
See Table 2-2.  
Unique

**Register Address: 40H, 64**

MSR_LASTBRANCH_0_FROM_IP

Last Branch Record 0 From IP (R/W)  
One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also:  
• Last Branch Record Stack TOS at 1C9H.  
• Section 18.5.  
Unique
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 41H, 65</td>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 1 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42H, 66</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 2 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43H, 67</td>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 3 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44H, 68</td>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 4 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 45H, 69</td>
<td>MSR_LASTBRANCH_5_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 5 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 46H, 70</td>
<td>MSR_LASTBRANCH_6_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 6 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 47H, 71</td>
<td>MSR_LASTBRANCH_7_FROM_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 7 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 60H, 96</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 0 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td>One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.</td>
</tr>
<tr>
<td>Register Address: 61H, 97</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 1 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 62H, 98</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 2 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 63H, 99</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 3 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64H, 100</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Last Branch Record 4 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 65H, 101</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td></td>
<td>Unique</td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

**Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 5 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 66H, 102</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 67H, 103</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 79H, 121</td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td></td>
</tr>
<tr>
<td>BIOS Update Trigger Register (W)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 8BH, 139</td>
<td>IA32_BIOS_SIGN_ID</td>
<td></td>
</tr>
<tr>
<td>BIOS Update Signature ID (R/W)</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C1H, 193</td>
<td>IA32_PMC0</td>
<td></td>
</tr>
<tr>
<td>Performance counter register</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C2H, 194</td>
<td>IA32_PMC1</td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: CDH, 205</td>
<td>MSR_FSB_FREQ</td>
<td></td>
</tr>
<tr>
<td>Scaleable Bus Speed (R/O)</td>
<td>This field indicates the intended scalable bus clock speed for processors based on Intel Atom microarchitecture.</td>
<td>Shared</td>
</tr>
</tbody>
</table>
| 2:0 | - 111B: 083 MHz (FSB 333)  
- 101B: 100 MHz (FSB 400)  
- 011B: 133 MHz (FSB 533)  
- 011B: 167 MHz (FSB 667)  
133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.  
166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B. |
| 63:3 | Reserved. |
| Register Address: E7H, 231 | IA32_MPERF |
| Maximum Performance Frequency Clock Count (R/W) | See Table 2-2. | Unique |
| Register Address: E8H, 232 | IA32_APERF |
| Actual Performance Frequency Clock Count (R/W) | See Table 2-2. | Unique |
| Register Address: FEH, 254 | IA32_MTRRCAP |
### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Type Range Register (R)</td>
<td>See Table 2-2.</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 11EH, 281</td>
<td>MSR_BBL_CR_CTL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control Register 3</td>
<td>Used to configure the L2 Cache.</td>
<td></td>
<td>Shared</td>
</tr>
</tbody>
</table>

#### 0

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Hardware Enabled (R/O)</td>
<td>1 = Indicates the L2 is hardware-enabled. 0 = Indicates the L2 is hardware-disabled.</td>
<td></td>
</tr>
<tr>
<td>7:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L2 Enabled (R/W)</td>
<td>1 = L2 cache has been initialized. 0 = Disabled (default).</td>
</tr>
<tr>
<td>Until this bit is set, the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:9</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>L2 Not Present (R/O)</td>
<td>0 = L2 Present. 1 = L2 Not Present.</td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td></td>
</tr>
</tbody>
</table>

#### Global Machine Check Status

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIPV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.</td>
<td></td>
</tr>
<tr>
<td>EIPV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>MOIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 186H, 390</td>
<td>IA32_PERFEVTSEL0</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 187H, 391</td>
<td>IA32_PERFEVTSEL1</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>MSR_PERF_STATUS</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Performance Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Current Performance State Value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44:40</td>
<td>Maximum Bus Ratio (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicates maximum bus ratio configured for the processor.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:45</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19AH, 410</td>
<td>IA32_CLOCK_MODULATION</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Clock Modulation (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19BH, 411</td>
<td>IA32_THERM_INTERRUPT</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Thermal Interrupt Control (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19DH, 413</td>
<td>MSR_THERM2_CTL</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Thermal Monitor 2 Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>TM_SELECT (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode of automatic thermal monitor:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Thermal Monitor 2 (thermally-initiated frequency transitions).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:17</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable Misc. Processor Features (R/W)</td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>0</td>
<td>Fast-Strings Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2. Default value is 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>FERR# Multiplexing Enable (R/W)</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>1 = FERR# asserted by the processor to indicate a pending break event within the processor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Indicates compatible FERR# signaling behavior.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This bit must be set to 1 to support XAPIC interrupt model usage.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Processor Event Based Sampling Unavailable (R/O)</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>TM2 Enable (R/W)</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this bit is cleared (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The processor is operating out of specification if both this bit and the TM1 bit are set to 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

#### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:14</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Enhanced Intel SpeedStep Technology Enable (R/W) See Table 2-2.</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM (R/W) See Table 2-2.</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 20                            | Enhanced Intel SpeedStep Technology Select Lock (R/W0) When set, this bit causes the following bits to become read-only:  
  - Enhanced Intel SpeedStep Technology Select Lock (this bit).  
  - Enhanced Intel SpeedStep Technology Enable bit.  
  The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset. |                                   | Shared         |
| 21                            | Reserved.                            |                                   |                 |
| 22                            | Limit CPUID Maxval (R/W) See Table 2-2. |                                   | Unique         |
| 23                            | xTPR Message Disable (R/W) See Table 2-2. |                                   | Shared         |
| 33:24                         | Reserved.                            |                                   |                 |
| 34                            | XD Bit Disable (R/W) See Table 2-3. |                                   | Unique         |
| 63:35                         | Reserved.                            |                                   |                 |

### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

#### Register Address: 1C9H, 457

MSR_LASTBRANCH_TOS

Last Branch Record Stack TOS (R/W)  
Contains an index (bits 0-2) that points to the MSR containing the most recent branch record.  
See MSR_LASTBRANCH_0_FROM_IP (at 40H).

**Register Address:** 1D9H, 473

**IA32_DEBUGCTL**  
Debug Control (R/W)  
See Table 2-2.

**Register Address:** 1DDH, 477

**MSR_LER_FROM_LIP**  
Last Exception Record From Linear IP (R)  
Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

**Register Address:** 1DEH, 478

**MSR_LER_TO_LIP**  
Last Exception Record To Linear IP (R)  
This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

**Register Address:** 200H, 512

**IA32_MTRR_PHYSBASE0**  
See Table 2-2.

**Register Address:** 201H, 513

**IA32_MTRR_PHYSMASK0**  

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 202H, 514</td>
<td>IA32_MTRR_PHYSBASE1</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 203H, 515</td>
<td>IA32_MTRR_PHYSMASK1</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 204H, 516</td>
<td>IA32_MTRR_PHYSBASE2</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 205H, 517</td>
<td>IA32_MTRR_PHYSMASK2</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 206H, 518</td>
<td>IA32_MTRR_PHYSBASE3</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 207H, 519</td>
<td>IA32_MTRR_PHYSMASK3</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 208H, 520</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 209H, 521</td>
<td>IA32_MTRR_PHYSMASK4</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 20AH, 522</td>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 20BH, 523</td>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 20CH, 524</td>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 20DH, 525</td>
<td>IA32_MTRR_PHYSMASK6</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 20EH, 526</td>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 20FH, 527</td>
<td>IA32_MTRR_PHYSMASK7</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 250H, 592</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 258H, 600</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 259H, 601</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
</tbody>
</table>
### Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 269H, 617</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26AH, 618</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td>Fixed-Function Performance Counter Register 0 (R/W)</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td>Fixed-Function Performance Counter Register 1 (R/W)</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30BH, 779</td>
<td>IA32_FIXED_CTR2</td>
<td>Fixed-Function Performance Counter Register 2 (R/W)</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td>See Table 2-2. See Section 18.4.1, “IA32_DEBUGCTL MSR.”</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38DH, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td>Fixed-Function-Counter Control Register (R/W)</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38FH, 911</td>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 390H, 912</td>
<td>IA32_PERF_GLOBAL_OVF_CTRL</td>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Shared/Unique</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 20.6.2.4, &quot;Processor Event Based Sampling (PEBS).&quot;</td>
<td></td>
<td></td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Enable PEBS on IA32_PMC0 (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC1_CTL</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC1_STATUS</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MCO_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40DH, 1037</td>
<td>IA32_MC3_STATUS</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MC3_ADDR</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
<td>IA32_MC4_CTL</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 411H, 1041</td>
<td>IA32_MC4_STATUS</td>
<td></td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 412H, 1042</td>
<td>IA32_MC4_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear.
When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Register Address: 480H, 1152  
Reporting Register of Basic VMX Capabilities (R/O)  
See Table 2-2. See Appendix A.1, “Basic VMX Information.”

Register Address: 481H, 1153  
Capability Reporting Register of Pin-Based VM-Execution Controls (R/O)  
See Table 2-2. See Appendix A.3, “VM-Execution Controls.”

Register Address: 482H, 1154  
Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)  
See Appendix A.3, “VM-Execution Controls.”

Register Address: 483H, 1155  
Capability Reporting Register of VM-Exit Controls (R/O)  
See Table 2-2. See Appendix A.4, “VM-Exit Controls.”

Register Address: 484H, 1156  
Capability Reporting Register of VM-Entry Controls (R/O)  
See Table 2-2. See Appendix A.5, “VM-Entry Controls.”

Register Address: 485H, 1157  
Reporting Register of Miscellaneous VMX Capabilities (R/O)  
See Table 2-2. See Appendix A.6, “Miscellaneous Data.”

Register Address: 486H, 1158  
Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)  
See Table 2-2. See Appendix A.7, “VMX-Fixed Bits in CR0.”

Register Address: 487H, 1159  
Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)  
See Table 2-2. See Appendix A.7, “VMX-Fixed Bits in CR0.”

Register Address: 488H, 1160  
Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)  
See Table 2-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”

Register Address: 489H, 1161  
Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)  
See Table 2-2. See Appendix A.8, “VMX-Fixed Bits in CR4.”

Register Address: 48AH, 1162  
Capability Reporting Register of VMCS Field Enumeration (R/O)  
See Table 2-2. See Appendix A.9, “VMCS Enumeration.”

Register Address: 48BH, 1163  
IA32_VMX_PROCBASED_CTLS2

Table 2-4. MSRs in the 45 nm and 32 nm Intel Atom® Processor Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_VMX_BASIC</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_PINBASED_CTLS</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_PROCBASED_CTLS</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_EXIT_CTLS</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_ENTRY_CTLS</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_MISC</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_CRO_FIXED0</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_CRO_FIXED1</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_CR4_FIXED0</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_VMX_CR4_FIXED1</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
</tbody>
</table>
Table 2-5 lists model-specific registers (MSRs) that are specific to Intel Atom® processor with a CPUID Signature DisplayFamily_DisplayModel value of 06_27H.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)</strong>&lt;br&gt;See Appendix A.3, “VM-Execution Controls.”</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 600H, 1536</td>
<td>IA32_DS_AREA</td>
<td>DS Save Area (R/W)&lt;br&gt;See Table 2-2. See Section 20.6.3.4, “Debug Store (DS) Mechanism.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_EFER</td>
<td>Extended Feature Enables&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>System Call Target Address (R/W)&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_STAR</td>
<td>IA32_STAR</td>
<td>Unique</td>
</tr>
<tr>
<td>IA-32e Mode System Call Target Address (R/w)&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_LSTAR</td>
<td>IA32_LSTAR</td>
<td>Unique</td>
</tr>
<tr>
<td>System Call Flag Mask (R/w)&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FMASK</td>
<td>IA32_FMASK</td>
<td>Unique</td>
</tr>
<tr>
<td>Map of BASE Address of FS (R/w)&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_FS_BASE</td>
<td>IA32_FS_BASE</td>
<td>Unique</td>
</tr>
<tr>
<td>Map of BASE Address of GS (R/w)&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0101H</td>
<td>IA32_GS_BASE</td>
<td>IA32_GS_BASE</td>
<td>Unique</td>
</tr>
<tr>
<td>Swap Target of BASE Address of GS (R/w)&lt;br&gt;See Table 2-2.</td>
<td>Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0102H</td>
<td>IA32_KERNEL_GS_BASE</td>
<td>IA32_KERNEL_GS_BASE</td>
<td>Unique</td>
</tr>
</tbody>
</table>

Table 2-5 lists model-specific registers (MSRs) that are specific to Intel Atom® processor with a CPUID Signature DisplayFamily_DisplayModel value of 06_27H.

Table 2-5. MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_27H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
<td><strong>Scope</strong></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F8H, 1016</td>
<td>MSR_PKG_C2_RESIDENCY</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Package C2 Residency&lt;br&gt;Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>Package C2 Residency Counter (R/O)</td>
<td>Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3F9H, 1017</td>
<td>MSR_PKG_C4_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C4 Residency Counter. (R/O)</td>
<td>Time that this package is in processor-specific C4 states since last reset. Counts at 1 Mhz frequency.</td>
<td>Package</td>
</tr>
<tr>
<td>Package C4 Residency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FAH, 1018</td>
<td>MSR_PKG_C6_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C6 Residency Counter. (R/O)</td>
<td>Time that this package is in processor-specific C6 states since last reset. Counts at 1 Mhz frequency.</td>
<td>Package</td>
</tr>
</tbody>
</table>

2.4 MSRS IN INTEL PROCESSORS BASED ON SILVERMONT MICROARCHITECTURE

Table 2-6 lists model-specific registers (MSRs) common to Intel processors based on the Silvermont microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_37H, 06_4AH, 06_4DH, 06_5AH, or 06_5DH; see Table 2-1. The MSRs listed in Table 2-6 are also common to processors based on the Airmont microarchitecture and newer microarchitectures for next generation Intel Atom processors.

Table 2-7 lists MSRs common to processors based on the Silvermont and Airmont microarchitectures, but not newer microarchitectures.

Table 2-8, Table 2-9, and Table 2-10 lists MSRs that are model-specific across processors based on the Silvermont microarchitecture.

In the Silvermont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Silvermont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. "Package" means all processor cores in the physical package share the same MSR or bit interface.
### Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 9.10.5, “Monitor/Mwait Address Range Determination,” and Table 2-2.</td>
<td></td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 18.17, “Time-Stamp Counter,” and Table 2-2.</td>
<td></td>
<td>IA32_APIC_BASE</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 11.4.4, “Local APIC Status and Location,” and Table 2-2.</td>
<td></td>
<td>MSR_EBL_CR_POWERON</td>
<td>Processor Hard Power-On Configuration (R/W)</td>
<td>Module</td>
</tr>
<tr>
<td>Writes ignored.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Intel® Atom® Processors.”</td>
<td></td>
<td>MSR_SMI_COUNT</td>
<td>SMI Counter (R/O)</td>
<td>Core</td>
</tr>
<tr>
<td>31:0</td>
<td>SMI Count (R/O)</td>
<td>Running count of SMI events since last RESET.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td>BIOS Update Trigger Register (w)</td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_BIOS_SIGN_ID</td>
<td>BIOS Update Signature ID (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_PMC0</td>
<td>Performance Counter Register</td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_PMC1</td>
<td>Performance Counter Register</td>
<td>Core</td>
</tr>
<tr>
<td>See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td>MSR_PMG_IO_CAPTURE_BASE</td>
<td>Power Management IO Redirection in C-state (R/w)</td>
<td>Module</td>
</tr>
<tr>
<td>15:0</td>
<td>LVL_2 Base Address (R/W)</td>
<td>Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-6.  MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures)  (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>18:16</td>
<td>C-state Range (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100b - C4 is the max C-State to include</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>110b - C6 is the max C-State to include</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>111b - C7 is the max C-State to include</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7H, 231</td>
<td>IA32_MPERF</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Maximum Performance Frequency Clock Count (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E8H, 232</td>
<td>IA32_APERF</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Actual Performance Frequency Clock Count (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td>IA32_MTRRCAP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Memory Type Range Register (R)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 13CH, 316</td>
<td>MSR_FEATURE_CONFIG</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>AES Configuration (RW-L)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>AES Configuration (RW-L)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Upon a successful read of this MSR, the configuration of AES instruction sets availability is as follows:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Global Machine Check Status</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RI PV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>EI PV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MCIP</td>
<td>When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.</td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 186H, 390
IA32_PERFEVTSEL0

See Table 2-2.

| 7:0                           | Event Select                         | Core            |       |
| 15:8                          | UM ask                               |                 |       |
| 16                            | USR                                  |                 |       |
| 17                            | OS                                   |                 |       |
| 18                            | Edge                                 |                 |       |
| 19                            | PC                                   |                 |       |
| 20                            | INT                                  |                 |       |
| 21                            | Reserved                             |                 |       |
| 22                            | EN                                   |                 |       |
| 23                            | INV                                  |                 |       |
| 31:24                         | CMASK                                |                 |       |
| 63:32                         | Reserved                             |                 |       |

Register Address: 187H, 391
IA32_PERFEVTSEL1

See Table 2-2.

Register Address: 198H, 408
IA32_PERF_STATUS

See Table 2-2.

Register Address: 199H, 409
IA32_PERF_CTL

See Table 2-2.

Register Address: 19AH, 410
IA32_CLOCK_MODULATION

Clock Modulation (R/W)
See Table 2-2.
IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

Register Address: 19BH, 411
IA32_THERM_INTERRUPT

IA32_PERFEVTSEL0, IA32_PERFEVTSEL1, IA32_PERF_STATUS, IA32_PERF_CTL, IA32_CLOCK_MODULATION, IA32_THERM_INTERRUPT
Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td>Thermal Interrupt Control (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>1A2H, 418</td>
<td>MSR_TEMPERATURE_TARGET</td>
<td>Thermal Monitor Status (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>1A6H, 422</td>
<td>MSR_OFFCORE_RSP_0</td>
<td>Offcore Response Event Select Register (R/W)</td>
<td>See Table 2-2.</td>
<td>Module</td>
</tr>
<tr>
<td>1A7H, 423</td>
<td>MSR_OFFCORE_RSP_1</td>
<td>Offcore Response Event Select Register (R/W)</td>
<td>See Table 2-2.</td>
<td>Module</td>
</tr>
<tr>
<td>1B0H, 432</td>
<td>IA32_ENERGY_PERF_BIAS</td>
<td>Debug Control (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>1D9H, 473</td>
<td>IA32_DEBUGCTL</td>
<td>Last Exception Record From Linear IP (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>1DEH, 478</td>
<td>MSR_LER_FROM_LIP</td>
<td>Last Exception Record To Linear IP (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
</tbody>
</table>

**Temperature Target**

15:0: Reserved.

23:16: Temperature Target (R)

The default thermal throttling or PROCHOT# activation temperature in degrees C. The effective temperature for thermal throttling or PROCHOT# activation is “Temperature Target” + “Target Offset”.

29:24: Target Offset (R/W)

Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).

63:30: Reserved.
### Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>201H, 513</td>
<td>IA32_MTRR_PHYSMASK0</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>202H, 514</td>
<td>IA32_MTRR_PHYSBASE1</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>203H, 515</td>
<td>IA32_MTRR_PHYSMASK1</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>204H, 516</td>
<td>IA32_MTRR_PHYSBASE2</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>205H, 517</td>
<td>IA32_MTRR_PHYSMASK2</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>206H, 518</td>
<td>IA32_MTRR_PHYSBASE3</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>207H, 519</td>
<td>IA32_MTRR_PHYSMASK3</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>208H, 520</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>209H, 521</td>
<td>IA32_MTRR_PHYSMASK4</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>20AH, 522</td>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>20BH, 523</td>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>20CH, 524</td>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>20DH, 525</td>
<td>IA32_MTRR_PHYSMASK6</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>20EH, 526</td>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>20FH, 527</td>
<td>IA32_MTRR_PHYSMASK7</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>250H, 592</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>258H, 600</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>259H, 601</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>268H, 616</td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26AH, 618</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default Memory Types (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 0 (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 1 (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30BH, 779</td>
<td>IA32_FIXED_CTR2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 2 (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 18.4.1, “IA32_DEBUGCTL MSR.”</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 380H, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-Function-Counter Control Register (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38FH, 911</td>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td></td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FDH, 1021</td>
<td>MSR_CORE_C6_RESIDENCY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MC1_CTL</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MC1_STATUS</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_STATUS</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.” The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40H, 1037</td>
<td>IA32_MC3_STATUS</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MC3_ADDR</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.” The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
<td>IA32_MC4_CTL</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 411H, 1041</td>
<td>IA32_MC4_STATUS</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)
Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 412H, 1042</td>
<td>IA32_MC4_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td>The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 416H, 1046</td>
<td>IA32_MC5_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 480H, 1152</td>
<td>IA32_VMX_BASIC</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Reporting Register of Basic VMX Capabilities (R/O)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Appendix A.1, “Basic VMX Information.”</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 481H, 1153</td>
<td>IA32_VMX_PINBASED_CTLS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Capability Reporting Register of Pin-Based VM-Execution Controls (R/O)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Appendix A.3, “VM-Execution Controls.”</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 482H, 1154</td>
<td>IA32_VMX_PROCBASED_CTLS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.3, “VM-Execution Controls.”</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 483H, 1155</td>
<td>IA32_VMX_EXIT_CTLS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Exit Controls (R/O)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Appendix A.4, “VM-Exit Controls.”</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 484H, 1156</td>
<td>IA32_VMX_ENTRY_CTLS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Entry Controls (R/O)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Appendix A.5, “VM-Entry Controls.”</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 485H, 1157</td>
<td>IA32_VMX_MISC</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Appendix A.6, “Miscellaneous Data.”</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 486H, 1158</td>
<td>IA32_VMX_CR0_FIXED0</td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>
### Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>See Appendix A.7, “VMX-Fixed Bits in CR0.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 487H, 1159</td>
<td>IA32_VMX_CRO_FIXED1</td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>See Appendix A.7, “VMX-Fixed Bits in CR0.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 488H, 1160</td>
<td>IA32_VMX_CR4_FIXED0</td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>See Appendix A.8, “VMX-Fixed Bits in CR4.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 489H, 1161</td>
<td>IA32_VMX_CR4_FIXED1</td>
</tr>
<tr>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>See Appendix A.9, “VMCS Enumeration.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48AH, 1162</td>
<td>IA32_VMX_VMCS_ENUM</td>
</tr>
<tr>
<td>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Appendix A.3, “VM-Execution Controls.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48BH, 1163</td>
<td>IA32_VMX_PROCBASED_CTLS2</td>
</tr>
<tr>
<td>Capability Reporting Register of EPT and VPID (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48CH, 1164</td>
<td>IA32_VMX_EPT_VPID_ENUM</td>
</tr>
<tr>
<td>Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48DH, 1165</td>
<td>IA32_VMX_TRUE_PINBASED_CTLS</td>
</tr>
<tr>
<td>Capability Reporting Register of Primary Processor-based VM-Execution Flex Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48EH, 1166</td>
<td>IA32_VMX_TRUE_PROCBASED_CTLS</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Exit Flex Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48FH, 1167</td>
<td>IA32_VMX_TRUE_EXIT_CTLS</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Entry Flex Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 490H, 1168</td>
<td>IA32_VMX_TRUE_ENTRY_CTLS</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Function Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>Register Address: 4C1H, 1217</td>
<td>IA32_A_PMC0</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C2H, 1218</td>
<td>IA32_A_PMC1</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 600H, 1536</td>
<td>IA32_DS_AREA</td>
</tr>
<tr>
<td>See Table 2-2 and Section 20.6.3.4, &quot;Debug Store (DS) Mechanism.&quot;</td>
<td></td>
</tr>
<tr>
<td>Register Address: 660H, 1632</td>
<td>MSR_CORE_C1_RESIDENCY</td>
</tr>
<tr>
<td>Register Address: 6E0H, 1760</td>
<td>IA32_TSC_DEADLINE</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_EFER</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_STAR</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_LSTAR</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FMASK</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_FS_BASE</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0101H</td>
<td>IA32_GS_BASE</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0102H</td>
<td>IA32_KERNEL_GS_BASE</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0103H</td>
<td>IA32_TSC_AUX</td>
</tr>
</tbody>
</table>

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)
Table 2-7 lists model-specific registers (MSRs) that are common to Intel Atom® processors based on the Silvermont and Airmont microarchitectures but not newer microarchitectures.

### Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Specific Platform ID (R)</td>
<td>MSR_PLATFORM_ID</td>
<td>Model Specific Platform ID (R)</td>
<td>Module</td>
</tr>
<tr>
<td>7:0 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:8 Maximum Qualified Ratio (R)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The maximum allowed bus ratio.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49:13 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52:50 See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:33 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Control Features in Intel 64Processor (R/W)</td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>0 Lock (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Enable VMX outside SMX operation (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Section 18.5 and record format in Section 18.4.8.1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Last Branch Record 0 From IP (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 1 From IP (R/W)</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 2 From IP (R/W)</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 3 From IP (R/W)</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 4 From IP (R/W)</td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
</tr>
<tr>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-6. MSRs Common to Intel Atom® Processors (Silvermont and Newer Microarchitectures) (Contd.)
### Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>45H, 69</td>
<td>MSR_LASTBRANCH_5_FROM_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>46H, 70</td>
<td>MSR_LASTBRANCH_6_FROM_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>47H, 71</td>
<td>MSR_LASTBRANCH_7_FROM_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>60H, 96</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>61H, 97</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>62H, 98</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>63H, 99</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>64H, 100</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>65H, 101</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>66H, 102</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>67H, 103</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>CEH, 206</td>
<td>MSR_PLATFORM_INFO</td>
<td>Platform Information: Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
<td>This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * Scalable Bus Frequency.</td>
<td>Package</td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>C-State Configuration Control (R/W)</strong></td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td>Module</td>
</tr>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-state support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only)</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td>When set, will map IO_read instructions sent to IO register specified by MSR_PKG_IO_CAPTURE_BASE to MWAIT instructions.</td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/W0)</td>
<td>When set, locks bits 15:0 of this register until next reset.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 11EH, 281</td>
<td>MSR_BBL_CR_CTL3</td>
<td>Control Register 3 Used to configure the L2 Cache.</td>
<td>Module</td>
</tr>
<tr>
<td>0</td>
<td>L2 Hardware Enabled (R/O)</td>
<td>1 = If the L2 is hardware-enabled. 0 = Indicates if the L2 is hardware-disabled.</td>
<td></td>
</tr>
<tr>
<td>7:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L2 Enabled (R/W)</td>
<td>1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.</td>
<td></td>
</tr>
<tr>
<td>22:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>L2 Not Present (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = L2 Present.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = L2 Not Present.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable Misc. Processor Features (R/W)</td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Fast-Strings Enable</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2. Default value is 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Processor Event Based Sampling Unavailable (R/O)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Enhanced Intel SpeedStep Technology Enable (R/W)</td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM (R/W)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Limit CPUID Maxval (R/W)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>XD Bit Disable (R/W)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>See Table 2-3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Turbo Mode Disable (R/W)

When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be cleared (CPUID.06H: EAX[1]=0).
When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor’s support of turbo mode is enabled.
Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.

### Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>Turbo Mode Disable (R/W)</td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td>63:39</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 1C8H, 456 MSR_LBR_SELECT

Last Branch Record Filtering Select Register (R/W)

See Section 18.9.2, “Filtering of Last Branch Records.”

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPL_EQ_0</td>
</tr>
<tr>
<td>1</td>
<td>CPL_NEQ_0</td>
</tr>
<tr>
<td>2</td>
<td>JCC</td>
</tr>
<tr>
<td>3</td>
<td>NEAR_REL_CALL</td>
</tr>
<tr>
<td>4</td>
<td>NEAR_IND_CALL</td>
</tr>
<tr>
<td>5</td>
<td>NEAR_RET</td>
</tr>
<tr>
<td>6</td>
<td>NEAR_IND_JMP</td>
</tr>
<tr>
<td>7</td>
<td>NEAR_REL_JMP</td>
</tr>
<tr>
<td>8</td>
<td>FAR_BRANCH</td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 1C9H, 457 MSR_LASTBRANCH_TOS

Last Branch Record Stack TOS (R/W)

Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.

Register Address: 38EH, 910 IA32_PERF_GLOBAL_STATUS

See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”

Register Address: 390H, 912 IA32_PERF_GLOBAL_OVF_CTRL

See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”

Register Address: 3F1H, 1009 IA32_PEMS_ENABLE (MSR_PEMS_ENABLE)

See Table 2-2. See Section 20.6.2.4, “Processor Event Based Sampling (PEBS).”

Register Address: 3FAH, 1018 MSR_PKG_C6_RESIDENCY

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Register Address: 3FAH, 1018 MSR_PKG_C6_RESIDENCY

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

0     | Enable PEBS for precise event on IA32_PMC0 (R/W) |

Register Address: 3FAH, 1018 MSR_PKG_C6_RESIDENCY

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

63:0  | Package C6 Residency Counter (R/O) |
Value since last reset that this package is in processor-specific C6 states. Counts at the TSC Frequency.
Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>Register Address: 664H, 1636</td>
<td>MSR_MC6_RESIDENCY_COUNTER</td>
</tr>
<tr>
<td>Module C6 Residency Counter (R/0)</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
</tr>
<tr>
<td>63:0</td>
<td>Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.</td>
</tr>
</tbody>
</table>

2.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 2-8 lists MSRs that are specific to the Intel Atom® processor E3000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_37H) and Intel Atom processors (CPUID Signature DisplayFamily_DisplayModel value of 06_4AH, 06_5AH, or 06_5DH).

Table 2-8. Specific MSRs Supported by Intel Atom® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_37H, 06_4AH, 06_5AH, or 06_5DH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>Register Address: CDH, 205</td>
<td>MSR_FSB_FREQ</td>
</tr>
<tr>
<td>Scaleable Bus Speed (R/O)</td>
<td>This field indicates the intended scalable bus clock speed for processors based on Silvermont microarchitecture.</td>
</tr>
</tbody>
</table>
| 2:0 | • 100B: 080.0 MHz  
     • 000B: 083.3 MHz  
     • 001B: 100.0 MHz  
     • 010B: 133.3 MHz  
     • 011B: 116.7 MHz |
| 63:3 | Reserved. |
| Register Address: 606H, 1542 | MSR_RAPL_POWER_UNIT |
| Unit Multipliers used in RAPL Interfaces (R/O) | See Section 15.10.1, “RAPL Interfaces.” |
| 3:0 | Power Units  
     Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment. |
| 7:4 | Reserved. |
| 12:8 | Energy Status Units  
     Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment. |
| 15:13 | Reserved. |
| 19:16 | Time Unit  
     The value is 0000b, indicating time unit is in one second. |
| 63:20 | Reserved. |
| Register Address: 610H, 1552 | MSR_PKG_POWER_LIMIT |
| PKG RAPL Power Limit Control (R/W) |

2-110 Vol. 4
Table 2-9 lists model-specific registers (MSRs) that are specific to the Intel Atom® processor E3000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_37H).

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:0</td>
<td>Package Power Limit #1 (R/W)</td>
<td>See Section 15.10.3, “Package RAPL Domain,” and MSR_RAPL_POWER_UNIT in Table 2-8.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Enable Power Limit #1 (R/W)</td>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Package Clamping Limitation #1 (R/W)</td>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td></td>
</tr>
<tr>
<td>23:17</td>
<td>Time Window for Power Limit #1 (R/W)</td>
<td>In unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.</td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>611H, 1553</td>
<td>MSR_PKG_ENERGY_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKG Energy Status (R/O)</td>
<td>Package</td>
<td>See Section 15.10.3, “Package RAPL Domain,” and MSR_RAPL_POWER_UNIT in Table 2-8.</td>
<td></td>
</tr>
<tr>
<td>639H, 1593</td>
<td>MSR_PP0_ENERGY_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PP0 Energy Status (R/O)</td>
<td>Package</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains,” and MSR_RAPL_POWER_UNIT in Table 2-8.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-9 lists model-specific registers (MSRs) that are specific to the Intel Atom® processor E3000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_37H).

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>668H, 1640</td>
<td>MSR_CC6_DEMOTION_POLICY_CONFIG</td>
<td>Core C6 Demotion Policy Config MSR</td>
<td>Package</td>
</tr>
<tr>
<td>63</td>
<td>Controls per-core C6 demotion policy. Writing a value of 0 disables core level HW demotion policy.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>669H, 1641</td>
<td>MSR_MC6_DEMOTION_POLICY_CONFIG</td>
<td>Module C6 Demotion Policy Config MSR</td>
<td>Package</td>
</tr>
<tr>
<td>63</td>
<td>Controls module (i.e., two cores sharing the second-level cache) C6 demotion policy. Writing a value of 0 disables module level HW demotion policy.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>664H, 1636</td>
<td>MSR_MC6_RESIDENCY_COUNTER</td>
<td>Module C6 Residency Counter (R/O)</td>
<td>Module</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-10 lists model-specific registers (MSRs) that are specific to Intel Atom® processor C2000 Series (CPUID Signature DisplayFamily_DisplayModel value of 06_4DH).

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 1A4H, 420</td>
<td>MSR_MISC_FEATURE_CONTROL</td>
<td>0</td>
<td>L2 Hardware Prefetcher Disable (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DCU Hardware Prefetcher Disable (R/W)</td>
<td></td>
<td>L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</td>
<td>Core</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td>Maximum Ratio Limit of Turbo Mode (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C</td>
<td></td>
<td>Maximum turbo ratio limit of 1 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 2C</td>
<td></td>
<td>Maximum turbo ratio limit of 2 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 3C</td>
<td></td>
<td>Maximum turbo ratio limit of 3 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 4C</td>
<td></td>
<td>Maximum turbo ratio limit of 4 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 5C</td>
<td></td>
<td>Maximum turbo ratio limit of 5 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for 6C</td>
<td></td>
<td>Maximum turbo ratio limit of 6 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for 7C</td>
<td></td>
<td>Maximum turbo ratio limit of 7 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>63:56</td>
<td>Maximum Ratio Limit for 8C</td>
<td></td>
<td>Maximum turbo ratio limit of 8 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 606H, 1542</td>
<td>MSR_RAPL_POWER_UNIT</td>
<td>Unit Multipliers used in RAPL Interfaces (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.10.1, “RAPL Interfaces.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Power Units</td>
<td></td>
<td>Power related information (in milliWatts) is based on the multiplier, $2^{PU}$, where $PU$ is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.</td>
<td>Package</td>
</tr>
<tr>
<td>7:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.4.2  MSRs in Intel Atom® Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 2-6, Table 2-7, Table 2-8, and Table 2-11. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_4CH; see Table 2-1.

### Table 2-11. MSRs in Intel Atom® Processors Based on Airmont Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: CDH, 205</td>
<td>MSR_FSB_FREQ</td>
<td>Scaleable Bus Speed (R/O)</td>
<td>Module</td>
</tr>
<tr>
<td>This field indicates the intended scalable bus clock speed for processors based on Airmont microarchitecture.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>• 0000B: 083.3 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0001B: 100.0 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0010B: 133.3 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0011B: 116.7 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0100B: 080.0 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0101B: 093.3 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0110B: 090.0 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 0111B: 088.9 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 1000B: 087.5 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:5</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-11. MSRs in Intel Atom® Processors Based on Airmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C-State Configuration Control (R/W)</strong></td>
<td></td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td>When set, will map IO_read instructions sent to IO register specified by MSR_PKG_IO_CAPTURE_BASE to MWAIT instructions.</td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/W)</td>
<td>When set, locks bits 15:0 of this register until next reset.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: E4H, 228 MSR_PKG_IO_CAPTURE_BASE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>LVL_2 Base Address (R/W)</td>
<td>Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.</td>
<td></td>
</tr>
<tr>
<td>18:16</td>
<td>C-state Range (R/W)</td>
<td>Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL(bit10): 000b - C3 is the max C-State to include. 001b - Deep Power Down Technology is the max C-State. 010b - C7 is the max C-State to include.</td>
<td></td>
</tr>
<tr>
<td>63:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 638H, 1592 MSR_PP0POWER_LIMIT</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PP0 RAPL Power Limit Control (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
Intel Atom processors based on the Goldmont microarchitecture support MSRs listed in Table 2-6 and Table 2-12. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_5CH; see Table 2-1.

In the Goldmont microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Goldmont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. “Package” means all processor cores in the physical package share the same MSR or bit interface.

### Table 2-11. MSRs in Intel Atom® Processors Based on Airmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:0</td>
<td>PP0 Power Limit #1 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains,” and MSR_RAPL_POWER_UNIT in Table 2-8.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Enable Power Limit #1 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:17</td>
<td>Time Window for Power Limit #1 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies the time duration over which the average power must remain below PP0_POWER_LIMIT #1(14:0). Supported Encodings:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0: 1 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x1: 5 second time duration (Default).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x2: 10 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3: 15 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x4: 20 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x5: 25 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x6: 30 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x7: 35 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8: 40 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x9: 45 second time duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xB-0x7F - reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 17H, 23</td>
<td>MSR_PLATFORM_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Specific Platform ID (R)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52:50</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:33</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td>3AH, 58</td>
<td>Control Features in Intel 64 Processor (R/W) See Table 2-2.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Lock (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable VMX inside SMX operation (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable VMX outside SMX operation (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>SENTER local functions enables (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SENTER global functions enable (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>SGX global functions enable (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_TSC_ADJUST</td>
<td>3BH, 59</td>
<td>Per-Core TSC ADJUST (R/w) See Table 2-2.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>IA32_PMC2</td>
<td>C3H, 195</td>
<td>Performance Counter Register See Table 2-2.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>IA32_PMC3</td>
<td>C4H, 196</td>
<td>Performance Counter Register See Table 2-2.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>MSR_PLATFORM_INFO</td>
<td>CEH, 206</td>
<td>Platform Information Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O) This is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * 100 MHz.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O) When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O) When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Programmable TJ OFFSET (R/O) When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify a temperature offset.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:31</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.</td>
<td>Package</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C-State Configuration Control (R/W)</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td>Core</td>
</tr>
<tr>
<td>3:0</td>
<td>Package C-State Limit (R/w)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/w)</td>
<td>When set, will map IO_read instructions sent to IO register specified by MSR_PKG_IO_CAPTURE_BASE to MWAIT instructions.</td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/WO)</td>
<td>When set, locks bits 15:0 of this register until next reset.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17DH, 381</td>
<td>MSR_SMM_MCA_CAP</td>
<td>Enhanced SMM Capabilities (SMM-RO) Reports SMM capability enhancement. Accessible only while in SMM.</td>
<td>Core</td>
</tr>
<tr>
<td>57:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>SMM_Code_Access_Chk (SMM-RO)</td>
<td>If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>Long_Flow_Indication (SMM-RO)</td>
<td>If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.</td>
<td></td>
</tr>
<tr>
<td>63:60</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 188H, 392</td>
<td>IA32_PERFEVTSEL2</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 189H, 393</td>
<td>IA32_PERFEVTSEL3</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
<td>Enable Misc. Processor Features (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Fast-Strings Enable</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>2:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td>See Table 2-2. Default value is 1.</td>
<td>Package</td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>10:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>12</td>
<td>Processor Event Based Sampling Unavailable (R/O)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Enhanced Intel SpeedStep Technology Enable (R/W)</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>21:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Limit CPUID Maxval (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>33:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>XD Bit Disable (R/W)</td>
<td>See Table 2-3.</td>
<td>Core</td>
</tr>
<tr>
<td>37:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>Turbo Mode Disable (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor’s support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:39</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A4H, 420</td>
<td>MSR_MISC_FEATURE_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous Feature Control (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>L2 Hardware Prefetcher Disable (R/W)</td>
<td>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</td>
<td>Core</td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DCU Hardware Prefetcher Disable (R/W)</td>
<td>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</td>
<td>Core</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1AAH, 426</td>
<td>MSR_MISC_PWR_MGMT</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0</td>
<td>EIST Hardware Coordination Disable (R/W)</td>
<td>When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores. When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.</td>
<td>Package</td>
</tr>
<tr>
<td>21:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Thermal Interrupt Coordination Enable (R/w)</td>
<td>If set, then thermal interrupt on one core is routed to all cores.</td>
<td></td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode by Core Groups (R/W)</td>
<td>Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically. For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for Active Cores in Group 0</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 0 threshold.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for Active Cores in Group 1</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 1 threshold, and greater than the Group 0 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for Active Cores in Group 2</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 2 threshold, and greater than the Group 1 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for Active Cores in Group 3</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 3 threshold, and greater than the Group 2 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for Active Cores in Group 4</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 4 threshold, and greater than the Group 3 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for Active Cores in Group 5</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 5 threshold, and greater than the Group 4 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for Active Cores in Group 6</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 6 threshold, and greater than the Group 5 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td>63:56</td>
<td>Maximum Ratio Limit for Active Cores in Group 7</td>
<td>Maximum turbo ratio limit when the number of active cores is less than or equal to the Group 7 threshold, and greater than the Group 6 threshold.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Register Address: 1AEH, 430</strong></td>
<td>MSR_TURBO_GROUP_CORECNT</td>
<td>Group Size of Active Cores for Turbo Mode Operation (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Writes of 0 threshold is ignored.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Group 0 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 0 Max Turbo Ratio limit.</td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Group 1 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 1 Max Turbo Ratio limit. Must be greater than the Group 0 Core Count.</td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Group 2 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 2 Max Turbo Ratio limit. Must be greater than the Group 1 Core Count.</td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Group 3 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 3 Max Turbo Ratio limit. Must be greater than the Group 2 Core Count.</td>
<td>Package</td>
</tr>
<tr>
<td>39:32</td>
<td>Group 4 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 4 Max Turbo Ratio limit. Must be greater than the Group 3 Core Count.</td>
<td>Package</td>
</tr>
<tr>
<td>47:40</td>
<td>Group 5 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 5 Max Turbo Ratio limit. Must be greater than the Group 4 Core Count.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>55:48</td>
<td>Group 6 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 6 Max Turbo Ratio limit. Must be greater than the Group 5 Core Count.</td>
<td>Package</td>
</tr>
<tr>
<td>63:56</td>
<td>Group 7 Core Count Threshold</td>
<td>Maximum number of active cores to operate under the Group 7 Max Turbo Ratio limit. Must be greater than the Group 6 Core Count, and not less than the total number of processor cores in the package. E.g., specify 255.</td>
<td>Package</td>
</tr>
</tbody>
</table>
| Register Address: 1C8H, 456    | MSR_LBR_SELECT                        | Last Branch Record Filtering Select Register (R/W)  
See Section 18.9.2, “Filtering of Last Branch Records.” | Core     |
|                               |                                       | 0 CPL_EQ_0     |         |
|                               |                                       | 1 CPL_NEQ_0    |         |
|                               |                                       | 2 JCC          |         |
|                               |                                       | 3 NEAR_REL_CALL|         |
|                               |                                       | 4 NEAR_IND_CALL|         |
|                               |                                       | 5 NEAR_RET     |         |
|                               |                                       | 6 NEAR_IND_JMP |         |
|                               |                                       | 7 NEAR_REL_JMP |         |
|                               |                                       | 8 FAR_BRANCH   |         |
|                               |                                       | 9 EN_CALL_STACK|         |
|                               |                                       | 63:10 Reserved. |         |
| Register Address: 1C9H, 457    | MSR_LASTBRANCH_TOS                    | Last Branch Record Stack TOS (R/W)  
Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.  
See MSR_LASTBRANCH_O_FROM_IP. | Core     |
| Register Address: 1FCH, 508    | MSR_POWER_CTL                         | Power Control Register  
See http://biosbits.org. | Core     |
|                               |                                       | 0 Reserved.    |         |
|                               |                                       | 1 C1E Enable (R/W)  
When set to ‘1’, will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1). | Package |
<p>|                               |                                       | 63:2 Reserved. |         |
| Register Address: 210H, 528    | IA32_MTRR_PHYSBASE8                   | See Table 2-2. | Core     |
| Register Address: 211H, 529    | IA32_MTRR_PHYSMASK8                   | See Table 2-2. | Core     |
| Register Address: 212H, 530    | IA32_MTRR_PHYSBASE9                   | See Table 2-2. | Core     |</p>
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MTRR_PHYSMASK9</td>
<td>213H, 531</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>IA32_MC0_CTL2</td>
<td>280H, 640</td>
<td>See Table 2-2.</td>
<td>Module</td>
</tr>
<tr>
<td>IA32_MC1_CTL2</td>
<td>281H, 641</td>
<td>See Table 2-2.</td>
<td>Module</td>
</tr>
<tr>
<td>IA32_MC2_CTL2</td>
<td>282H, 642</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>IA32_MC3_CTL2</td>
<td>283H, 643</td>
<td>See Table 2-2.</td>
<td>Module</td>
</tr>
<tr>
<td>IA32_MC4_CTL2</td>
<td>284H, 644</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC5_CTL2</td>
<td>285H, 645</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC6_CTL2</td>
<td>286H, 646</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>MSR_SGXOWNEREPOCH0</td>
<td>300H, 768</td>
<td>Lower 64 Bit CR_SGXOWNEREPOCH (W)</td>
<td>Package</td>
</tr>
<tr>
<td>Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.</td>
<td>63:0</td>
<td>Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.</td>
<td></td>
</tr>
<tr>
<td>MSR_SGXOWNEREPOCH1</td>
<td>301H, 769</td>
<td>Upper 64 Bit CR_SGXOWNEREPOCH (w)</td>
<td>Package</td>
</tr>
<tr>
<td>Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.</td>
<td>63:0</td>
<td>Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.</td>
<td></td>
</tr>
<tr>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td>38EH, 910</td>
<td>See Table 2-2 and Section 20.2.4, “Architectural Performance Monitoring Version 4.”</td>
<td>Core</td>
</tr>
<tr>
<td>Ovf_PMC0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ovf_PMC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ovf_PMC2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ovf_PMC3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved.</td>
<td>31:4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ovf_FixedCtr0</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ovf_FixedCtr1</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ovf_FixedCtr2</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved.</td>
<td>54:35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace_ToPA_PMI</td>
<td>55</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>57:56</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>LBR_Frz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>CTR_Frz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>ASCII</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Ovf_Uncore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Ovf_BufDSSAVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>CondChgd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 390H, 912  
IA32_PERF_GLOBAL_STATUS_RESET

See Table 2-2 and Section 20.2.4, “Architectural Performance Monitoring Version 4.”

<table>
<thead>
<tr>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>31:4</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>54:35</td>
</tr>
<tr>
<td>55</td>
</tr>
<tr>
<td>57:56</td>
</tr>
<tr>
<td>58</td>
</tr>
<tr>
<td>59</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>61</td>
</tr>
<tr>
<td>62</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

Register Address: 391H, 913  
IA32_PERF_GLOBAL_STATUS_SET

See Table 2-2 and Section 20.2.4, “Architectural Performance Monitoring Version 4.”

<table>
<thead>
<tr>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>31:4</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>54:35</td>
</tr>
</tbody>
</table>

Vol. 4 2-123
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>392H, 914</td>
<td>IA32_PERF_GLOBAL_INUSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3F1H, 1009</td>
<td>IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>3F8H, 1016</td>
<td>MSR_PKG_C3_RESIDENCY</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>3F9H, 1017</td>
<td>MSR_PKG_C6_RESIDENCY</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>406H, 1030</td>
<td>IA32_MC1_ADDR</td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td>418H, 1048</td>
<td>IA32_MC6_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>419H, 1049</td>
<td>IA32_MC6_STATUS</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

- **Package C3 Residency Counter (R/O)**: Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.

- **Package C6 Residency Counter (R/O)**: Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.

- **CORE C3 Residency Counter (R/O)**: Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.

- **Module IA32_MCI_ADDR MSRs.**
  - The IA32_MCI_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MCI_STATUS register is clear.
  - When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

- **Package IA32_MC6_CTL**
  - See Section 16.3.2.1, “IA32_MCI_CTL MSRS."

- **Package IA32_MC6_STATUS**
  - See Section 16.3.2.2, “IA32_MCI_STATUS MSRS,” and Chapter 17.
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 41AH, 1050</td>
<td>IA32_MCI_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCI_ADDR MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C3H, 1219</td>
<td>IA32_A_PMC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C4H, 1220</td>
<td>IA32_A_PMC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4EOH, 1248</td>
<td>MSR_SMM_FEATURE_CONTROL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td><strong>Enhanced SMM Feature Control (SMM-Rw)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reports SMM capability Enhancement. Accessible only while in SMM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Lock (SMM-RwO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When set to ‘1’ locks this register from further changes.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SMM_Code_Chk_En (SMM-Rw)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to ‘0’ (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to ‘1’ any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 4E2H, 1250</td>
<td>MSR_SMM_DELAYED</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td><strong>SMM Delayed (SMM-Ro)</strong></td>
<td>LOG_PROC_STATE (SMM-Ro)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-1:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=OBH, ECX=PKG_LVL):EBX[15:0] can be updated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:N</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 4E3H, 1251</td>
<td>MSR_SMM_BLOCKED</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td><strong>SMM Blocked (SMM-Ro)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reports the blocked state of all logical processors in the package. Available only while in SMM.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-1:0</td>
<td>LOG_PROC_STATE (SMM-RO)</td>
<td>Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.</td>
<td>Core</td>
</tr>
<tr>
<td>63:N</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 500H, 1280</td>
<td>IA32_SGX_SVN_STATUS</td>
<td>Status and SVN Threshold of SGX Support for ACM (R/O)</td>
<td>Core</td>
</tr>
<tr>
<td>0</td>
<td>Lock</td>
<td>See Section 39.11.3, “Interactions with Authenticated Code Modules (ACMs).”</td>
<td></td>
</tr>
<tr>
<td>15:1</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>SGX_SVN_SINIT</td>
<td>See Section 39.11.3, “Interactions with Authenticated Code Modules (ACMs).”</td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 560H, 1376</td>
<td>IA32_RTIT_OUTPUT_BASE</td>
<td>Trace Output Base Register (R/W) See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 561H, 1377</td>
<td>IA32_RTIT_OUTPUT_MASK_PTRS</td>
<td>Trace Output Mask Pointers Register (R/W) See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 570H, 1392</td>
<td>IA32_RTIT_CTL</td>
<td>Trace Control Register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>0</td>
<td>TraceEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CYCEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>User</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CR3Filter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ToPA</td>
<td>Writing 0 will #GP if also setting TraceEn.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MTCEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>TSCEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>DisRETC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>BranchEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:14</td>
<td>MTCFreq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:19</td>
<td>CycThresh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>PSBFreq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:28</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35:32</td>
<td>ADDRO_CFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:36</td>
<td>ADDR1_CFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:40</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 571H, 1393</td>
<td>IA32_RTIT_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing Status Register (R/W)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>FilterEn</td>
<td>Writes ignored.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ContexEn</td>
<td>Writes ignored.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TriggerEn</td>
<td>Writes ignored.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Error (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Stopped</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:6</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48:32</td>
<td>PacketByteCnt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:49</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 572H, 1394</td>
<td>IA32_RTIT.CR3_MATCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace Filter CR3 Match Register (R/W)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 580H, 1408</td>
<td>IA32_RTIT.ADDR0.A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region 0 Start Address (R/w)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 581H, 1409</td>
<td>IA32_RTIT.ADDR0.B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region 0 End Address (R/w)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 582H, 1410</td>
<td>IA32_RTIT.ADDR1.A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region 1 Start Address (R/w)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 583H, 1411</td>
<td>IA32_RTIT.ADDR1.B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region 1 End Address (R/w)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

#### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 606H, 1542</td>
<td>MSR_RAPL_POWER_UNIT</td>
<td>Unit Multipliers used in RAPL Interfaces (R/O)</td>
<td>Power Units: Power related information (in Watts) is in unit of 1W/2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:8</td>
<td></td>
<td>Energy Status Units</td>
<td>Energy related information (in Joules) is in unit of 1Joule/ (2^ESU); where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.</td>
<td>Package</td>
</tr>
<tr>
<td>15:13</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:16</td>
<td></td>
<td>Time Unit</td>
<td>Time related information (in seconds) is in unit of 1S/2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 60AH, 1546</td>
<td>MSR_PKGC3_IRTL</td>
<td>Package C3 Interrupt Response Limit (R/W)</td>
<td>Interrupt Response Time Limit (R/w): Specifies the limit that should be used to decide if the package should be put into a package C3 state.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td>Interrupt Response Time Limit (R/w)</td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C3 state.</td>
<td>Package</td>
</tr>
<tr>
<td>12:10</td>
<td></td>
<td>Time Unit</td>
<td>Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.</td>
<td>Package</td>
</tr>
<tr>
<td>14:13</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Valid (R/w)</td>
<td>Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.</td>
<td>Package</td>
</tr>
<tr>
<td>63:16</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 60BH, 1547</td>
<td>MSR_PKGC_C7S_IRTL</td>
<td>Package C6/C7S Interrupt Response Limit 1 (R/W)</td>
<td>Interrupt Response Time Limit (R/w): Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7S state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td>Interrupt Response Time Limit (R/w)</td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.</td>
<td>Package</td>
</tr>
</tbody>
</table>
Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:10</td>
<td>Time Unit (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Valid (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 60CH, 1548</td>
<td>MSR_PKG_C2_IRTL2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Package C7 Interrupt Response Limit 2 (R/w)
This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C7 state.

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

| 9:0                             | Interrupt Response Time Limit (R/w) |                 | Package |
|                                | Specifies the limit that should be used to decide if the package should be put into a package C7 state. |                 |       |
| 12:10                          | Time Unit (R/w)                      |                 | Package |
|                                | Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings. |                 |       |
| 14:13                          | Reserved.                             |                 | Package |
| 15                             | Valid (R/w)                           |                 | Package |
|                                | Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management. |                 |       |
| 63:16                          | Reserved.                             |                 | Package |
| Register Address: 60DH, 1549    | MSR_PKG_C2.RESIDENCY                  |                 | Package |

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

| 63:0                           | Package C2 Residency Counter (R/O)   |                 | Package |
|                                | Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC. |                 |       |
| Register Address: 610H, 1552    | MSR_PKG_POWER_LIMIT                   |                 | Package |

PKG RAPL Power Limit Control (R/w)
See Section 15.10.3, “Package RAPL Domain.”

| Register Address: 611H, 1553    | MSR_PKG_ENERGY_STATUS                  |                 | Package |

PKG Energy Status (R/O)
See Section 15.10.3, “Package RAPL Domain.”

| Register Address: 613H, 1555    | MSR_PKG_PERF_STATUS                     |                 | Package |

PKG Perf Status (R/O)
See Section 15.10.3, “Package RAPL Domain.”

| Register Address: 614H, 1556    | MSR_PKG_POWER_INFO                      |                 | Package |

PKG RAPL Parameters (R/W)
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:0</td>
<td>Thermal Spec Power (R/W)</td>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:16</td>
<td>Minimum Power (R/W)</td>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46:32</td>
<td>Maximum Power (R/W)</td>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54:48</td>
<td>Maximum Time Window (R/W)</td>
<td>Specified by $2^Y \times (1.0 + Z/4.0) \times \text{Time_Unit}$, where “Y” is the unsigned integer value represented by bits 52:48, “Z” is an unsigned integer represented by bits 54:53. “Time_Unit” is specified by the “Time Units” field of MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>63:55</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM RAPL Power Limit Control (R/W)</td>
<td>MSR_DRAM_POWER_LIMIT</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>DRAM Energy Status (R/O)</td>
<td>MSR_DRAM_ENERGY_STATUS</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>DRAM Performance Throttling Status (R/O)</td>
<td>MSR_DRAM_PERF_STATUS</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>DRAM RAPL Parameters (R/W)</td>
<td>MSR_DRAM_POWER_INFO</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td>MSR_PKG_C10_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td>Package</td>
</tr>
<tr>
<td>63:0</td>
<td>Package C10 Residency Counter (R/O)</td>
<td>Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.</td>
<td></td>
</tr>
<tr>
<td>PP0 Energy Status (R/O)</td>
<td>MSR_PP0_ENERGY_STATUS</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td>PP1 Energy Status (R/O)</td>
<td>MSR_PP1_ENERGY_STATUS</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
</tbody>
</table>

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.
Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigTDP Control (R/w)</td>
<td>MAX_NON_TURBO_RATIO (Rw/L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>System BIOS can program this field.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:8</td>
<td>TURBO_ACTIVATION_RATIO_LOCK (Rw/L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this bit is set, the content of this register is locked until a reset.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64FH, 1615</td>
<td>MSR_CORE_PERF_LIMIT_REASONS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicator of Frequency Clipping in Processor Cores (R/w)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Frequency refers to processor core frequency.)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to a thermal event.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Package-Level Power Limiting PL1 Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to package-level power limiting PL1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Package-Level PL2 Power Limiting Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to package-level power limiting PL2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8:4</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Core Power Limiting Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to domain-level power limiting.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>VR Therm Alert Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Max Turbo Limit Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to multi-core turbo limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Electrical Design Point Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Turbo Transition Attenuation Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Maximum Efficiency Frequency Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the maximum efficiency frequency.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Package-Level PL1 Power Limiting Log</td>
<td>When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Package-Level PL2 Power Limiting Log</td>
<td>When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>24:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Core Power Limiting Log</td>
<td>When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>VR Therm Alert Log</td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Max Turbo Limit Log</td>
<td>When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Electrical Design Point Log</td>
<td>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Turbo Transition Attenuation Log</td>
<td>When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Maximum Efficiency Frequency Log</td>
<td>When set, indicates that the Maximum Efficiency Frequency Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>63:31</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 680H, 1664

MSR_LASTBRANCH_0_FROM_IP
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Last Branch Record 0 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the source instruction. See also:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Section 18.6 and record format in Section 18.4.8.1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:47</td>
<td>From Linear Address (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:48</td>
<td>Signed extension of bits 47:0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Mispred</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 681H, 1665</td>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 1 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 682H, 1666</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 2 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 683H, 1667</td>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 3 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 684H, 1668</td>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 4 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 685H, 1669</td>
<td>MSR_LASTBRANCH_5_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 5 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 686H, 1670</td>
<td>MSR_LASTBRANCH_6_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 6 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 687H, 1671</td>
<td>MSR_LASTBRANCH_7_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 7 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 688H, 1672</td>
<td>MSR_LASTBRANCH_8_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 8 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 689H, 1673</td>
<td>MSR_LASTBRANCH_9_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 9 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68AH, 1674</td>
<td>MSR_LASTBRANCH_10_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 10 From IP (R/W)</strong></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68BH, 1675</td>
<td>MSR_LASTBRANCH_11_FROM_IP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 11 From IP (R/W)</td>
<td>MSR_LASTBRANCH_12_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 68CH, 1676</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12 From IP (R/W)</td>
<td>MSR_LASTBRANCH_13_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 68DH, 1677</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13 From IP (R/W)</td>
<td>MSR_LASTBRANCH_14_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 68EH, 1678</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14 From IP (R/W)</td>
<td>MSR_LASTBRANCH_15_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 68FH, 1679</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 15 From IP (R/W)</td>
<td>MSR_LASTBRANCH_16_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 690H, 1680</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 16 From IP (R/W)</td>
<td>MSR_LASTBRANCH_17_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 691H, 1681</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 17 From IP (R/W)</td>
<td>MSR_LASTBRANCH_18_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 692H, 1682</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 18 From IP (R/W)</td>
<td>MSR_LASTBRANCH_19_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 693H, 1683</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 19 From IP (R/W)</td>
<td>MSR_LASTBRANCH_20_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 694H, 1684</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 20 From IP (R/W)</td>
<td>MSR_LASTBRANCH_21_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 695H, 1685</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 21 From IP (R/W)</td>
<td>MSR_LASTBRANCH_22_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 696H, 1686</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 22 From IP (R/W)</td>
<td>MSR_LASTBRANCH_23_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 697H, 1687</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 23 From IP (R/W)</td>
<td>MSR_LASTBRANCH_24_FROM_IP</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 698H, 1688</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 24 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 699H, 1689</td>
<td>MSR_LASTBRANCH_25_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 25 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 69AH, 1690</td>
<td>MSR_LASTBRANCH_26_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 26 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 69BH, 1691</td>
<td>MSR_LASTBRANCH_27_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 27 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 69CH, 1692</td>
<td>MSR_LASTBRANCH_28_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 28 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 69DH, 1693</td>
<td>MSR_LASTBRANCH_29_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 29 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 69EH, 1694</td>
<td>MSR_LASTBRANCH_30_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 30 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 69FH, 1695</td>
<td>MSR_LASTBRANCH_31_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 31 From IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C0H, 1728</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 0 To IP (R/W)</td>
<td>One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the Destination instruction and elapsed cycles from last LBR update. See Section 18.6.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:47 Target Linear Address (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48 Elapsed cycles from last update to the LBR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C1H, 1729</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 1 To IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C2H, 1730</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 2 To IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C3H, 1731</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 3 To IP (R/W)</strong></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C4H, 1732</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Last Branch Record 4 To IP (R/W)</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6C5H, 1733</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 5 To IP (R/W)</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6C6H, 1734</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6 To IP (R/W)</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6C7H, 1735</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7 To IP (R/W)</td>
<td>MSR_LASTBRANCH_8_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6C8H, 1736</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 8 To IP (R/W)</td>
<td>MSR_LASTBRANCH_9_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6C9H, 1737</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 9 To IP (R/W)</td>
<td>MSR_LASTBRANCH_10_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6CAH, 1738</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 10 To IP (R/W)</td>
<td>MSR_LASTBRANCH_11_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6CBH, 1739</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11 To IP (R/W)</td>
<td>MSR_LASTBRANCH_12_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6CCH, 1740</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12 To IP (R/W)</td>
<td>MSR_LASTBRANCH_13_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6CDH, 1741</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13 To IP (R/W)</td>
<td>MSR_LASTBRANCH_14_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6CEH, 1742</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14 To IP (R/W)</td>
<td>MSR_LASTBRANCH_15_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6CFH, 1743</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 15 To IP (R/W)</td>
<td>MSR_LASTBRANCH_16_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6D0H, 1744</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 16 To IP (R/W)</td>
<td>MSR_LASTBRANCH_17_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 6D1H, 1745</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Last Branch Record 17 To IP (R/W)</td>
<td>MSR_LASTBRANCH_18_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D2H, 1746</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 18 To IP (R/W)</td>
<td>MSR_LASTBRANCH_19_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D3H, 1747</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 19 To IP (R/W)</td>
<td>MSR_LASTBRANCH_20_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D4H, 1748</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 20 To IP (R/W)</td>
<td>MSR_LASTBRANCH_21_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D5H, 1749</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 21 To IP (R/W)</td>
<td>MSR_LASTBRANCH_22_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D6H, 1750</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 22 To IP (R/W)</td>
<td>MSR_LASTBRANCH_23_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D7H, 1751</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 23 To IP (R/W)</td>
<td>MSR_LASTBRANCH_24_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D8H, 1752</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 24 To IP (R/W)</td>
<td>MSR_LASTBRANCH_25_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6D9H, 1753</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 25 To IP (R/W)</td>
<td>MSR_LASTBRANCH_26_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DAH, 1754</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 26 To IP (R/W)</td>
<td>MSR_LASTBRANCH_27_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DBH, 1755</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 27 To IP (R/W)</td>
<td>MSR_LASTBRANCH_28_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DCH, 1756</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 28 To IP (R/W)</td>
<td>MSR_LASTBRANCH_29_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DEH, 1757</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 29 To IP (R/W)</td>
<td>MSR_LASTBRANCH_30_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Hex, Decimal</th>
<th>Register Name ( Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 30 To IP (R/W)</td>
<td>MSR_LASTBRANCH_30_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DFH, 1759</td>
<td>MSR_LASTBRANCH_31_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Last Branch Record 31 To IP (R/W)</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 802H, 2050</td>
<td>IA32_X2APIC_APICID</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC ID register (R/O)</td>
<td>IA32_X2APIC_VERSION</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 803H, 2051</td>
<td>IA32_X2APIC_TPR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC Task Priority register (R/W)</td>
<td>IA32_X2APIC_PPR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 80AH, 2058</td>
<td>IA32_X2APIC_EOI</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC EOI register (w/o)</td>
<td>IA32_X2APIC_LDR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 80DH, 2061</td>
<td>IA32_X2APIC_SIVR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC Spurious Interrupt Vector register (R/w)</td>
<td>IA32_X2APIC_ISR0</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 810H, 2064</td>
<td>IA32_X2APIC_ISR1</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC In-Service register bits [31:0] (R/O)</td>
<td>IA32_X2APIC_ISR2</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 812H, 2066</td>
<td>IA32_X2APIC_ISR3</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC In-Service register bits [63:32] (R/O)</td>
<td>IA32_X2APIC_ISR4</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 813H, 2067</td>
<td>IA32_X2APIC_ISR5</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC In-Service register bits [95:64] (R/O)</td>
<td>IA32_X2APIC_ISR6</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 815H, 2069</td>
<td>IA32_X2APIC_ISR7</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC In-Service register bits [127:96] (R/O)</td>
<td>IA32_X2APIC_TMR0</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 816H, 2070</td>
<td>IA32_X2APIC_TMR1</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [31:0] (R/O)</td>
<td>IA32_X2APIC_TMR2</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 817H, 2071</td>
<td>IA32_X2APIC_TMR3</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [223:192] (R/O)</td>
<td>IA32_X2APIC_TMR4</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 818H, 2072</td>
<td>IA32_X2APIC_TMR5</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [255:224] (R/O)</td>
<td>IA32_X2APIC_TMR6</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 819H, 2073</td>
<td>IA32_X2APIC_TMR7</td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>x2APIC Trigger Mode register bits [63:32] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 81AH, 2074</td>
<td>IA32_X2APIC_TMR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [95:64] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 81BH, 2075</td>
<td>IA32_X2APIC_TMR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [127:96] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 81CH, 2076</td>
<td>IA32_X2APIC_TMR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [159:128] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 81DH, 2077</td>
<td>IA32_X2APIC_TMR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [191:160] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 81EH, 2078</td>
<td>IA32_X2APIC_TMR6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [223:192] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 81FH, 2079</td>
<td>IA32_X2APIC_TMR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode register bits [255:224] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 820H, 2080</td>
<td>IA32_X2APIC_IRR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [31:0] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 821H, 2081</td>
<td>IA32_X2APIC_IRR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [63:32] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 822H, 2082</td>
<td>IA32_X2APIC_IRR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [95:64] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 823H, 2083</td>
<td>IA32_X2APIC_IRR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [127:96] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 824H, 2084</td>
<td>IA32_X2APIC_IRR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [159:128] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 825H, 2085</td>
<td>IA32_X2APIC_IRR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [191:160] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 826H, 2086</td>
<td>IA32_X2APIC_IRR6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [223:192] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 827H, 2087</td>
<td>IA32_X2APIC_IRR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request register bits [255:224] (R/O)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 828H, 2088</td>
<td>IA32_X2APIC_ESR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Error Status register (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 829H, 2095</td>
<td>IA32_X2APIC_LVT_CMCI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Corrected Machine Check Interrupt register (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 830H, 2096</td>
<td>IA32_X2APIC_ICR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Command register (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 832H, 2098</td>
<td>IA32_X2APIC_LVT_TIMER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Timer Interrupt register (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>
### Table 2-12. MSRs in Intel Atom® Processors Based on Goldmont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 833H, 2099</td>
<td>IA32_X2APIC_LVT_THERMAL</td>
<td>x2APIC LVT Thermal Sensor Interrupt register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 834H, 2100</td>
<td>IA32_X2APIC_LVT_PMI</td>
<td>x2APIC LVT Performance Monitor register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 835H, 2101</td>
<td>IA32_X2APIC_LVT_LINT0</td>
<td>x2APIC LVT LINT0 register (R/w)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 836H, 2102</td>
<td>IA32_X2APIC_LVT_LINT1</td>
<td>x2APIC LVT LINT1 register (R/w)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 837H, 2103</td>
<td>IA32_X2APIC_LVT_ERROR</td>
<td>x2APIC LVT Error register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 838H, 2104</td>
<td>IA32_X2APIC_INIT_COUNT</td>
<td>x2APIC Initial Count register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 839H, 2105</td>
<td>IA32_X2APIC_CUR_COUNT</td>
<td>x2APIC Current Count register (R/O)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 83EH, 2110</td>
<td>IA32_X2APIC_DIV_CONF</td>
<td>x2APIC Divide Configuration register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 83FH, 2111</td>
<td>IA32_X2APIC_SELF_IPI</td>
<td>x2APIC Self IPI register (w/O)</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: C8FH, 3215</td>
<td>IA32_PQR_ASSOC</td>
<td>Resource Association Register (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>31:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33:32</td>
<td>COS (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63: 34</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D10H, 3344</td>
<td>IA32_L2_QOS_MASK_0</td>
<td>L2 Class Of Service Mask - COS 0 (R/W)</td>
<td>Module</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=0.</td>
<td></td>
</tr>
<tr>
<td>0:7</td>
<td>CBM: Bit vector of available L2 ways for COS 0 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D11H, 3345</td>
<td>IA32_L2_QOS_MASK_1</td>
<td>L2 Class Of Service Mask - COS 1 (R/W)</td>
<td>Module</td>
</tr>
<tr>
<td>0:7</td>
<td>CBM: Bit vector of available L2 ways for COS 0 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D12H, 3346</td>
<td>IA32_L2_QOS_MASK_2</td>
<td>L2 Class Of Service Mask - COS 2 (R/W)</td>
<td>Module</td>
</tr>
<tr>
<td>0:7</td>
<td>CBM: Bit vector of available L2 ways for COS 0 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.6 **MSRS IN INTEL ATOM® PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE**

Intel Atom processors based on the Goldmont Plus microarchitecture support MSRs listed in Table 2-6, Table 2-12, and Table 2-13. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_7AH; see Table 2-1. For an MSR listed in Table 2-13 that also appears in the model-specific tables of prior generations, Table 2-13 supersedes prior generation tables.

In the Goldmont Plus microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Goldmont Plus microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. “Package” means all processor cores in the physical package share the same MSR or bit interface.

**Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td>Control Features in Intel 64Processor (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>0</td>
<td>Lock (R/WL)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable VMX inside SMX operation (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable VMX outside SMX operation (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>SENTER local functions enables (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SENTER global functions enable (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>SGX Launch Control Enable (R/WL)</td>
<td>This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Valid if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>SGX global functions enable (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 8CH, 140</td>
<td>IA32_SGXLEPUBKEYHASH0</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 8DH, 141</td>
<td>IA32_SGXLEPUBKEYHASH1</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 8EH, 142</td>
<td>IA32_SGXLEPUBKEYHASH2</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 8FH, 143</td>
<td>IA32_SGXLEPUBKEYHASH3</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBs_ENABLE (MSR_PEBs_ENABLE)</td>
<td>(R/W) See Table 2-2. See Section 20.6.2.4, “Processor Event Based Sampling (PEBS).”</td>
<td>Core</td>
</tr>
<tr>
<td>0</td>
<td>Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Enable PEBS trigger and recording for IA32_FIXED_CTR0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Enable PEBS trigger and recording for IA32_FIXED_CTR1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Enable PEBS trigger and recording for IA32_FIXED_CTR2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 570H, 1392</td>
<td>IA32_RTIT_CTL</td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>

#### Trace Control Register (R/W)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TraceEn</td>
</tr>
<tr>
<td>1</td>
<td>CYCEn</td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
</tr>
<tr>
<td>3</td>
<td>User</td>
</tr>
<tr>
<td>4</td>
<td>PwrEvtEn</td>
</tr>
<tr>
<td>5</td>
<td>FUPonPTW</td>
</tr>
<tr>
<td>6</td>
<td>FabricEn</td>
</tr>
<tr>
<td>7</td>
<td>CR3Filter</td>
</tr>
<tr>
<td>8</td>
<td>ToPA</td>
</tr>
<tr>
<td></td>
<td>Writing 0 will #GP if also setting TraceEn.</td>
</tr>
<tr>
<td>9</td>
<td>MTCEn</td>
</tr>
<tr>
<td>10</td>
<td>TSCEn</td>
</tr>
</tbody>
</table>
### Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>DisRETC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>PTWEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>BranchEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:14</td>
<td>MTCFreq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:19</td>
<td>CycThresh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>PSBFreq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:28</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35:32</td>
<td>ADDR0_CFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:36</td>
<td>ADDR1_CFG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:40</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 680H, 1664</td>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Last Branch Record 0 From IP (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>One of the three MSRs that make up the first entry of the 32-entry LBR stack. The From_IP part of the stack contains pointers to the source instruction. See also:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Section 18.7, &quot;Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 681H—69FH, 1665—1695</td>
<td>MSR_LASTBRANCH_i_FROM_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Last Branch Record / From IP (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP; i = 1-31.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6COH, 1728</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Last Branch Record 0 To IP (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>One of the three MSRs that make up the first entry of the 32-entry LBR stack. The To_IP part of the stack contains pointers to the Destination instruction. See also:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Section 18.7, &quot;Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C1H—6DFH, 1729—1759</td>
<td>MSR_LASTBRANCH_i_TO_IP</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Last Branch Record / To IP (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP; i = 1-31.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DC0H, 3520</td>
<td>MSR_LASTBRANCH_INFO_0</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Last Branch Record 0 Additional Information (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>One of the three MSRs that make up the first entry of the 32-entry LBR stack. This part of the stack contains flag and elapsed cycle information. See also:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Section 18.9.1, &quot;LBR Stack.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DC1H, 3521</td>
<td>MSR_LASTBRANCH_INFO_1</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Last Branch Record 1 Additional Information (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See description of MSR_LASTBRANCH_INFO_0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC2H, 3522</td>
<td>MSR_LASTBRANCH_INFO_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC3H, 3523</td>
<td>MSR_LASTBRANCH_INFO_3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC4H, 3524</td>
<td>MSR_LASTBRANCH_INFO_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC5H, 3525</td>
<td>MSR_LASTBRANCH_INFO_5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC6H, 3526</td>
<td>MSR_LASTBRANCH_INFO_6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC7H, 3527</td>
<td>MSR_LASTBRANCH_INFO_7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC8H, 3528</td>
<td>MSR_LASTBRANCH_INFO_8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC9H, 3529</td>
<td>MSR_LASTBRANCH_INFO_9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCAH, 3530</td>
<td>MSR_LASTBRANCH_INFO_10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCBH, 3531</td>
<td>MSR_LASTBRANCH_INFO_11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCCH, 3532</td>
<td>MSR_LASTBRANCH_INFO_12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCDH, 3533</td>
<td>MSR_LASTBRANCH_INFO_13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCEH, 3534</td>
<td>MSR_LASTBRANCH_INFO_14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-13. MSRs in Intel Atom® Processors Based on Goldmont Plus Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCFH, 3535</td>
<td>MSR_LASTBRANCH_INFO_15</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DDOH, 3536</td>
<td>MSR_LASTBRANCH_INFO_16</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD1H, 3537</td>
<td>MSR_LASTBRANCH_INFO_17</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD2H, 3538</td>
<td>MSR_LASTBRANCH_INFO_18</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD3H, 3539</td>
<td>MSR_LASTBRANCH_INFO_19</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD4H, 3540</td>
<td>MSR_LASTBRANCH_INFO_20</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD5H, 3541</td>
<td>MSR_LASTBRANCH_INFO_21</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD6H, 3542</td>
<td>MSR_LASTBRANCH_INFO_22</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD7H, 3543</td>
<td>MSR_LASTBRANCH_INFO_23</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD8H, 3544</td>
<td>MSR_LASTBRANCH_INFO_24</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DD9H, 3545</td>
<td>MSR_LASTBRANCH_INFO_25</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DDAH, 3546</td>
<td>MSR_LASTBRANCH_INFO_26</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>DDBH, 3547</td>
<td>MSR_LASTBRANCH_INFO_27</td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>
2.7 MSRS IN INTEL ATOM® PROCESSORS BASED ON TREMONT MICROARCHITECTURE

Processors based on the Tremont microarchitecture support MSRs listed in Table 2-6, Table 2-12, Table 2-13, and Table 2-14. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_86H, 06_96H, or 06_9CH; see Table 2-1. For an MSR listed in Table 2-14 that also appears in the model-specific tables of prior generations, Table 2-14 supersedes prior generation tables.

In the Tremont microarchitecture, the scope column indicates the following: “Core” means each processor core has a separate MSR, or a bit field not shared with another processor core. “Module” means the MSR or the bit field is shared by a subset of the processor cores in the physical package. The number of processor cores in this subset is model specific and may differ between different processors. For all processors based on Tremont microarchitecture, the L2 cache is also shared between cores in a module and thus CPUID leaf 04H enumeration can be used to figure out which processors are in the same module. “Package” means all processor cores in the physical package share the same MSR or bit interface.

### Table 2-14. MSRs in Intel Atom® Processors Based on Tremont Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Control Register</td>
<td>MSR_MEMORY_CTRL</td>
<td>Bit Description</td>
<td>Core</td>
</tr>
<tr>
<td>28:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>SPLIT_LOCK_DISABLE</td>
<td>If set to 1, a split lock will cause an #AC(0) exception. See Section 9.1.2.3, “Features to Disable Bus Locks.”</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: CFH, 207</td>
<td>IA32_CORE_CAPABILITIES</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
</tbody>
</table>

See Table 2-6, Table 2-12, and Table 2-13 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_7AH.
Table 2-14. MSRs in Intel Atom® Processors Based on Tremont Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SPLIT_LOCK_DISABLE_SUPPORTED When read as 1, software can set bit 29 of MSR_MEMORY_CTRL (MSR address 33H).</td>
<td></td>
</tr>
<tr>
<td>63:6</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 2A0H, 672 MSR_PRMRR_BASE_0

Processor Reserved Memory Range Register - Physical Base Control Register (R/W) Core

<table>
<thead>
<tr>
<th>Register Address</th>
<th>3F1H, 1009 IA32_PEMS_ENABLE (MSR_PEMS_ENABLE) (R/W) See Table 2-2. See Section 20.6.2.4, “Processor Event Based Sampling (PEBS).”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core n:0</td>
<td>Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMCx. The maximum value n can be determined from CPUID.0AH:EAX[15:8].</td>
</tr>
<tr>
<td>31:n+1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>32+m:32</td>
<td>Enable PEBS trigger and recording for IA32_FIXED_CTRx. The maximum value m can be determined from CPUID.0AH:EDX[4:0].</td>
</tr>
<tr>
<td>59:33+m</td>
<td>Reserved.</td>
</tr>
<tr>
<td>60</td>
<td>Pend a PerfMon Interrupt (PMI) after each PEBS event.</td>
</tr>
<tr>
<td>62:61</td>
<td>Specifies PEBS output destination. Encodings: 00B: DS Save Area. 01B: Intel PT trace output. Supported if IA32_PERF_CAPABILITIES.PEMS_OUTPUT_PT_AVAIL[16] and CPUID.07H.0.EBX[25] are set. 10B: Reserved. 11B: Reserved.</td>
</tr>
<tr>
<td>63</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 1309H–130BH, 4873–4875 MSR_RELOAD_FIXED_CTRx

Reload value for IA32_FIXED_CTRx (R/w)

<table>
<thead>
<tr>
<th>Register Address</th>
<th>14C1H–14C4H, 5313–5316 MSR_RELOAD_PMCx</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:0</td>
<td>Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEMS_ENABLE, and FIXED_CTRx is overflowed.</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
MODEL-SPECIFIC REGISTERS (MSRS)

2.8 MSRS IN PROCESSORS BASED ON NEHALEM MICROARCHITECTURE

Table 2-15 lists model-specific registers (MSRs) that are common for Nehalem microarchitecture. These include the Intel Core i7 and i5 processor family. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_1AH, 06_1EH, 06_1FH, or 06_2EH; see Table 2-1. Additional MSRs specific to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_1AH, 06_1EH, or 06_1FH are listed in Table 2-16. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at http://biosbits.org.

The column "Scope" represents the package/core/thread scope of individual bit field of an MSR. "Thread" means this bit field must be programmed on each logical processor independently. "Core" means the bit field must be programmed on each processor core independently, logical processors in the same core will be affected by change of this bit on the other logical processor in the same core. "Package" means the bit field must be programmed once for each physical package. Change of a bit filed with a package scope will affect all logical processors in that physical package.

Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>Reload value for IA32_PMCx (R/W)</td>
<td>Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PBS_ENABLE, and PMCx is overflowed.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>47:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-6, Table 2-12, Table 2-13, and Table 2-14 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_86H.
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 11.4.4, “Local APIC Status and Location,” and Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 34H, 52</td>
<td><strong>MSR_SMI_COUNT</strong></td>
<td><strong>SMI Count (R/O)</strong>&lt;br&gt;Running count of SMI events since last RESET.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 3AH, 58</td>
<td><strong>IA32_FEATURE_CONTROL</strong></td>
<td><strong>Control Features in Intel 64Processor (R/W)</strong>&lt;br&gt;See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 79H, 121</td>
<td><strong>IA32_BIOS_UPDT_TRIG</strong></td>
<td>BIOS Update Trigger Register (w)&lt;br&gt;See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> BBH, 139</td>
<td><strong>IA32_BIOS_SIGN_ID</strong></td>
<td>BIOS Update Signature ID (R/W)&lt;br&gt;See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> C1H, 193</td>
<td><strong>IA32_PMC0</strong></td>
<td>Performance Counter Register&lt;br&gt;See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> C2H, 194</td>
<td><strong>IA32_PMC1</strong></td>
<td>Performance Counter Register&lt;br&gt;See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> C3H, 195</td>
<td><strong>IA32_PMC2</strong></td>
<td>Performance Counter Register&lt;br&gt;See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> C4H, 196</td>
<td><strong>IA32_PMC3</strong></td>
<td>Performance Counter Register&lt;br&gt;See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> CEH, 206</td>
<td><strong>MSR_PLATFORM_INFO</strong></td>
<td><strong>Platform Information</strong>&lt;br&gt;Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>7:0</strong></td>
<td></td>
<td><strong>Reserved.</strong>&lt;br&gt;Maximum Non-Turbo Ratio (R/O)&lt;br&gt;This is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>15:8</strong></td>
<td></td>
<td><strong>Reserved.</strong>&lt;br&gt;Programmable Ratio Limit for Turbo Mode (R/O)&lt;br&gt;When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>27:16</strong></td>
<td></td>
<td><strong>Reserved.</strong>&lt;br&gt;Maximum Non-Turbo Ratio (R/O)&lt;br&gt;This is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>28</strong></td>
<td></td>
<td><strong>Reserved.</strong>&lt;br&gt;Programmable Ratio Limit for Turbo Mode (R/O)&lt;br&gt;When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td>Package</td>
</tr>
</tbody>
</table>

**Vol. 4 2-149**
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Programmable TDC-TDP Limit for Turbo Mode (R/O)</td>
<td>When set to 1, indicates that TDC and TDP Limits for Turbo mode are programmable. When set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.</td>
<td>Package</td>
</tr>
<tr>
<td>39:30</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 133.33MHz.</td>
<td>Package</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### C-State Configuration Control (R/W)

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The following C-state code name encodings are supported:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>000b: C0 (no package C-state support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>001b: C1 (Behavior is the same as 000b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010b: C3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>011b: C6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100b: C7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>101b and 110b: Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111: No package C-state limit.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: This field cannot be used to limit package C-state to C3.</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.</td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/W/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, locks bits 15:0 of this register until next reset.</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Interrupt filtering enable (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>C3 state auto demotion enable (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>C1 state auto demotion enable (R/W)&lt;br&gt;When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Enable C3 Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Enable C1 Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Package C State Demotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Package C State Undemotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:31</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: E4H, 228<br>MSR_PMG_IO_CAPTURE_BASE<br>Power Management IO Redirection in C-state (R/W)<br>See http://biosbits.org.<br>Core

| 15:0                          | LVL_2 Base Address (R/W)<br>Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software. |                |       |
| 18:16                         | C-state Range (R/W)<br>Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]:<br>000b - C3 is the max C-State to include.<br>001b - C6 is the max C-State to include.<br>010b - C7 is the max C-State to include. |                |       |
| 63:19                         | Reserved.                       |                |       |

Register Address: E7H, 231<br>IA32_MPERF<br>Maximum Performance Frequency Clock Count (R/W)<br>See Table 2-2.<br>Thread

Register Address: E8H, 232<br>IA32_APERF<br>Actual Performance Frequency Clock Count (R/W)<br>See Table 2-2.<br>Thread

Register Address: FEH, 254<br>IA32_MTRRCAP<br>See Table 2-2.<br>Thread

Register Address: 174H, 372<br>IA32_SYSENTER_CS<br>See Table 2-2.<br>Thread

Register Address: 175H, 373<br>IA32_SYSENTER_ESP<br>See Table 2-2.<br>Thread

Register Address: 176H, 374<br>IA32_SYSENTER_EIP<br>See Table 2-2.<br>Thread

Register Address: 179H, 377<br>IA32_MCG_CAP<br>See Table 2-2.<br>Thread
Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td>Global Machine Check Status</td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>RIPV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.</td>
<td>Thread</td>
</tr>
<tr>
<td>1</td>
<td>EIPV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.</td>
<td>Thread</td>
</tr>
<tr>
<td>2</td>
<td>MCIP</td>
<td>When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.</td>
<td>Thread</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>186H, 390</td>
<td>IA32_PERFEVTSEL0</td>
<td>7:0 Event Select</td>
<td>Thread</td>
</tr>
<tr>
<td>15:8</td>
<td>UMask</td>
<td>16 USR</td>
<td>Thread</td>
</tr>
<tr>
<td>16</td>
<td>USR</td>
<td>17 OS</td>
<td>Thread</td>
</tr>
<tr>
<td>17</td>
<td>OS</td>
<td>18 Edge</td>
<td>Thread</td>
</tr>
<tr>
<td>18</td>
<td>Edge</td>
<td>19 PC</td>
<td>Thread</td>
</tr>
<tr>
<td>19</td>
<td>PC</td>
<td>20 INT</td>
<td>Thread</td>
</tr>
<tr>
<td>20</td>
<td>INT</td>
<td>21 AnyThread</td>
<td>Thread</td>
</tr>
<tr>
<td>21</td>
<td>AnyThread</td>
<td>22 EN</td>
<td>Thread</td>
</tr>
<tr>
<td>22</td>
<td>EN</td>
<td>23 INV</td>
<td>Thread</td>
</tr>
<tr>
<td>23</td>
<td>INV</td>
<td>31:24 CMASK</td>
<td>Thread</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>187H, 391</td>
<td>IA32_PERFEVTSEL1</td>
<td>15:0 Current Performance State Value.</td>
<td>Thread</td>
</tr>
<tr>
<td>188H, 392</td>
<td>IA32_PERFEVTSEL2</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>189H, 393</td>
<td>IA32_PERFEVTSEL3</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>15:0</td>
<td>Current Performance State Value.</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19AH, 410</td>
<td>IA32_CLOCK_MODULATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Modulation (R/W)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:1</td>
<td>On demand Clock Modulation Duty Cycle (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>4</td>
<td>On demand Clock Modulation Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:5</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19BH, 411</td>
<td>IA32_THERM_INTERRUPT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thermal Interrupt Control (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable Misc. Processor Features (R/W)</td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Fast-Strings Enable</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Default value is 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Processor Event Based Sampling Unavailable (R/O)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Enhanced Intel SpeedStep Technology Enable (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM. (R/W) See Table 2-2.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>21:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Limit CPUID Maxval (R/W)</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>33:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>XD Bit Disable (R/W)</td>
<td>See Table 2-3.</td>
<td>Thread</td>
</tr>
<tr>
<td>37:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Turbo Mode Disable (R/W)</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
|                               | When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0).  
|                               | When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor’s support of turbo mode is enabled.  
|                               | Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.  |     |       |
| 63:39                         | Reserved.                            |                 |       |
| Register Address: 1A2H, 418    | MSR_TEMPERATURE_TARGET               | Temperature Target | Thread|
| 15:0                          | Reserved.                            |                 |       |
| 23:16                         | Temperature Target (R)               | The minimum temperature at which PROCHOT# will be asserted. The value is degrees C. |       |
| 63:24                         | Reserved.                            |                 |       |
| Register Address: 1A4H, 420    | MSR_MISC_FEATURE_CONTROL             | Miscellaneous Feature Control (R/W) | |
| 0                             | L2 Hardware Prefetcher Disable (R/W) | If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache. | Core  |
| 1                             | L2 Adjacent Cache Line Prefetcher Disable (R/W) | If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes). | Core  |
| 2                             | DCU Hardware Prefetcher Disable (R/W) | If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache. | Core  |
| 3                             | DCU IP Prefetcher Disable (R/W)      | If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines. | Core  |
| 63:4                          | Reserved.                            |                 |       |
| Register Address: 1A6H, 422    | MSR_OFFCORE_RSP_0                    |                 |       |
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offcore Response Event Select Register (R/W)</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1AAH, 426</td>
<td>MSR_MISC_PWR_MGMT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>EIST Hardware Coordination Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores. When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Energy/Performance Bias Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1ACH, 428</td>
<td>MSR_TURBO_POWER_CURRENT_LIMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:0</td>
<td>TDP Limit (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDP limit in 1/8 Watt granularity.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>TDP Limit Override Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A value = 0 indicates override is not active; a value = 1 indicates override is active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:16</td>
<td>TDC Limit (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDC limit in 1/8 Amp granularity.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>TDC Limit Override Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A value = 0 indicates override is not active; a value = 1 indicates override is active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 1 core active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 2C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 2 core active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 3C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 3 core active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 4C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 4 core active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1C8H, 456</td>
<td>MSR_LBR_SELECT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record Filtering Select Register (R/w)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 18.9.2, “Filtering of Last Branch Records.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CPL_EQ_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CPL_NEQ_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JCC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NEAR_REL_CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NEAR_IND_CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>NEAR_RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>NEAR_IND_JMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>NEAR_REL_JMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>FAR_BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1C9H, 457</strong></td>
<td>MSR_LASTBRANCH_TOS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Last Branch Record Stack TOS (R/w)</td>
<td>Contains an index (bits 0-3) that points to the MSR containing the most recent branch record.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See MSR_LASTBRANCH_0_FROM_IP (at 680H).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1D9H, 473</strong></td>
<td>IA32_DEBUGCTL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Debug Control (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1DDH, 477</strong></td>
<td>MSR_LER_FROM_LIP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Last Exception Record From Linear IP (R)</td>
<td>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1DEH, 478</strong></td>
<td>MSR_LER_TO_LIP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Last Exception Record To Linear IP (R)</td>
<td>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1F2H, 498</strong></td>
<td>IA32_SMRR_PHYSBASE</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1F3H, 499</strong></td>
<td>IA32_SMRR_PHYSMASK</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 1FCH, 508</strong></td>
<td>MSR_POWER_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C1E Enable (R/w)</td>
<td>When set to ‘1’, will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).</td>
<td>Package</td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MTRR_PHYSBASE0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSMASK0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSMASK1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSMASK2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSMASK3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSMASK4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MTRR_PHYSBASE9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Information / Bit Fields

See Table 2-2.
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 213H, 531</td>
<td>IA32_MTRR_PHYSMASK9</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 250H, 592</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 258H, 600</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 259H, 601</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 26AH, 618</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>IA32_MTRR_FIX4K_DB0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 280H, 640</td>
<td>IA32_MCO_CCTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 281H, 641</td>
<td>IA32_MC1_CCTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 282H, 642</td>
<td>IA32_MC2_CCTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 283H, 643</td>
<td>IA32_MC3_CCTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 284H, 644</td>
<td>IA32_MC4_CCTL2</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 285H, 645</td>
<td>IA32_MC5_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 286H, 646</td>
<td>IA32_MC6_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 287H, 647</td>
<td>IA32_MC7_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 288H, 648</td>
<td>IA32_MC8_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Default Memory Types (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Fixed-Function Performance Counter Register 0 (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Fixed-Function Performance Counter Register 1 (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 30BH, 779</td>
<td>IA32_FIXED_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Fixed-Function Performance Counter Register 2 (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 18.4.1, “IA32_DEBUGCTL MSR.”</td>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>5:0</td>
<td>LBR Format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PEBS Record Format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PEBS_SAVEARCH_REGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>PEBS_REC_FORMAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SMM_FREEZE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 380H, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Fixed-Function-Counter Control Register (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 38EH, 910</td>
<td>MSR_PERF_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provides single-bit status used by software to query the overflow condition of each performance counter. (R/O)</td>
<td></td>
<td>Thread</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>UNC_Ovf</td>
<td>Uncore overflowed if 1.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 38FH, 911</td>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>CLR_UNC_Ovf</td>
<td>Set 1 to clear UNC_Ovf.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBE_ENABLE (MSR_PEBE_ENABLE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.1.1, “Processor Event Based Sampling (PEBS).”</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Enable PEBS on IA32_PMC0 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable PEBS on IA32_PMC1 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable PEBS on IA32_PMC2 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Enable PEBS on IA32_PMC3 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Enable Load Latency on IA32_PMC0 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Enable Load Latency on IA32_PMC1 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Enable Load Latency on IA32_PMC2 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Enable Load Latency on IA32_PMC3 (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:36</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F6H, 1014</td>
<td>MSR_PEBE_LD_LAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.1.2, “Load Latency Performance Monitoring Facility.”</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Minimum threshold latency value of tagged load operation that will be counted. (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:36</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F8H, 1016</td>
<td>MSR_PKG_C3_RESIDENCY</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C3 Residency Counter (R/O)</td>
<td>Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3F9H, 1017</td>
<td>MSR_PKG_C6_RESIDENCY</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>Package C6 Residency Counter (R/O)</td>
<td>Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3FAH, 1018</td>
<td>MSR_PKG_C6_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C7 Residency Counter (R/O)</td>
<td>Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3FCH, 1020</td>
<td>MSR_PKG_C7_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C3 Residency Counter (R/O)</td>
<td>Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 3FDH, 1021</td>
<td>MSR_CORE_C3_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C6 Residency Counter (R/O)</td>
<td>Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCI_STATUS MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCI_ADDR MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 403H, 1027</td>
<td>IA32_MC0_MISC</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MCI_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MCI1_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCI_STATUS MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 406H, 1030</td>
<td>IA32_MCI1_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCI_ADDR MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The IA32_MCI1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MCI1_STATUS register is clear.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vol. 4 2-161
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 407H, 1031</td>
<td>IA32_MC1_MISC</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40BH, 1035</td>
<td>IA32_MC2_MISC</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 40DH, 1037</td>
<td>IA32_MC3_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MC3_ADDR</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40FH, 1039</td>
<td>IA32_MC3_MISC</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
<td>IA32_MC4_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 411H, 1041</td>
<td>IA32_MC4_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 412H, 1042</td>
<td>IA32_MC4_ADDR</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 413H, 1043</td>
<td>IA32_MC4_MISC</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 416H, 1046</td>
<td>IA32_MC5_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 417H, 1047</td>
<td>IA32_MC5_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 418H, 1048</td>
<td>IA32_MC6_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 419H, 1049</td>
<td>IA32_MC6_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs,” and Chapter 17.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41AH, 1050</td>
<td>IA32_MC6_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41BH, 1051</td>
<td>IA32_MC6_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41CH, 1052</td>
<td>IA32_MC7_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41DH, 1053</td>
<td>IA32_MC7_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41EH, 1054</td>
<td>IA32_MC7_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41FH, 1055</td>
<td>IA32_MC7_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 420H, 1056</td>
<td>IA32_MC8_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 421H, 1057</td>
<td>IA32_MC8_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 422H, 1058</td>
<td>IA32_MC8_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 423H, 1059</td>
<td>IA32_MC8_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 480H, 1152</td>
<td>IA32_VMX_BASIC</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Reporting Register of Basic VMX Capabilities (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2 and Appendix A.1, “Basic VMX Information.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 481H, 1153</td>
<td>IA32_VMX_PINBASED_CTLS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Capability Reporting Register of Pin-based VM-execution Controls (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2 and Appendix A.3, “VM-Execution Controls.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 482H, 1154</td>
<td>IA32_VMX_PROCBASED_CTLS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.3, “VM-Execution Controls.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 483H, 1155</td>
<td>IA32_VMX_EXIT_CTLS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Reporting Register of VMX Exit Control (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Exit Controls (R/O)</td>
<td>IA32_VMX_ENTRY_CTLS</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 484H, 1156</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Entry Controls (R/O)</td>
<td>IA32_VMX_MISC</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 485H, 1157</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
<td>IA32_VMX_CR0_FIXED0</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 486H, 1158</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
<td>IA32_VMX_CR0_FIXED1</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 487H, 1159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
<td>IA32_VMX_CR4_FIXED0</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 488H, 1160</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
<td>IA32_VMX_CR4_FIXED1</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 489H, 1161</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)</td>
<td>IA32_VMX_PROCBASED_CTLCS2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 48AH, 1162</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS Save Area (R/W)</td>
<td>IA32_DS_AREA</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 600H, 1536</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 0 From IP (R/W)</td>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 680H, 1664</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 1 From IP (R/W)</td>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 681H, 1665</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 2 From IP (R/W)</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 682H, 1666</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Last Branch Record 2 From IP (R/W)</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 683H, 1667</td>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 3 From IP (R/W)</td>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 684H, 1668</td>
<td>MSR_LASTBRANCH_5_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 4 From IP (R/W)</td>
<td>MSR_LASTBRANCH_6_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 685H, 1669</td>
<td>MSR_LASTBRANCH_7_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 5 From IP (R/W)</td>
<td>MSR_LASTBRANCH_8_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 686H, 1670</td>
<td>MSR_LASTBRANCH_9_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6 From IP (R/W)</td>
<td>MSR_LASTBRANCH_10_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 687H, 1671</td>
<td>MSR_LASTBRANCH_11_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7 From IP (R/W)</td>
<td>MSR_LASTBRANCH_12_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 688H, 1672</td>
<td>MSR_LASTBRANCH_13_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 8 From IP (R/W)</td>
<td>MSR_LASTBRANCH_14_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 689H, 1673</td>
<td>MSR_LASTBRANCH_15_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 9 From IP (R/W)</td>
<td>MSR_LASTBRANCH_16_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 68AH, 1674</td>
<td>MSR_LASTBRANCH_17_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 10 From IP (R/W)</td>
<td>MSR_LASTBRANCH_18_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 68BH, 1675</td>
<td>MSR_LASTBRANCH_19_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11 From IP (R/W)</td>
<td>MSR_LASTBRANCH_20_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 68CH, 1676</td>
<td>MSR_LASTBRANCH_21_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12 From IP (R/W)</td>
<td>MSR_LASTBRANCH_22_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 68DH, 1677</td>
<td>MSR_LASTBRANCH_23_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13 From IP (R/W)</td>
<td>MSR_LASTBRANCH_24_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 68EH, 1678</td>
<td>MSR_LASTBRANCH_25_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14 From IP (R/W)</td>
<td>MSR_LASTBRANCH_26_FROM_IP</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 68FH, 1679</td>
<td>MSR_LASTBRANCH_27_FROM_IP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 15 From IP (R/W)</td>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C0H, 1728</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 0 To IP (R/w)</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C1H, 1729</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 1 To IP (R/w)</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C2H, 1730</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 2 To IP (R/w)</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C3H, 1731</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 3 To IP (R/w)</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C4H, 1732</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 4 To IP (R/w)</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C5H, 1733</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 5 To IP (R/w)</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C6H, 1734</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6 To IP (R/w)</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C7H, 1735</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7 To IP (R/w)</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C8H, 1736</td>
<td>MSR_LASTBRANCH_8_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 8 To IP (R/w)</td>
<td>MSR_LASTBRANCH_8_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C9H, 1737</td>
<td>MSR_LASTBRANCH_9_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 9 To IP (R/w)</td>
<td>MSR_LASTBRANCH_9_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6CAH, 1738</td>
<td>MSR_LASTBRANCH_10_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 10 To IP (R/w)</td>
<td>MSR_LASTBRANCH_10_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6CBH, 1739</td>
<td>MSR_LASTBRANCH_11_TO_IP</td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11 To IP (R/w)</td>
<td>MSR_LASTBRANCH_11_TO_IP</td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 6CCH, 1740</td>
<td>MSR_LASTBRANCH_12_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6CDH, 1741</td>
<td>MSR_LASTBRANCH_13_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6CEH, 1742</td>
<td>MSR_LASTBRANCH_14_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6CFH, 1743</td>
<td>MSR_LASTBRANCH_15_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 15 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 802H, 2050</td>
<td>IA32_X2APIC_APICID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC ID Register (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 803H, 2051</td>
<td>IA32_X2APIC_VERSION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Version Register (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 808H, 2056</td>
<td>IA32_X2APIC_TPR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Task Priority Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80AH, 2058</td>
<td>IA32_X2APIC_PPR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Processor Priority Register (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80BH, 2059</td>
<td>IA32_X2APIC_EOI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC EOI Register (W/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80DH, 2061</td>
<td>IA32_X2APIC_LDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Logical Destination Register (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80FH, 2063</td>
<td>IA32_X2APIC_SIVR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Spurious Interrupt Vector Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 810H, 2064</td>
<td>IA32_X2APIC_ISR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [31:0] (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 811H, 2065</td>
<td>IA32_X2APIC_ISR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [63:32] (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 812H, 2066</td>
<td>IA32_X2APIC_ISR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [95:64] (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 813H, 2067</td>
<td>IA32_X2APIC_ISR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [127:96] (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 814H, 2068</td>
<td>IA32_X2APIC_ISR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [159:128] (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 815H, 2069</td>
<td>IA32_X2APIC_ISR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [191:160] (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSRs in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 816H, 2070</td>
<td>IA32_X2APIC_ISR6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [223:192] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 817H, 2071</td>
<td>IA32_X2APIC_ISR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC In-Service Register Bits [255:224] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 818H, 2072</td>
<td>IA32_X2APIC_TMR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [31:0] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 819H, 2073</td>
<td>IA32_X2APIC_TMR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [63:32] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 81AH, 2074</td>
<td>IA32_X2APIC_TMR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [95:64] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 81BH, 2075</td>
<td>IA32_X2APIC_TMR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [127:96] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 81CH, 2076</td>
<td>IA32_X2APIC_TMR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [159:128] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 81DH, 2077</td>
<td>IA32_X2APIC_TMR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [191:160] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 81EH, 2078</td>
<td>IA32_X2APIC_TMR6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [223:192] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 81FH, 2079</td>
<td>IA32_X2APIC_TMR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Trigger Mode Register Bits [255:224] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 820H, 2080</td>
<td>IA32_X2APIC_IRR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [31:0] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 821H, 2081</td>
<td>IA32_X2APIC_IRR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [63:32] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 822H, 2082</td>
<td>IA32_X2APIC_IRR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [95:64] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 823H, 2083</td>
<td>IA32_X2APIC_IRR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [127:96] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 824H, 2084</td>
<td>IA32_X2APIC_IRR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [159:128] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 825H, 2085</td>
<td>IA32_X2APIC_IRR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [191:160] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 826H, 2086</td>
<td>IA32_X2APIC_IRR6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [223:192] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 827H, 2087</td>
<td>IA32_X2APIC_IRR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Request Register Bits [255:224] (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 828H, 2088</td>
<td>IA32_X2APIC_ESR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-15. MSR\(\text{s}\) in Processors Based on Nehalem Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>x2APIC Error Status Register (R/W)</td>
<td>IA32_X2APIC_LVT_CMCI</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B2FH, 2095</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Corrected Machine Check Interrupt Register (R/W)</td>
<td>IA32_X2APIC_ICR</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B30H, 2096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Interrupt Command Register (R/W)</td>
<td>IA32_X2APIC_LVT_TIMER</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B32H, 2098</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Timer Interrupt Register (R/W)</td>
<td>IA32_X2APIC_LVT_THERMAL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B33H, 2099</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Thermal Sensor Interrupt Register (R/W)</td>
<td>IA32_X2APIC_LVT_PMI</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B34H, 2100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Performance Monitor Register (R/W)</td>
<td>IA32_X2APIC_LVT_LINT0</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B35H, 2101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT LINT0 Register (R/W)</td>
<td>IA32_X2APIC_LVT_LINT1</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B36H, 2102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT LINT1 Register (R/W)</td>
<td>IA32_X2APIC_LVT_ERROR</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B37H, 2103</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC LVT Error Register (R/W)</td>
<td>IA32_X2APIC_INIT_COUNT</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B38H, 2104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Initial Count Register (R/W)</td>
<td>IA32_X2APIC_CUR_COUNT</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B39H, 2105</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Current Count Register (R/O)</td>
<td>IA32_X2APIC_DIV_CONF</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B3EH, 2110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Divide Configuration Register (R/W)</td>
<td>IA32_X2APIC_SELF_IPI</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: B3FH, 2111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Self IPI Register (W/O)</td>
<td>IA32_EFER</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Feature Enables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_STAR</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>System Call Target Address (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_LSTAR</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>IA-32e Mode System Call Target Address (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FMASK</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>System Call Flag Mask (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_FS_BASE</td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
2.8.1 Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

The Intel Xeon Processor 5500 and 3400 series supports additional model-specific registers listed in Table 2-16. These MSRs also apply to the Intel Core i7 and i5 processor family with a CPUID Signature DisplayFamily_DisplayModel value of 06_1AH, 06_1EH, or 06_1FH; see Table 2-1.

### Table 2-16. Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Turbo Ratio Limit 1C (R/O)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Maximum Turbo mode ratio limit with 1 core active.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Turbo Ratio Limit 2C (R/O)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Maximum Turbo mode ratio limit with 2 cores active.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Turbo Ratio Limit 3C (R/O)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Maximum Turbo mode ratio limit with 3 cores active.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Turbo Ratio Limit 4C (R/O)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Maximum Turbo mode ratio limit with 4 cores active.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>301H, 769</td>
<td>MSR_GQ_SNOOP_MESF</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0</td>
<td>From M to S (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>1</td>
<td>From E to S (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>2</td>
<td>From S to S (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>3</td>
<td>From F to S (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>4</td>
<td>From M to I (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-16. Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>From E to I (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>From S to I (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>From F to I (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 391H, 913</td>
<td>MSR_UNCORE_PERF_GLOBAL_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.1, “Uncore Performance Monitoring Management Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 392H, 914</td>
<td>MSR_UNCORE_PERF_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.1, “Uncore Performance Monitoring Management Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 393H, 915</td>
<td>MSR_UNCORE_PERF_GLOBAL_OVF_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.1, “Uncore Performance Monitoring Management Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 394H, 916</td>
<td>MSR_UNCORE_FIXED_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.1, “Uncore Performance Monitoring Management Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 395H, 917</td>
<td>MSR_UNCORE_FIXED_CTR_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.1, “Uncore Performance Monitoring Management Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 396H, 918</td>
<td>MSR_UNCORE_ADDR_OPCODE.Match</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.3, “Uncore Address/Opcode Match MSR.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B0H, 960</td>
<td>MSR_UNCORE_PMC0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B1H, 961</td>
<td>MSR_UNCORE_PMC1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B2H, 962</td>
<td>MSR_UNCORE_PMC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B3H, 963</td>
<td>MSR_UNCORE_PMC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B4H, 964</td>
<td>MSR_UNCORE_PMC4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B5H, 965</td>
<td>MSR_UNCORE_PMC5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B6H, 966</td>
<td>MSR_UNCORE_PMC6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3B7H, 967</td>
<td>MSR_UNCORE_PMC7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3COH, 944</td>
<td>MSR_UNCORE_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3C1H, 945</td>
<td>MSR_UNCORE_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.2.2, “Uncore Performance Event Configuration Facility.”</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 3C2H, 946</td>
<td>MSR_UNCORE_PERFEVTSEL2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.8.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

The Intel Xeon Processor 7500 series supports MSRs listed in Table 2-15 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-17. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2EH.

Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td>1ADH, 429</td>
<td>Reserved. Attempt to read/write will cause #UD.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC9_CTL2</td>
<td>289H, 649</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC10_CTL2</td>
<td>28AH, 650</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC11_CTL2</td>
<td>28BH, 651</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC12_CTL2</td>
<td>28CH, 652</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC13_CTL2</td>
<td>28DH, 653</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC14_CTL2</td>
<td>28EH, 654</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC15_CTL2</td>
<td>28FH, 655</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>IA32_MC16_CTL2</td>
<td>290H, 656</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 294H, 660</td>
<td>IA32_MC20_CTL2</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 295H, 661</td>
<td>IA32_MC21_CTL2</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 394H, 816</td>
<td>MSR_W_PMON_FIXED_CTR</td>
<td>Uncore W-box perfmon fixed counter.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 395H, 817</td>
<td>MSR_W_PMON_FIXED_CTR_CTL</td>
<td>Uncore U-box perfmon fixed counter control MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 424H, 1060</td>
<td>IA32_MC9_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 425H, 1061</td>
<td>IA32_MC9_STATUS</td>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td>IA32_MC9_ADDR</td>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td>IA32_MC9_MISC</td>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td>IA32_MC10_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td>IA32_MC10_STATUS</td>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td>IA32_MC10_ADDR</td>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42BH, 1067</td>
<td>IA32_MC10_MISC</td>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42CH, 1068</td>
<td>IA32_MC11_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42DH, 1069</td>
<td>IA32_MC11_STATUS</td>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42EH, 1070</td>
<td>IA32_MC11_ADDR</td>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42FH, 1071</td>
<td>IA32_MC11_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-17.  Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
</table>
| See Section 16.3.2.4, "IA32_MCi_MISC MSRs."
Register Address: 430H, 1072   | IA32_MC12_CTL                        |                | Package |
| See Section 16.3.2.1, "IA32_MCi_CTL MSRs."
Register Address: 431H, 1073   | IA32_MC12_STATUS                     |                | Package |
| See Section 16.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 17.
Register Address: 432H, 1074   | IA32_MC12_ADDR                       |                | Package |
| See Section 16.3.2.3, "IA32_MCi_ADDR MSRs."
Register Address: 433H, 1075   | IA32_MC12_MISC                       |                | Package |
| See Section 16.3.2.1, "IA32_MCi_CTL MSRs."
Register Address: 434H, 1076   | IA32_MC13_CTL                        |                | Package |
| See Section 16.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 17.
Register Address: 435H, 1077   | IA32_MC13_STATUS                     |                | Package |
| See Section 16.3.2.3, "IA32_MCi_ADDR MSRs."
Register Address: 436H, 1078   | IA32_MC13_ADDR                       |                | Package |
| See Section 16.3.2.4, "IA32_MCi_MISC MSRs."
Register Address: 437H, 1079   | IA32_MC13_MISC                       |                | Package |
| See Section 16.3.2.1, "IA32_MCi_CTL MSRs."
Register Address: 438H, 1080   | IA32_MC14_CTL                        |                | Package |
| See Section 16.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 17.
Register Address: 439H, 1081   | IA32_MC14_STATUS                     |                | Package |
| See Section 16.3.2.3, "IA32_MCi_ADDR MSRs."
Register Address: 43AH, 1082   | IA32_MC14_ADDR                       |                | Package |
| See Section 16.3.2.4, "IA32_MCi_MISC MSRs."
Register Address: 43BH, 1083   | IA32_MC14_MISC                       |                | Package |
| See Section 16.3.2.1, "IA32_MCi_CTL MSRs."
Register Address: 43CH, 1084   | IA32_MC15_CTL                        |                | Package |
| See Section 16.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 17.
Register Address: 43DH, 1085   | IA32_MC15_STATUS                     |                | Package |
| See Section 16.3.2.3, "IA32_MCi_ADDR MSRs."
Register Address: 43EH, 1086   | IA32_MC15_ADDR                       |                | Package |
| See Section 16.3.2.4, "IA32_MCi_MISC MSRs."
Register Address: 43FH, 1087   | IA32_MC15_MISC                       |                | Package |
| See Section 16.3.2.1, "IA32_MCi_CTL MSRs."
Register Address: 440H, 1088   | IA32_MC16_CTL                        |                | Package |
| See Section 16.3.2.2, "IA32_MCi_STATUS MSRs," and Chapter 17.  
Register Address: 441H, 1089   | IA32_MC16_STATUS                     |                | Package |
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 442H, 1090</td>
<td>IA32_MC16_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 443H, 1091</td>
<td>IA32_MC16_MISC</td>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 444H, 1092</td>
<td>IA32_MC17_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 445H, 1093</td>
<td>IA32_MC17_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 446H, 1094</td>
<td>IA32_MC17_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 447H, 1095</td>
<td>IA32_MC17_MISC</td>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 448H, 1096</td>
<td>IA32_MC18_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 449H, 1097</td>
<td>IA32_MC18_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44AH, 1098</td>
<td>IA32_MC18_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44BH, 1099</td>
<td>IA32_MC18_MISC</td>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44DH, 1101</td>
<td>IA32_MC19_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44EH, 1102</td>
<td>IA32_MC19_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44FH, 1103</td>
<td>IA32_MC19_MISC</td>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 450H, 1104</td>
<td>IA32_MC20_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 451H, 1105</td>
<td>IA32_MC20_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 452H, 1106</td>
<td>IA32_MC20_ADDR</td>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 453H, 1107</td>
<td>IA32_MC20_MISC</td>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 454H, 1108</td>
<td>IA32_MC21_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs.&quot;</td>
<td>Package</td>
<td>IA32_MC21_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 455H, 1109</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs,&quot; and Chapter 17.</td>
<td>Package</td>
<td>IA32_MC21_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 456H, 1110</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td>IA32_MC21_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 457H, 1111</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C00H, 3072</td>
<td>Uncore U-box perfmon global control MSR.</td>
<td>MSR_U_PMON_GLOBAL_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C01H, 3073</td>
<td>Uncore U-box perfmon global status MSR.</td>
<td>MSR_U_PMON_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C02H, 3074</td>
<td>Uncore U-box perfmon global overflow control MSR.</td>
<td>MSR_U_PMON_GLOBAL_OVF_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C10H, 3088</td>
<td>Uncore U-box perfmon event select MSR.</td>
<td>MSR_U_PMON_EVNT_SEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C11H, 3089</td>
<td>Uncore U-box perfmon counter MSR.</td>
<td>MSR_U_PMON_CTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C20H, 3104</td>
<td>Uncore B-box 0 perfmon local box control MSR.</td>
<td>MSR_B0_PMON_BOX_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C21H, 3105</td>
<td>Uncore B-box 0 perfmon local box status MSR.</td>
<td>MSR_B0_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C22H, 3106</td>
<td>Uncore B-box 0 perfmon local box overflow control MSR.</td>
<td>MSR_B0_PMON_BOX_OVF_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C30H, 3120</td>
<td>Uncore B-box 0 perfmon event select MSR.</td>
<td>MSR_B0_PMON_EVNT_SEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C31H, 3121</td>
<td>Uncore B-box 0 perfmon counter MSR.</td>
<td>MSR_B0_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C32H, 3122</td>
<td>Uncore B-box 0 perfmon event select MSR.</td>
<td>MSR_B0_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C33H, 3123</td>
<td>Uncore B-box 0 perfmon counter MSR.</td>
<td>MSR_B0_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C34H, 3124</td>
<td>Uncore B-box 0 perfmon event select MSR.</td>
<td>MSR_B0_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C35H, 3125</td>
<td>Uncore B-box 0 perfmon counter MSR.</td>
<td>MSR_B0_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C36H, 3126</td>
<td>Uncore B-box 0 perfmon event select MSR.</td>
<td>MSR_B0_PMON_EVNT_SEL3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>C37H, 3127</td>
<td>MSR_B0_PMON_CTR3</td>
<td>Uncore B-box 0 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C40H, 3136</td>
<td>MSR_SO_PMON_BOX_CTRL</td>
<td>Uncore S-box 0 perfmon local box control MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C41H, 3137</td>
<td>MSR_SO_PMON_BOX_STATUS</td>
<td>Uncore S-box 0 perfmon local box status MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C42H, 3138</td>
<td>MSR_SO_PMON_BOX_OVF_CTRL</td>
<td>Uncore S-box 0 perfmon local box overflow control MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C50H, 3152</td>
<td>MSR_SO_PMON_EVNT_SEL0</td>
<td>Uncore S-box 0 perfmon event select MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C51H, 3153</td>
<td>MSR_SO_PMON_CTR0</td>
<td>Uncore S-box 0 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C52H, 3154</td>
<td>MSR_SO_PMON_EVNT_SEL1</td>
<td>Uncore S-box 0 perfmon event select MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C53H, 3155</td>
<td>MSR_SO_PMON_CTR1</td>
<td>Uncore S-box 0 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C54H, 3156</td>
<td>MSR_SO_PMON_EVNT_SEL2</td>
<td>Uncore S-box 0 perfmon event select MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C55H, 3157</td>
<td>MSR_SO_PMON_CTR2</td>
<td>Uncore S-box 0 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C56H, 3158</td>
<td>MSR_SO_PMON_EVNT_SEL3</td>
<td>Uncore S-box 0 perfmon event select MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C57H, 3159</td>
<td>MSR_SO_PMON_CTR3</td>
<td>Uncore S-box 0 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C60H, 3168</td>
<td>MSR_B1_PMON_BOX_CTRL</td>
<td>Uncore B-box 1 perfmon local box control MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C61H, 3169</td>
<td>MSR_B1_PMON_BOX_STATUS</td>
<td>Uncore B-box 1 perfmon local box status MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C62H, 3170</td>
<td>MSR_B1_PMON_BOX_OVF_CTRL</td>
<td>Uncore B-box 1 perfmon local box overflow control MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C70H, 3184</td>
<td>MSR_B1_PMON_EVNT_SEL0</td>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C71H, 3185</td>
<td>MSR_B1_PMON_CTR0</td>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C72H, 3186</td>
<td>MSR_B1_PMON_EVNT_SEL1</td>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>C73H, 3187</td>
<td>MSR_B1_PMON_CTR1</td>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Register Address: C74H, 3188</td>
<td>MSR_B1_PMON_EVNT_SEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td>Register Address: C75H, 3189</td>
<td>MSR_B1_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Register Address: C76H, 3190</td>
<td>MSR_B1_PMON_EVNT_SEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td>Register Address: C77H, 3191</td>
<td>MSR_B1_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Register Address: C80H, 3120</td>
<td>MSR_W_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon local box control MSR.</td>
<td>Register Address: C81H, 3121</td>
<td>MSR_W_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon local box status MSR.</td>
<td>Register Address: C82H, 3122</td>
<td>MSR_W_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon local box overflow control MSR.</td>
<td>Register Address: C90H, 3136</td>
<td>MSR_W_PMON_EVNT_SEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon event select MSR.</td>
<td>Register Address: C91H, 3137</td>
<td>MSR_W_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon counter MSR.</td>
<td>Register Address: C92H, 3138</td>
<td>MSR_W_PMON_EVNT_SEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon event select MSR.</td>
<td>Register Address: C93H, 3139</td>
<td>MSR_W_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon counter MSR.</td>
<td>Register Address: C94H, 3140</td>
<td>MSR_W_PMON_EVNT_SEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon event select MSR.</td>
<td>Register Address: C95H, 3141</td>
<td>MSR_W_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon counter MSR.</td>
<td>Register Address: C96H, 3142</td>
<td>MSR_W_PMON_EVNT_SEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon event select MSR.</td>
<td>Register Address: C97H, 3143</td>
<td>MSR_W_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore W-box perfmon counter MSR.</td>
<td>Register Address: CA0H, 3232</td>
<td>MSR_M0_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon local box control MSR.</td>
<td>Register Address: CA1H, 3233</td>
<td>MSR_M0_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon local box status MSR.</td>
<td>Register Address: CA2H, 3234</td>
<td>MSR_M0_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon local box overflow control MSR.</td>
<td></td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>Register Address: CA4H, 3236</td>
<td>MSR_M0_PMON_TIMESTAMP</td>
<td>Uncore M-box 0 perfmon time stamp unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CA5H, 3237</td>
<td>MSR_M0_PMON_DSP</td>
<td>Uncore M-box 0 perfmon DSP unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CA6H, 3238</td>
<td>MSR_M0_PMON_ISS</td>
<td>Uncore M-box 0 perfmon ISS unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CA7H, 3239</td>
<td>MSR_M0_PMON_MAP</td>
<td>Uncore M-box 0 perfmon MAP unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CA8H, 3240</td>
<td>MSR_M0_PMON_MSC_THR</td>
<td>Uncore M-box 0 perfmon MIC THR select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CA9H, 3241</td>
<td>MSR_M0_PMON_PGT</td>
<td>Uncore M-box 0 perfmon PGT unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CAAH, 3242</td>
<td>MSR_M0_PMON_PLD</td>
<td>Uncore M-box 0 perfmon PLD unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CABH, 3243</td>
<td>MSR_M0_PMON_ZDP</td>
<td>Uncore M-box 0 perfmon ZDP unit select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB0H, 3248</td>
<td>MSR_M0_PMON_EVNT_SEL0</td>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB1H, 3249</td>
<td>MSR_M0_PMON_CTR0</td>
<td>Uncore M-box 0 perfmon counter MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB2H, 3250</td>
<td>MSR_M0_PMON_EVNT_SEL1</td>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB3H, 3251</td>
<td>MSR_M0_PMON_CTR1</td>
<td>Uncore M-box 0 perfmon counter MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB4H, 3252</td>
<td>MSR_M0_PMON_EVNT_SEL2</td>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB5H, 3253</td>
<td>MSR_M0_PMON_CTR2</td>
<td>Uncore M-box 0 perfmon counter MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB6H, 3254</td>
<td>MSR_M0_PMON_EVNT_SEL3</td>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB7H, 3255</td>
<td>MSR_M0_PMON_CTR3</td>
<td>Uncore M-box 0 perfmon counter MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB8H, 3256</td>
<td>MSR_M0_PMON_EVNT_SEL4</td>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CB9H, 3257</td>
<td>MSR_M0_PMON_CTR4</td>
<td>Uncore M-box 0 perfmon counter MSR.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: CBAH, 3258</td>
<td>MSR_M0_PMON_EVNT_SEL5</td>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore M-box 0 perfmon event select MSR.</td>
<td>MSR_M0_PMON_CTR5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CBBH, 3259</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon counter MSR.</td>
<td>MSR_S1_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CCOH, 3264</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon local box control MSR.</td>
<td>MSR_S1_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CC1H, 3265</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon local box status MSR.</td>
<td>MSR_S1_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CC2H, 3266</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon local box overflow control MSR.</td>
<td>MSR_S1_PMON_EVNT_SEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD0H, 3280</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon event select MSR.</td>
<td>MSR_S1_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD1H, 3281</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon counter MSR.</td>
<td>MSR_S1_PMON_EVNT_SEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD2H, 3282</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon event select MSR.</td>
<td>MSR_S1_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD3H, 3283</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon counter MSR.</td>
<td>MSR_S1_PMON_EVNT_SEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD4H, 3284</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon event select MSR.</td>
<td>MSR_S1_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD5H, 3285</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon counter MSR.</td>
<td>MSR_S1_PMON_EVNT_SEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD6H, 3286</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon event select MSR.</td>
<td>MSR_S1_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CD7H, 3287</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon counter MSR.</td>
<td>MSR_M1_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE0H, 3296</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon local box control MSR.</td>
<td>MSR_M1_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE1H, 3297</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon local box status MSR.</td>
<td>MSR_M1_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE2H, 3298</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon local box overflow control MSR.</td>
<td>MSR_M1_PMON_TIMESTAMP</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE4H, 3300</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon time stamp unit select MSR.</td>
<td>MSR_M1_PMON_DSP</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE5H, 3301</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon DSP unit select MSR.</td>
<td>MSR_M1_PMON_ISS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE6H, 3302</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon ISS unit select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: CE7H, 3303</td>
<td>MSR_M1_PMON_MAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon MAP unit select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE8H, 3304</td>
<td>MSR_M1_PMON_MSC_THR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon MIC THR select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CE9H, 3305</td>
<td>MSR_M1_PMON_PGT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon PGT unit select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CEAH, 3306</td>
<td>MSR_M1_PMON_PLD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon PLD unit select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CEBH, 3307</td>
<td>MSR_M1_PMON_ZDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon ZDP unit select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF0H, 3312</td>
<td>MSR_M1_PMON_EVNT_SEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon event select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF1H, 3313</td>
<td>MSR_M1_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon counter MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF2H, 3314</td>
<td>MSR_M1_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon event select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF3H, 3315</td>
<td>MSR_M1_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon counter MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF4H, 3316</td>
<td>MSR_M1_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon event select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF5H, 3317</td>
<td>MSR_M1_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon counter MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF6H, 3318</td>
<td>MSR_M1_PMON_EVNT_SEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon event select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF7H, 3319</td>
<td>MSR_M1_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon counter MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF8H, 3320</td>
<td>MSR_M1_PMON_EVNT_SEL4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon event select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CF9H, 3321</td>
<td>MSR_M1_PMON_CTR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon counter MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CFAH, 3322</td>
<td>MSR_M1_PMON_EVNT_SEL5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon event select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: CFBH, 3323</td>
<td>MSR_M1_PMON_CTR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon counter MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D00H, 3328</td>
<td>MSR_C0_PMON_BOX_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon local box control MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D01H, 3329</td>
<td>MSR_C0_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------</td>
<td>----------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon local box status MSR.</td>
<td>D02H, 3330</td>
<td>MSR_CO_PMON_BOX_OVF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon local box overflow control MSR.</td>
<td>D10H, 3344</td>
<td>MSR_CO_PMON_EVNT_SEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon event select MSR.</td>
<td>D11H, 3345</td>
<td>MSR_CO_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon counter MSR.</td>
<td>D12H, 3346</td>
<td>MSR_CO_PMON_EVNT_SEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon event select MSR.</td>
<td>D13H, 3347</td>
<td>MSR_CO_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon counter MSR.</td>
<td>D14H, 3348</td>
<td>MSR_CO_PMON_EVNT_SEL2</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon event select MSR.</td>
<td>D15H, 3349</td>
<td>MSR_CO_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon counter MSR.</td>
<td>D16H, 3350</td>
<td>MSR_CO_PMON_EVNT_SEL3</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon event select MSR.</td>
<td>D17H, 3351</td>
<td>MSR_CO_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon counter MSR.</td>
<td>D18H, 3352</td>
<td>MSR_CO_PMON_EVNT_SEL4</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon event select MSR.</td>
<td>D19H, 3353</td>
<td>MSR_CO_PMON_CTR4</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 0 perfmon counter MSR.</td>
<td>D1AH, 3354</td>
<td>MSR_CO_PMON_EVNT_SEL5</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 4 perfmon local box status MSR.</td>
<td>D20H, 3360</td>
<td>MSR_C4_PMON_BOX_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 4 perfmon local box status MSR.</td>
<td>D21H, 3361</td>
<td>MSR_C4_PMON_BOX_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 4 perfmon local box overflow control MSR.</td>
<td>D22H, 3362</td>
<td>MSR_C4_PMON_BOX_OVF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 4 perfmon event select MSR.</td>
<td>D30H, 3376</td>
<td>MSR_C4_PMON_EVNT_SEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 4 perfmon event select MSR.</td>
<td>D31H, 3377</td>
<td>MSR_C4_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>D32H, 3378</td>
<td>MSR_C4_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D33H, 3379</td>
<td>MSR_C4_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D34H, 3380</td>
<td>MSR_C4_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D35H, 3381</td>
<td>MSR_C4_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D36H, 3382</td>
<td>MSR_C4_PMON_EVNT_SEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D37H, 3383</td>
<td>MSR_C4_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D38H, 3384</td>
<td>MSR_C4_PMON_EVNT_SEL4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D39H, 3385</td>
<td>MSR_C4_PMON_CTR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3AH, 3386</td>
<td>MSR_C4_PMON_EVNT_SEL5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3BH, 3387</td>
<td>MSR_C4_PMON_CTR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D40H, 3392</td>
<td>MSR_C2_PMON_BOX_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D41H, 3393</td>
<td>MSR_C2_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D42H, 3394</td>
<td>MSR_C2_PMON_BOX_OVF_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D50H, 3408</td>
<td>MSR_C2_PMON_EVNT_SEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D51H, 3409</td>
<td>MSR_C2_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D52H, 3410</td>
<td>MSR_C2_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D53H, 3411</td>
<td>MSR_C2_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D54H, 3412</td>
<td>MSR_C2_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D55H, 3413</td>
<td>MSR_C2_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Uncore C-box 2 perfmon counter MSR.</td>
<td>D56H, 3414</td>
<td>MSR_C2_PMON_EVNT_SEL3</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 perfmon event select MSR.</td>
<td>D57H, 3415</td>
<td>MSR_C2_PMON_CTR3</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 perfmon counter MSR.</td>
<td>D58H, 3416</td>
<td>MSR_C2_PMON_EVNT_SEL4</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 perfmon event select MSR.</td>
<td>D59H, 3417</td>
<td>MSR_C2_PMON_CTR4</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 perfmon counter MSR.</td>
<td>D5AH, 3418</td>
<td>MSR_C2_PMON_EVNT_SEL5</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 perfmon event select MSR.</td>
<td>D5BH, 3419</td>
<td>MSR_C2_PMON_CTR5</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon local box control MSR.</td>
<td>D60H, 3424</td>
<td>MSR_C6_PMON_BOX_CTRL</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon local box status MSR.</td>
<td>D61H, 3425</td>
<td>MSR_C6_PMON_BOX_STATUS</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon local box overflow control MSR.</td>
<td>D62H, 3426</td>
<td>MSR_C6_PMON_BOX_OVF_CTRL</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>D70H, 3440</td>
<td>MSR_C6_PMON_EVNT_SEL0</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon counter MSR.</td>
<td>D71H, 3441</td>
<td>MSR_C6_PMON_CTR0</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>D72H, 3442</td>
<td>MSR_C6_PMON_EVNT_SEL1</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon counter MSR.</td>
<td>D73H, 3443</td>
<td>MSR_C6_PMON_CTR1</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>D74H, 3444</td>
<td>MSR_C6_PMON_EVNT_SEL2</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon counter MSR.</td>
<td>D75H, 3445</td>
<td>MSR_C6_PMON_CTR2</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>D76H, 3446</td>
<td>MSR_C6_PMON_EVNT_SEL3</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>D77H, 3447</td>
<td>MSR_C6_PMON_CTR3</td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>D78H, 3448</td>
<td>MSR_C6_PMON_EVNT_SEL4</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>D79H, 3449</td>
<td>Uncore C-box 6 perfmon counter MSR.</td>
<td>MSR_C6_PMON_CTR4</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D7AH, 3450</td>
<td>Uncore C-box 6 perfmon event select MSR.</td>
<td>MSR_C6_PMON_EVNT_SEL5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D7BH, 3451</td>
<td>Uncore C-box 6 perfmon counter MSR.</td>
<td>MSR_C6_PMON_CTR5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D80H, 3456</td>
<td>Uncore C-box 1 perfmon local box control MSR.</td>
<td>MSR_C1_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D81H, 3457</td>
<td>Uncore C-box 1 perfmon local box status MSR.</td>
<td>MSR_C1_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D82H, 3458</td>
<td>Uncore C-box 1 perfmon local box overflow control MSR.</td>
<td>MSR_C1_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D90H, 3472</td>
<td>Uncore C-box 1 perfmon event select MSR.</td>
<td>MSR_C1_PMON_EVNT_SEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D91H, 3473</td>
<td>Uncore C-box 1 perfmon counter MSR.</td>
<td>MSR_C1_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D92H, 3474</td>
<td>Uncore C-box 1 perfmon event select MSR.</td>
<td>MSR_C1_PMON_EVNT_SEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D93H, 3475</td>
<td>Uncore C-box 1 perfmon counter MSR.</td>
<td>MSR_C1_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D94H, 3476</td>
<td>Uncore C-box 1 perfmon event select MSR.</td>
<td>MSR_C1_PMON_EVNT_SEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D95H, 3477</td>
<td>Uncore C-box 1 perfmon counter MSR.</td>
<td>MSR_C1_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D96H, 3478</td>
<td>Uncore C-box 1 perfmon event select MSR.</td>
<td>MSR_C1_PMON_EVNT_SEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D97H, 3479</td>
<td>Uncore C-box 1 perfmon counter MSR.</td>
<td>MSR_C1_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D98H, 3480</td>
<td>Uncore C-box 1 perfmon event select MSR.</td>
<td>MSR_C1_PMON_EVNT_SEL4</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D9AH, 3482</td>
<td>Uncore C-box 1 perfmon event select MSR.</td>
<td>MSR_C1_PMON_EVNT_SEL5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>D9BH, 3483</td>
<td>Uncore C-box 1 perfmon counter MSR.</td>
<td>MSR_C1_PMON_CTR5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>DA0H, 3488</td>
<td>Uncore C-box 5 perfmon local box control MSR.</td>
<td>MSR_C5_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
**Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-box 5 perfmon local box control MSR.</td>
<td>Register Address: DA1H, 3489</td>
<td>MSR_C5_PMON_Box_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon local box status MSR.</td>
<td>Register Address: DA2H, 3490</td>
<td>MSR_C5_PMON_Box_0VF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon local box overflow control MSR.</td>
<td>Register Address: DB0H, 3504</td>
<td>MSR_C5_PMON_EVNT_SEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DB1H, 3505</td>
<td>MSR_C5_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon counter MSR.</td>
<td>Register Address: DB2H, 3506</td>
<td>MSR_C5_PMON_EVNT_SEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DB3H, 3507</td>
<td>MSR_C5_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon counter MSR.</td>
<td>Register Address: DB4H, 3508</td>
<td>MSR_C5_PMON_EVNT_SEL2</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DB5H, 3509</td>
<td>MSR_C5_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DB6H, 3510</td>
<td>MSR_C5_PMON_EVNT_SEL3</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon counter MSR.</td>
<td>Register Address: DB7H, 3511</td>
<td>MSR_C5_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DB8H, 3512</td>
<td>MSR_C5_PMON_EVNT_SEL4</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DB9H, 3513</td>
<td>MSR_C5_PMON_CTR4</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon event select MSR.</td>
<td>Register Address: DBAH, 3514</td>
<td>MSR_C5_PMON_EVNT_SEL5</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 5 perfmon counter MSR.</td>
<td>Register Address: DCOH, 3520</td>
<td>MSR_C3_PMON_Box_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon local box control MSR.</td>
<td>Register Address: DC1H, 3521</td>
<td>MSR_C3_PMON_Box_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon local box status MSR.</td>
<td>Register Address: DC2H, 3522</td>
<td>MSR_C3_PMON_Box_0VF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon local box overflow control MSR.</td>
<td>Register Address: DD0H, 3536</td>
<td>MSR_C3_PMON_EVNT_SEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon event select MSR.</td>
<td>Register Address: DDOH, 3515</td>
<td>MSR_C3_PMON_CTR5</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: DD1H, 3537</td>
<td>MSR_C3_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD2H, 3538</td>
<td>MSR_C3_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD3H, 3539</td>
<td>MSR_C3_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD4H, 3540</td>
<td>MSR_C3_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD5H, 3541</td>
<td>MSR_C3_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD6H, 3542</td>
<td>MSR_C3_PMON_EVNT_SEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD7H, 3543</td>
<td>MSR_C3_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD8H, 3544</td>
<td>MSR_C3_PMON_EVNT_SEL4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DD9H, 3545</td>
<td>MSR_C3_PMON_CTR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DDAH, 3546</td>
<td>MSR_C3_PMON_EVNT_SEL5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DDBH, 3547</td>
<td>MSR_C3_PMON_CTR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DE0H, 3552</td>
<td>MSR_C7_PMON_BOX_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon local box control MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DE1H, 3553</td>
<td>MSR_C7_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon local box status MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DE2H, 3554</td>
<td>MSR_C7_PMON_BOX_OVF_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon local box overflow control MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DF0H, 3568</td>
<td>MSR_C7_PMON_EVNT_SEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DF1H, 3569</td>
<td>MSR_C7_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DF2H, 3570</td>
<td>MSR_C7_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DF3H, 3571</td>
<td>MSR_C7_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon counter MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DF4H, 3572</td>
<td>MSR_C7_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon event select MSR.</td>
<td>MSR_C7_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DF5H, 3573</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon counter MSR.</td>
<td>MSR_C7_PMON_EVNT_SEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DF6H, 3574</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon event select MSR.</td>
<td>MSR_C7_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DF7H, 3575</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon counter MSR.</td>
<td>MSR_C7_PMON_EVNT_SEL4</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DF8H, 3576</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon event select MSR.</td>
<td>MSR_C7_PMON_CTR4</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DF9H, 3577</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon counter MSR.</td>
<td>MSR_C7_PMON_EVNT_SEL5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DFAH, 3578</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon event select MSR.</td>
<td>MSR_C7_PMON_CTR5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DFBH, 3579</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 7 perfmon counter MSR.</td>
<td>MSR_R0_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E00H, 3584</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon local box control MSR.</td>
<td>MSR_R0_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E01H, 3585</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon local box status MSR.</td>
<td>MSR_R0_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E02H, 3586</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon local box overflow control MSR.</td>
<td>MSR_R0_PMON_IPERF0_P0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E04H, 3588</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 0 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E05H, 3589</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 1 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E06H, 3590</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 2 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E07H, 3591</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 3 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P4</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E08H, 3592</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 4 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P5</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E09H, 3593</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 5 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P6</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E0AH, 3594</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 6 select MSR.</td>
<td>MSR_R0_PMON_IPERF0_P7</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E0BH, 3595</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon IPERF0 unit Port 7 select MSR.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0CH, 3596</td>
<td>MSR_R0_PMON_QLX_P0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0DH, 3597</td>
<td>MSR_R0_PMON_QLX_P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0EH, 3598</td>
<td>MSR_R0_PMON_QLX_P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0FH, 3599</td>
<td>MSR_R0_PMON_QLX_P3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E10H, 3600</td>
<td>MSR_R0_PMON_EVNT_SEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E11H, 3601</td>
<td>MSR_R0_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E12H, 3602</td>
<td>MSR_R0_PMON_EVNT_SEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E13H, 3603</td>
<td>MSR_R0_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E14H, 3604</td>
<td>MSR_R0_PMON_EVNT_SEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E15H, 3605</td>
<td>MSR_R0_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E16H, 3606</td>
<td>MSR_R0_PMON_EVNT_SEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E17H, 3607</td>
<td>MSR_R0_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E18H, 3608</td>
<td>MSR_R0_PMON_EVNT_SEL4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E19H, 3609</td>
<td>MSR_R0_PMON_CTR4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1AH, 3610</td>
<td>MSR_R0_PMON_EVNT_SEL5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1BH, 3611</td>
<td>MSR_R0_PMON_CTR5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1CH, 3612</td>
<td>MSR_R0_PMON_EVNT_SEL6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1DH, 3613</td>
<td>MSR_R0_PMON_CTR6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1EH, 3614</td>
<td>MSR_R0_PMON_EVNT_SEL7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore R-box 0 perfmon event select MSR.</td>
<td>MSR_R0_PMON_CTR7</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E1FH, 3615</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 0 perfmon counter MSR.</td>
<td>MSR_R1_PMON_BOX_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E20H, 3616</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon local box control MSR.</td>
<td>MSR_R1_PMON_BOX_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E21H, 3617</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon local box status MSR.</td>
<td>MSR_R1_PMON_BOX_OVF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E22H, 3618</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon local box overflow control MSR.</td>
<td>MSR_R1_PMON_IPERF1_P8</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E24H, 3620</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 8 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P9</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E25H, 3621</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 9 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P10</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E26H, 3622</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 10 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P11</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E27H, 3623</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 11 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P12</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E28H, 3624</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 12 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P13</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E29H, 3625</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 13 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P14</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2AH, 3626</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 14 select MSR.</td>
<td>MSR_R1_PMON_IPERF1_P15</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2BH, 3627</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon IPERF1 unit Port 15 select MSR.</td>
<td>MSR_R1_PMON_QLX_P4</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2CH, 3628</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon QLX unit Port 4 select MSR.</td>
<td>MSR_R1_PMON_QLX_P5</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2DH, 3629</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon QLX unit Port 5 select MSR.</td>
<td>MSR_R1_PMON_QLX_P6</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2EH, 3630</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon QLX unit Port 6 select MSR.</td>
<td>MSR_R1_PMON_QLX_P7</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2FH, 3631</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon QLX unit Port 7 select MSR.</td>
<td>MSR_R1_PMON_EVNT_SEL8</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E30H, 3632</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event select MSR.</td>
<td>MSR_R1_PMON_CTR8</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E31H, 3633</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter MSR.</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: E32H, 3634</td>
<td>MSR_R1_PMON_EVNT_SEL9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E33H, 3635</td>
<td>MSR_R1_PMON_CTR9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E34H, 3636</td>
<td>MSR_R1_PMON_EVNT_SEL10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E35H, 3637</td>
<td>MSR_R1_PMON_CTR10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E36H, 3638</td>
<td>MSR_R1_PMON_EVNT_SEL11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E37H, 3639</td>
<td>MSR_R1_PMON_CTR11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E38H, 3640</td>
<td>MSR_R1_PMON_EVNT_SEL12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E39H, 3641</td>
<td>MSR_R1_PMON_CTR12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E3AH, 3642</td>
<td>MSR_R1_PMON_EVNT_SEL13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E3BH, 3643</td>
<td>MSR_R1_PMON_CTR13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E3CH, 3644</td>
<td>MSR_R1_PMON_EVNT_SEL14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E3DH, 3645</td>
<td>MSR_R1_PMON_CTR14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E3FH, 3647</td>
<td>MSR_R1_PMON_CTR15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore R-box 1 perfmon counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E45H, 3653</td>
<td>MSR_B0_PMON_MATCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore B-box 0 perfmon local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>box match MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E46H, 3654</td>
<td>MSR_B0_PMON_MASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore B-box 0 perfmon local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>box mask MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E49H, 3657</td>
<td>MSR_S0_PMON_MATCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 0 perfmon local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>box match MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E4AH, 3658</td>
<td>MSR_S0_PMON_MASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore S-box 0 perfmon local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>box mask MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E4DH, 3661</td>
<td>MSR_B1_PMON_MATCH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Intel® Xeon® Processor 5600 Series is based on Westmere microarchitecture and supports the MSR interfaces listed in Table 2-15, Table 2-16, plus additional MSRs listed in Table 2-18. These MSRs apply to the Intel Core i7, i5, and i3 processor family with a CPUID Signature DisplayFamily_DisplayModel value of 06_25H or 06_2CH; see Table 2-1.

### Table 2-17. Additional MSRs in the Intel® Xeon® Processor 7500 Series (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore B-box 1 perfmon local box match MSR.</td>
<td>MSR_B1_PMON_MASK</td>
</tr>
<tr>
<td>Register Address: E4EH, 3662</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon local box mask MSR.</td>
<td>MSR_B1_PMON_ADDR_MASK</td>
</tr>
<tr>
<td>Register Address: E54H, 3668</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon local box address match/mask config MSR.</td>
<td>MSR_M0_PMON_MM_CONFIG</td>
</tr>
<tr>
<td>Register Address: E55H, 3669</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon local box address match MSR.</td>
<td>MSR_M0_PMON_ADDR_MATCH</td>
</tr>
<tr>
<td>Register Address: E56H, 3670</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 0 perfmon local box address mask MSR.</td>
<td>MSR_M0_PMON_ADDR_MASK</td>
</tr>
<tr>
<td>Register Address: E59H, 3673</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon local box match MSR.</td>
<td>MSR_S1_PMON_MATCH</td>
</tr>
<tr>
<td>Register Address: E5AH, 3674</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore S-box 1 perfmon local box mask MSR.</td>
<td>MSR_S1_PMON_ADDR_MATCH</td>
</tr>
<tr>
<td>Register Address: E5CH, 3676</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon local box address match/mask config MSR.</td>
<td>MSR_M1_PMON_MM_CONFIG</td>
</tr>
<tr>
<td>Register Address: E5DH, 3677</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon local box address match MSR.</td>
<td>MSR_M1_PMON_ADDR_MATCH</td>
</tr>
<tr>
<td>Register Address: E5EH, 3678</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore M-box 1 perfmon local box address mask MSR.</td>
<td>MSR_UNCORE_PMC5</td>
</tr>
<tr>
<td>Register Address: 3B5H, 965</td>
<td>Package</td>
</tr>
</tbody>
</table>

See Section 20.3.1.2.2, "Uncore Performance Event Configuration Facility."

### Table 2-18. Additional MSRs Supported by Intel® Processors Based on Westmere Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 13CH, 316</td>
<td>MSR_FEATURE_CONFIG</td>
</tr>
</tbody>
</table>

AES Configuration (RW-L)
Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.

Core
2.10  MSRS IN THE INTEL® XEON® PROCESSOR E7 FAMILY BASED ON WESTMERE MICROARCHITECTURE

The Intel® Xeon® Processor E7 Family is based on the Westmere microarchitecture and supports the MSR interfaces listed in Table 2-15 (except MSR address 1ADH), Table 2-16, plus additional MSRs listed in Table 2-19. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2FH.
### Table 2-19. Additional MSRs Supported by the Intel® Xeon® Processor E7 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES Configuration (RW-L)</td>
<td>13CH, 316</td>
<td>MSR_FEATURE_CONFIG</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>1A7H, 423</td>
<td>MSR_OFFCORE_RSP_1</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Offcore Response Event Select Register (R/W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>F40H, 3904</td>
<td>MSR_C8_PMON_BOX_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 8 perfmon local box control MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1B0H, 432</td>
<td>F41H, 3905</td>
<td>MSR_C8_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 8 perfmon local box status MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F42H, 3906</td>
<td>F50H, 3920</td>
<td>MSR_C8_PMON_BOX_OVF_CTRL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 8 perfmon local box overflow control MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F41H, 3921</td>
<td>F52H, 3922</td>
<td>MSR_C8_PMON_EVNT_SEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 8 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F42H, 3923</td>
<td>F53H, 3924</td>
<td>MSR_C8_PMON_EVNT_SEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 8 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F53H, 3925</td>
<td>F54H, 3926</td>
<td>MSR_C8_PMON_EVNT_SEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-box 8 perfmon event select MSR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-19. Additional MSRs Supported by the Intel® Xeon® Processor E7 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>F57H, 3927</td>
<td>MSR_C8_PMON_CTR3</td>
<td>Uncore C-box 8 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F58H, 3928</td>
<td>MSR_C8_PMON_EVNT_SEL4</td>
<td>Uncore C-box 8 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F59H, 3929</td>
<td>MSR_C8_PMON_CTR4</td>
<td>Uncore C-box 8 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5AH, 3930</td>
<td>MSR_C8_PMON_EVNT_SEL5</td>
<td>Uncore C-box 8 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5BH, 3931</td>
<td>MSR_C8_PMON_CTR5</td>
<td>Uncore C-box 8 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC0H, 4032</td>
<td>MSR_C9_PMON_BOX_CTRL</td>
<td>Uncore C-box 9 perfmon local box control MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC1H, 4033</td>
<td>MSR_C9_PMON_BOX_STATUS</td>
<td>Uncore C-box 9 perfmon local box status MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC2H, 4034</td>
<td>MSR_C9_PMON_BOX_OVF_CTRL</td>
<td>Uncore C-box 9 perfmon local box overflow control MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD0H, 4048</td>
<td>MSR_C9_PMON_EVNT_SEL0</td>
<td>Uncore C-box 9 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD1H, 4049</td>
<td>MSR_C9_PMON_CTR0</td>
<td>Uncore C-box 9 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD2H, 4050</td>
<td>MSR_C9_PMON_EVNT_SEL1</td>
<td>Uncore C-box 9 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD3H, 4051</td>
<td>MSR_C9_PMON_CTR1</td>
<td>Uncore C-box 9 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD4H, 4052</td>
<td>MSR_C9_PMON_EVNT_SEL2</td>
<td>Uncore C-box 9 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD5H, 4053</td>
<td>MSR_C9_PMON_CTR2</td>
<td>Uncore C-box 9 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD6H, 4054</td>
<td>MSR_C9_PMON_EVNT_SEL3</td>
<td>Uncore C-box 9 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD7H, 4055</td>
<td>MSR_C9_PMON_CTR3</td>
<td>Uncore C-box 9 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD8H, 4056</td>
<td>MSR_C9_PMON_EVNT_SEL4</td>
<td>Uncore C-box 9 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD9H, 4057</td>
<td>MSR_C9_PMON_CTR4</td>
<td>Uncore C-box 9 perfmon counter MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDAH, 4058</td>
<td>MSR_C9_PMON_EVNT_SEL5</td>
<td>Uncore C-box 9 perfmon event select MSR.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MODEL-SPECIFIC REGISTERS (MSRS)

Table 2-19. Additional MSRs Supported by the Intel® Xeon® Processor E7 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0H, 0</td>
<td>IA32_P5_MC_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1H, 1</td>
<td>IA32_P5_MC_TYPE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6H, 6</td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1BH, 27</td>
<td>IA32_APIC_BASE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34H, 52</td>
<td>MSR_SMI_COUNT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.11 MSRS IN THE INTEL® PROCESSOR FAMILY BASED ON SANDY BRIDGE MICROARCHITECTURE

Table 2-20 lists model-specific registers (MSRs) that are common to the Intel® processor family based on Sandy Bridge microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH or 06_2DH; see Table 2-1. Additional MSRs specific to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH are listed in Table 2-21.

Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0H, 0</td>
<td>IA32_P5_MC_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1H, 1</td>
<td>IA32_P5_MC_TYPE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6H, 6</td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1BH, 27</td>
<td>IA32_APIC_BASE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34H, 52</td>
<td>MSR_SMI_COUNT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 31:0                          | SMI Count (R/O)                      |                                  |                |       |
|                               | Count SMI.                           |                                  |                |       |
| 63:32                         | Reserved.                            |                                  |                |       |
| 3AH, 58                       | IA32_FEATURE_CONTROL                 |                                  |                |       |

Control Features in Intel 64 Processor (R/W)

<p>| 0                             | Lock (R/WL)                          |                                  |                | Thread |
| 1                             | Enable VMX Inside SMX Operation (R/WL) |                                  |                |       |
| 2                             | Enable VMX Outside SMX Operation (R/WL) |                                  |                |       |</p>
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:8</td>
<td>SENTER Local Functions Enables (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SENTER Global Functions Enable (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 79H, 121</td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIOS Update Trigger Register (w)</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 8BH, 139</td>
<td>IA32_BIOS_SIGN_ID</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>BIOS Update Signature ID (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C1H, 193</td>
<td>IA32_PMC0</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C2H, 194</td>
<td>IA32_PMC1</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C3H, 195</td>
<td>IA32_PMC2</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C4H, 196</td>
<td>IA32_PMC3</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C5H, 197</td>
<td>IA32_PMC4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: C6H, 198</td>
<td>IA32_PMC5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: C7H, 199</td>
<td>IA32_PMC6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: C8H, 200</td>
<td>IA32_PMC7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: CEH, 206</td>
<td>MSR_PLATFORM_INFO</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Platform Information</td>
<td>Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
<td>When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### MODEL-SPECIFIC REGISTERS (MSRs)

#### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
<td>When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.</td>
<td>Package</td>
</tr>
<tr>
<td>39:30</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.</td>
<td>Package</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Register Address:</strong> E2H, 226</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C-State Configuration Control (R/W)**  
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.  

| 2:0                           | Package C-State Limit (R/W)          | Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit Note: This field cannot be used to limit package C-state to C3. | Core |
| 9:3                           | Reserved.                             |                 |       |
| 10                            | I/O MWAIT Redirection Enable (R/W)   | When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions. |       |
| 14:11                         | Reserved.                             |                 |       |
| 15                            | CFG Lock (R/W/O)                      | When set, locks bits 15:0 of this register until next reset. |       |
| 24:16                         | Reserved.                             |                 |       |
| 25                            | C3 State Auto Demotion Enable (R/w)   | When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information. |       |
| 26                            | C1 State Auto Demotion Enable (R/w)   | When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information. |       |
| 27                            | Enable C3 Undemotion (R/w)            | When set, enables undemotion from demoted C3. |       |
Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Enable C1 Undemotion (R/W)</td>
<td>When set, enables undemotion from demoted C1.</td>
<td></td>
</tr>
<tr>
<td>63:29</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E4H, 228</td>
<td>MSR_PMG_IO_CAPTURE_BASE</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>15:0</td>
<td>LVL_2 Base Address (R/W)</td>
<td>Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.</td>
<td>Core</td>
</tr>
<tr>
<td>18:16</td>
<td>C-State Range (R/W)</td>
<td>Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include. 001b - C6 is the max C-State to include. 010b - C7 is the max C-State to include.</td>
<td>Core</td>
</tr>
<tr>
<td>63:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7H, 231</td>
<td>IA32_MPERF</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Maximum Performance Frequency Clock Count (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E8H, 232</td>
<td>IA32_APERF</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Actual Performance Frequency Clock Count (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td>IA32_MTRRCAP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 13CH, 316</td>
<td>MSR_FEATURE_CONFIG</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>AES Configuration (Rw-L)</td>
<td>Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>AES Configuration (Rw-L)</td>
<td>Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.</td>
<td>Core</td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Global Machine Check Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>RIPV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.</td>
<td>Thread</td>
</tr>
<tr>
<td>1</td>
<td>EIPV</td>
<td>When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.</td>
<td>Thread</td>
</tr>
<tr>
<td>2</td>
<td>MCIP</td>
<td>When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.</td>
<td>Thread</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 186H, 390</td>
<td>IA32_PERFEVTSEL0</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 187H, 391</td>
<td>IA32_PERFEVTSEL1</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 188H, 392</td>
<td>IA32_PERFEVTSEL2</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 189H, 393</td>
<td>IA32_PERFEVTSEL3</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 18AH, 394</td>
<td>IA32_PERFEVTSEL4</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 188H, 395</td>
<td>IA32_PERFEVTSEL5</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 18CH, 396</td>
<td>IA32_PERFEVTSEL6</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 18DH, 397</td>
<td>IA32_PERFEVTSEL7</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td></td>
<td>Core</td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td></td>
<td>Current Performance State Value</td>
<td>Package</td>
</tr>
<tr>
<td>63:16</td>
<td></td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 198H, 408</td>
<td>MSR_PERF_STATUS</td>
</tr>
<tr>
<td>Performance Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:32</td>
<td></td>
<td>Core Voltage (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2^13).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 19AH, 410</td>
<td>IA32_CLOCK_MODULATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Modulation (R/W)</td>
<td>IA32_CLOCK_MODULATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td>On demand Clock Modulation Duty Cycle (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>In 6.25% increment.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>On demand Clock Modulation Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>63:5</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 19BH, 411</td>
<td>IA32_THERM_INTERRUPT</td>
</tr>
<tr>
<td>Thermal Interrupt Control (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Thermal Status (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Thermal Status Log (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>PROTCRCHOT # or FORCEPR# Status (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>PROTCRCHOT # or FORCEPR# Log (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Critical Temperature Status (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Critical Temperature Status Log (R/W/C0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Thermal Threshold #1 Status (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Thermal Threshold #1 Log (R/WCO)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Thermal Threshold #2 Status (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Thermal Threshold #2 Log (R/WCO)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Power Limitation Status (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Power Limitation Log (R/WCO)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:16</td>
<td>Digital Readout (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>26:23</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:27</td>
<td>Resolution in Degrees Celsius (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reading Valid (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MISC_ENABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 1A0H, 416

**Enable Misc. Processor Features (R/W)**

Allows a variety of processor functions to be enabled and disabled.

| 0                             | Fast-Strings Enable                 | Thread        |
| 6:1                           | Reserved                             |               |
| 7                             | Performance Monitoring Available (R)| Thread        |
| 10:8                          | Reserved                             |               |
| 11                            | Branch Trace Storage Unavailable (R/O)| Thread     |
| 12                            | Processor Event Based Sampling Unavailable (R/O) | Thread |
| 15:13                         | Reserved                             |               |
| 16                            | Enhanced Intel SpeedStep Technology Enable (R/W) | Package |
| 18                            | ENABLE MONITOR FSM (R/W)             | Thread        |
| 21:19                         | Reserved                             |               |
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Limit CPUID Maxval (R/W)</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>33:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>XD Bit Disable (R/W)</td>
<td>See Table 2-3.</td>
<td>Thread</td>
</tr>
<tr>
<td>37:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Turbo Mode Disable (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. Note: The power-on default value is used by BIOS to detect hardware support of turbo mode. If the power-on default value is 1, turbo mode is available in the processor. If the power-on default value is 0, turbo mode is not available.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:39</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A2H, 418</td>
<td>MSR_TEMPERATURE_TARGET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temperature Target</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Temperature Target (R)</td>
<td>The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.</td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A4H, 420</td>
<td>MSR_MISC_FEATURE_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous Feature Control (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>L2 Hardware Prefetcher Disable (R/W)</td>
<td>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>1</td>
<td>L2 Adjacent Cache Line Prefetcher Disable (R/W)</td>
<td>If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>2</td>
<td>DCU Hardware Prefetcher Disable (R/W)</td>
<td>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>3</td>
<td>DCU IP Prefetcher Disable (R/W)</td>
<td>If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A6H, 422</td>
<td>MSR_OFFCORE_RSP_0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offcore Response Event Select Register (R/W)</td>
<td>Offcore Response Event Select Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A7H, 423</td>
<td>MSR_OFFCORE_RSP_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offcore Response Event Select Register (R/W)</td>
<td>Offcore Response Event Select Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1AAH, 426</td>
<td>MSR_MISC_PWR_MGMT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1B0H, 432</td>
<td>IA32_ENERGY_PERF_BIAS</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 1B1H, 433</td>
<td>IA32_PACKAGE_THERM_STATUS</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 1B2H, 434</td>
<td>IA32_PACKAGE_THERM_INTERRUPT</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 1C8H, 456</td>
<td>MSR_LBR_SELECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record Filtering Select Register (R/w)</td>
<td>Last Branch Record Filtering Select Register (R/w)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>See Section 18.9.2, “Filtering of Last Branch Records.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CPL_EQ_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CPL_NEQ_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JCC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NEAR_REL_CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NEAR_IND_CALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>NEAR_RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>NEAR_IND_JMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>NEAR_REL_JMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>FAR_BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1C9H, 457</td>
<td>MSR_LASTBRANCH_TOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record Stack TOS (R/w)</td>
<td>Last Branch Record Stack TOS (R/w)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Contains an index (bits 0-3) that points to the MSR containing the most recent branch record.</td>
<td>See MSR_LASTBRANCH_0_FROM_IP (at 680H).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D9H, 473</td>
<td>IA32_DEBUGCTL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>BTS_OFF_OS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BTS_OFF_USER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>FREEZE_LBR_ON_PMI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>FREEZE_PERFMON_ON_PMI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ENABLE_UNCORE_PMI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>FREEZE_WHILE_SMM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:15</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1DDH, 477</td>
<td>MSR_LER_FROM_LIP</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Last Exception Record From Linear IP (R/W)</strong></td>
<td>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1DEH, 478</td>
<td>MSR_LER_TO_LIP</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td><strong>Last Exception Record To Linear IP (R/W)</strong></td>
<td>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1F2H, 498</td>
<td>IA32_SMRR_PHYSBASE</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1F3H, 499</td>
<td>IA32_SMRR_PHYSMASK</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1FCH, 508</td>
<td>MSR_POWER_CTL</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 200H, 512</td>
<td>IA32_MTRR_PHYSBASE0</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 201H, 513</td>
<td>IA32_MTRR_PHYSMASK0</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 202H, 514</td>
<td>IA32_MTRR_PHYSBASE1</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 203H, 515</td>
<td>IA32_MTRR_PHYSMASK1</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 204H, 516</td>
<td>IA32_MTRR_PHYSBASE2</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 205H, 517</td>
<td>IA32_MTRR_PHYSMASK2</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 206H, 518</td>
<td>IA32_MTRR_PHYSBASE3</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 207H, 519</td>
<td>IA32_MTRR_PHYSMASK3</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 208H, 520</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
## Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_PHYSMASK4</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>209H, 521</td>
<td></td>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>20AH, 522</td>
<td></td>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_PHYSMASK6</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>20BH, 523</td>
<td></td>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_PHYSBASE8</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>20CH, 524</td>
<td></td>
<td>IA32_MTRR_PHYSBASE9</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_PHYSBASE10</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>20DH, 525</td>
<td></td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>20EH, 526</td>
<td></td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>20FH, 527</td>
<td></td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>210H, 528</td>
<td></td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX4K_00000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>211H, 529</td>
<td></td>
<td>IA32_MTRR_FIX4K_80000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX4K_A0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>212H, 530</td>
<td></td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>213H, 531</td>
<td></td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td>Thread</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td>Thread</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td>Thread</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td>Thread</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td>Thread</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 280H, 640</td>
<td>IA32_MC0_CTL2</td>
<td>Core</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 281H, 641</td>
<td>IA32_MC1_CTL2</td>
<td>Core</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 282H, 642</td>
<td>IA32_MC2_CTL2</td>
<td>Core</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 283H, 643</td>
<td>IA32_MC3_CTL2</td>
<td>Core</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 284H, 644</td>
<td>IA32_MC4_CTL2</td>
<td>Core</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Always 0 (CMCI not supported).</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td>Thread</td>
<td>Default Memory Types (R/W)</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td>Thread</td>
<td>Fixed-Function Performance Counter Register 0 (R/W)</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td>Thread</td>
<td>Fixed-Function Performance Counter Register 1 (R/W)</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 30BH, 779</td>
<td>IA32_FIXED_CTR2</td>
<td>Thread</td>
<td>Fixed-Function Performance Counter Register 2 (R/W)</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td>Thread</td>
<td>See Table 2-2 and Section 18.4.1, &quot;IA32_DEBUGCTL MSR.&quot;</td>
</tr>
<tr>
<td>See Table 2-2 and Section 18.4.1, &quot;IA32_DEBUGCTL MSR.&quot;</td>
<td>See Table 2-2 and Section 18.4.1, &quot;IA32_DEBUGCTL MSR.&quot;</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>5:0</td>
<td>LBR Format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PEBS Record Format.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>PEBSaveArchRegs</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>PEBS_REC_FORMAT</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SMM_FREEZE</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>63:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38DH, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td>Fixed-Function-Counter Control Register (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2 and Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Ovf_PMC0</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ovf_PMC1</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ovf_PMC2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ovf_PMC3</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ovf_PMC4 (if CPUID.0AH:EAX[15:8] &gt; 4)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Ovf_PMC5 (if CPUID.0AH:EAX[15:8] &gt; 5)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Ovf_PMC6 (if CPUID.0AH:EAX[15:8] &gt; 6)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Ovf_PMC7 (if CPUID.0AH:EAX[15:8] &gt; 7)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Ovf_Uncore</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Ovf_BufDSSAVE</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>CondChgd</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>38FH, 911</td>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2 and Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Set 1 to enable PMC0 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Set 1 to enable PMC1 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Set 1 to enable PMC2 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Set 1 to enable PMC3 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Set 1 to enable PMC4 to count (if CPUID.0AH:EAX[15:8] &gt; 4).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Set 1 to enable PMC5 to count (if CPUID.0AH:EAX[15:8] &gt; 5).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Set 1 to enable PMC6 to count (if CPUID.0AH:EAX[15:8] &gt; 6).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Set 1 to enable PMC7 to count (if CPUID.0AH:EAX[15:8] &gt; 7).</td>
<td>Core</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to enable FixedCtr0 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to enable FixedCtr1 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to enable FixedCtr2 to count.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>63:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 390H, 912</td>
<td>IA32_PERF_GLOBAL_OVF_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2 and Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Set 1 to clear Ovf_PMC0.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Set 1 to clear Ovf_PMC1.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Set 1 to clear Ovf_PMC2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Set 1 to clear Ovf_PMC3.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] &gt; 4).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] &gt; 5).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] &gt; 6).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] &gt; 7).</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to clear Ovf_FixedCtr0.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to clear Ovf_FixedCtr1.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to clear Ovf_FixedCtr2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>60:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Set 1 to clear Ovf_Uncore.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Set 1 to clear Ovf_BufDSSAVE.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Set 1 to clear CondChgd.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.3.1.1.1, “Processor Event Based Sampling (PEBS).”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Enable PEBS on IA32_PMC0. (R/w)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable PEBS on IA32_PMC1. (R/w)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable PEBS on IA32_PMC2. (R/w)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Enable PEBS on IA32_PMC3. (R/w)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Enable Load Latency on IA32_PMC0. (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Enable Load Latency on IA32_PMC1. (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Enable Load Latency on IA32_PMC2. (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Enable Load Latency on IA32_PMC3. (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:36</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Enable Precise Store (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F6H, 1014</td>
<td>MSR_PEBS_LD_LAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>See Section 20.3.1.1.2, “Load Latency Performance Monitoring Facility.”</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>15:0</td>
<td>Minimum threshold latency value of tagged load operation that will be counted. (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:36</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F8H, 1016</td>
<td>MSR_PKG_C3_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_PKG_C3_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63:0</td>
<td>Package C3 Residency Counter (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F9H, 1017</td>
<td>MSR_PKG_C6_RESIDENCY</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>MSR_PKG_C6_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FAH, 1018</td>
<td>MSR_PKG_C7_RESIDENCY</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>MSR_PKG_C7_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C7 Residency Counter (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FCH, 1020</td>
<td>MSR_CORE_C3_RESIDENCY</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>MSR_CORE_C3_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C3 Residency Counter (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FDH, 1021</td>
<td>MSR_CORE_C6_RESIDENCY</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>MSR_CORE_C6_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C6 Residency Counter (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FEH, 1022</td>
<td>MSR_CORE_C7_RESIDENCY</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>MSR_CORE_C7_RESIDENCY</td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C7 Residency Counter (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>IA32_MC0_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 403H, 1027</td>
<td>IA32_MC0_MISC</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MC1_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MC1_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 406H, 1030</td>
<td>IA32_MC1_ADDR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 407H, 1031</td>
<td>IA32_MC1_MISC</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40BH, 1035</td>
<td>IA32_MC2_MISC</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40DH, 1037</td>
<td>IA32_MC3_STATUS</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MC3_ADDR</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40FH, 1039</td>
<td>IA32_MC3_MISC</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
<td>IA32_MC4_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCU Hardware Error (R/W)</th>
<th>When set, enables signaling of PCU hardware detected errors.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCU Controller Error (R/W)</td>
<td>When set, enables signaling of PCU controller detected errors.</td>
</tr>
<tr>
<td>PCU Firmware Error (R/W)</td>
<td>When set, enables signaling of PCU firmware detected errors.</td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 411H, 1041  IA32_MC4_STATUS

See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.

Register Address: 480H, 1152  IA32_VMX_BASIC

Reporting Register of Basic VMX Capabilities (R/O)

See Table 2-2 and Appendix A.1, “Basic VMX Information.”

Register Address: 481H, 1153  IA32_VMX_PINBASED_CTLS

Capability Reporting Register of Pin-Based VM-Execution Controls (R/O)

See Table 2-2 and Appendix A.3, “VM-Execution Controls.”

Register Address: 482H, 1154  IA32_VMX_PROCBASED_CTLS

Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)

See Appendix A.3, “VM-Execution Controls.”

Register Address: 483H, 1155  IA32_VMX_EXIT_CTLS

Capability Reporting Register of VM-Exit Controls (R/O)

See Table 2-2 and Appendix A.4, “VM-Exit Controls.”

Register Address: 484H, 1156  IA32_VMX_ENTRY_CTLS

Capability Reporting Register of VM-Entry Controls (R/O)

See Table 2-2 and Appendix A.5, “VM-Entry Controls.”

Register Address: 485H, 1157  IA32_VMX_MISC

Reporting Register of Miscellaneous VMX Capabilities (R/O)

See Table 2-2 and Appendix A.6, “Miscellaneous Data.”

Register Address: 486H, 1158  IA32_VMX_CR0_FIXED0

Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)

See Table 2-2 and Appendix A.7, “VMX-Fixed Bits in CR0.”

Register Address: 487H, 1159  IA32_VMX_CR0_FIXED1

Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)

See Table 2-2 and Appendix A.7, “VMX-Fixed Bits in CR0.”

Register Address: 488H, 1160  IA32_VMX_CR4_FIXED0

Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)

See Table 2-2 and Appendix A.8, “VMX-Fixed Bits in CR4.”

Register Address: 489H, 1161  IA32_VMX_CR4_FIXED1

Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)

See Table 2-2 and Appendix A.8, “VMX-Fixed Bits in CR4.”

Register Address: 48AH, 1162  IA32_VMX_VMCS_ENUM

Capability Reporting Register of VMCS Field Enumeration (R/O)

See Table 2-2 and Appendix A.9, “VMCS Enumeration.”

Register Address: 48BH, 1163  IA32_VMX_PROCBASED_CTLS2

Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)

See Appendix A.3, “VM-Execution Controls.”
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>48CH, 1164</td>
<td>IA32_VMX_EPT_VPID_ENUM</td>
<td>Capability Reporting Register of EPT and VPID (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>48DH, 1165</td>
<td>IA32_VMX_TRUE_PINBASED_CTLS</td>
<td>Capability Reporting Register of Pin-Based VM-Execution Flex Controls (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>48EH, 1166</td>
<td>IA32_VMX_TRUE_PROCBASED_CTLS</td>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Flex Controls (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>48FH, 1167</td>
<td>IA32_VMX_TRUE_EXIT_CTLS</td>
<td>Capability Reporting Register of VM-Exit Flex Controls (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>490H, 1168</td>
<td>IA32_VMX_TRUE_ENTRY_CTLS</td>
<td>Capability Reporting Register of VM-Entry Flex Controls (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4C1H, 1217</td>
<td>IA32_A_PMC0</td>
<td>See Table 2-2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4C2H, 1218</td>
<td>IA32_A_PMC1</td>
<td>See Table 2-2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4C3H, 1219</td>
<td>IA32_A_PMC2</td>
<td>See Table 2-2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4C4H, 1220</td>
<td>IA32_A_PMC3</td>
<td>See Table 2-2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4C5H, 1221</td>
<td>IA32_A_PMC4</td>
<td>See Table 2-2</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4C6H, 1222</td>
<td>IA32_A_PMC5</td>
<td>See Table 2-2</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>4C7H, 1223</td>
<td>IA32_A_PMC6</td>
<td>See Table 2-2</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>4C8H, 1224</td>
<td>IA32_A_PMC7</td>
<td>See Table 2-2</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>600H, 1536</td>
<td>IA32_DS_AREA</td>
<td>DS Save Area (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>606H, 1542</td>
<td>MSR_RAPL_POWER_UNIT</td>
<td>Unit Multipliers used in RAPL Interfaces (R/O)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>60AH, 1546</td>
<td>MSR_PKGC3_IRTL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-2.
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package C3 Interrupt Response Limit (R/W)</td>
<td>Interrupt Response Time Limit (R/W)</td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C3 state.</td>
<td>Package</td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C3 state.</td>
<td></td>
</tr>
<tr>
<td>12:10</td>
<td>Time Unit (R/W)</td>
<td>Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns</td>
<td></td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Valid (R/W)</td>
<td>Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 60BH, 1547 MSR_PKGC6_IRTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package C6 Interrupt Response Limit (R/W)</td>
<td>Interrupt Response Time Limit (R/W)</td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C6 state.</td>
<td>Package</td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C6 state.</td>
<td></td>
</tr>
<tr>
<td>12:10</td>
<td>Time Unit (R/W)</td>
<td>Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns</td>
<td></td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Valid (R/W)</td>
<td>Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 600H, 1549</td>
<td>MSR_PKG_C2_RESIDENCY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C2 Residency Counter (R/O)</td>
<td>Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 610H, 1552</td>
<td>MSR_PKG_POWER_LIMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKG RAPL Power Limit Control (R/W)</td>
<td>See Section 15.10.3, &quot;Package RAPL Domain.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 611H, 1553</td>
<td>MSR_PKG_ENERGY_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKG Energy Status (R/O)</td>
<td>See Section 15.10.3, &quot;Package RAPL Domain.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 614H, 1556</td>
<td>MSR_PKG_POWER_INFO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKG RAPL Parameters (R/W)</td>
<td>See Section 15.10.3, &quot;Package RAPL Domain.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 638H, 1592</td>
<td>MSR_PP0_POWER_LIMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPO RAPL Power Limit Control (R/W)</td>
<td>See Section 15.10.4, &quot;PPO/PP1 RAPL Domains.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 680H, 1664</td>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
<td>Last Branch Record 0 From IP (R/W)</td>
<td>One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also:</td>
<td>Thread</td>
</tr>
<tr>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Section 18.9.1 and record format in Section 18.4.8.1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 681H, 1665</td>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td>Last Branch Record 1 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 682H, 1666</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td>Last Branch Record 2 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 683H, 1667</td>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td>Last Branch Record 3 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 684H, 1668</td>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td>Last Branch Record 4 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 685H, 1669</td>
<td>MSR_LASTBRANCH_5_FROM_IP</td>
<td>Last Branch Record 5 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td>Thread</td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 686H, 1670</td>
<td>MSR_LASTBRANCH_6_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 687H, 1671</td>
<td>MSR_LASTBRANCH_7_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 688H, 1672</td>
<td>MSR_LASTBRANCH_8_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 8 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 689H, 1673</td>
<td>MSR_LASTBRANCH_9_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 9 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68AH, 1674</td>
<td>MSR_LASTBRANCH_10_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 10 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68BH, 1675</td>
<td>MSR_LASTBRANCH_11_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68CH, 1676</td>
<td>MSR_LASTBRANCH_12_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68DH, 1677</td>
<td>MSR_LASTBRANCH_13_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68EH, 1678</td>
<td>MSR_LASTBRANCH_14_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68FH, 1679</td>
<td>MSR_LASTBRANCH_15_FROM_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 15 From IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6COH, 1728</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 0 To IP (R/W) One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.</td>
<td>See description of MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C1H, 1729</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 1 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C2H, 1730</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Last Branch Record 2 To IP (R/W)</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C3H, 1731</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 3 To IP (R/W)</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C4H, 1732</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 4 To IP (R/W)</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C5H, 1733</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 5 To IP (R/W)</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C6H, 1734</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6 To IP (R/W)</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C7H, 1735</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7 To IP (R/W)</td>
<td>MSR_LASTBRANCH_8_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C8H, 1736</td>
<td>MSR_LASTBRANCH_8_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 8 To IP (R/W)</td>
<td>MSR_LASTBRANCH_9_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6C9H, 1737</td>
<td>MSR_LASTBRANCH_9_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 9 To IP (R/W)</td>
<td>MSR_LASTBRANCH_10_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6CAH, 1738</td>
<td>MSR_LASTBRANCH_10_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 10 To IP (R/W)</td>
<td>MSR_LASTBRANCH_11_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6CBH, 1739</td>
<td>MSR_LASTBRANCH_11_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11 To IP (R/W)</td>
<td>MSR_LASTBRANCH_12_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6CCH, 1740</td>
<td>MSR_LASTBRANCH_12_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12 To IP (R/W)</td>
<td>MSR_LASTBRANCH_13_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6CDH, 1741</td>
<td>MSR_LASTBRANCH_13_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13 To IP (R/W)</td>
<td>MSR_LASTBRANCH_14_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6CFH, 1742</td>
<td>MSR_LASTBRANCH_14_TO_IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14 To IP (R/W)</td>
<td>MSR_LASTBRANCH_15_TO_IP</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 6CFH, 1743</td>
<td>MSR_LASTBRANCH_15_TO_IP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-20. MSRs Supported by Intel® Processors Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 15 To IP (R/W)</td>
<td>IA32_TSC_DEADLINE</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6E0H, 1760</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 802H—83FH, 2050—2111</td>
<td>X2APIC MSRs</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_EFER</td>
<td>Extended Feature Enables</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_STAR</td>
<td>System Call Target Address (R/w)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_LSTAR</td>
<td>IA-32e Mode System Call Target Address (R/w)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FMASK</td>
<td>System Call Flag Mask (R/w)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_FS_BASE</td>
<td>Map of BASE Address of FS (R/w)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0101H</td>
<td>IA32_GS_BASE</td>
<td>Map of BASE Address of GS (R/w)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0102H</td>
<td>IA32_KERNEL_GS_BASE</td>
<td>Swap Target of BASE Address of GS (R/w)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0103H</td>
<td>IA32_TSC_AUX</td>
<td>AUXILIARY TSC Signature (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2 and Section 18.17.2, “IA32_TSC_AUX Register and RDTSCP Support.”</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2.11.1 MSRs in the 2nd Generation Intel® Core™ Processor Family Based on Sandy Bridge Microarchitecture

Table 2-21 and Table 2-22 list model-specific registers (MSRs) that are specific to the 2nd generation Intel® Core™ processor family based on the Sandy Bridge microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH; see Table 2-1.
### Table 2-21. MSRs Supported by the 2nd Generation Intel® Core™ Processors (Sandy Bridge Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td>Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>60CH, 1548</td>
<td>MSR_PKGC7_IRTL</td>
<td>Package C7 Interrupt Response Limit (R/W)</td>
<td>This MSR defines the budget allocated for the package to exit from a C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td>Package</td>
</tr>
<tr>
<td>63AH, 1594</td>
<td>MSR_PP0_POLICY</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register Address: 1ADH, 429**

- **MSR_TURBO_RATIO_LIMIT**
  - **Bit Description**: Maximum Ratio Limit of Turbo Mode
  - **Scope**: Package

**Register Address: 60CH, 1548**

- **MSR_PKGC7_IRTL**
  - **Package C7 Interrupt Response Limit (R/W)**
    - This MSR defines the budget allocated for the package to exit from a C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency may be applicable depending on the actual C-state the core is in.
    - Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.

**Register Address: 63AH, 1594**

- **MSR_PP0_POLICY**
  - **Register Address: 639H, 1593**
    - **MSR_PP0_ENERGY_STATUS**
      - **PP0 Energy Status (R/O)**
        - See Section 15.10.4, "PP0/PP1 RAPL Domains."
### Table 2-21. MSRs Supported by the 2nd Generation Intel® Core™ Processors (Sandy Bridge Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_PP1_POWER_LIMIT</td>
<td>Package PP0 Balance Policy (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_PP1_ENERGY_STATUS</td>
<td>Package PP1 RAPL Power Limit Control (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_PP1_POLICY</td>
<td>Package PP1 Balance Policy (R/W)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-20, Table 2-21, and Table 2-22 for MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_2AH.

### Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_UNC_PERF_GLOBAL_CTRL</td>
<td>Uncore PMU Global Control</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0</td>
<td>Slice 0 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Slice 1 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Slice 2 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Slice 3 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Slice 4 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:5</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Enable all uncore counters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Enable wake on PML.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Enable Freezing counter when overflow.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_PERF_GLOBAL_STATUS</td>
<td>Uncore PMU Main Status</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0</td>
<td>Fixed counter overflowed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>An ARB counter overflowed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A CBox counter overflowed (on any slice).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>MSR_UNC_PERF_FIXED_CTRL</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_CONFIG</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_ARB_PERFCTR0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_ARB_PERFCTR1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_ARB_PERFEVTSEL0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_ARB_PERFEVTSEL1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_PERFEVTSEL0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_PERFEVTSEL1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_PERFEVTSEL2</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_PERFEVTSEL3</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_UNIT_STATUS</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_PERFCTR0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_UNC_CBO_0_PERFCTR1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)
Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>708H, 1800</td>
<td>MSR_UNC_CBO_0_PERFCTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0, Performance Counter 2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>709H, 1801</td>
<td>MSR_UNC_CBO_0_PERFCTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0, Performance Counter 3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>710H, 1808</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Counter 0 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>711H, 1809</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Counter 1 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>712H, 1810</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Counter 2 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>713H, 1811</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Counter 3 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>715H, 1813</td>
<td>MSR_UNC_CBO_1_UNIT_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Unit Status for Counter 0-3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>716H, 1814</td>
<td>MSR_UNC_CBO_1_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>717H, 1815</td>
<td>MSR_UNC_CBO_1_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>718H, 1816</td>
<td>MSR_UNC_CBO_1_PERFCTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>719H, 1817</td>
<td>MSR_UNC_CBO_1_PERFCTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>720H, 1824</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 0 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>721H, 1825</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 1 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>722H, 1826</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 2 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>723H, 1827</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 3 Event Select MSR</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>725H, 1829</td>
<td>MSR_UNC_CBO_2_UNIT_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Unit Status for Counter 0-3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>726H, 1830</td>
<td>MSR_UNC_CBO_2_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Performance Counter 0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>727H, 1831</td>
<td>MSR_UNC_CBO_2_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Performance Counter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>728H, 1832</td>
<td>MSR_UNC_CBO_3_PERFCTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 2</td>
<td>MSR_UNC_CBO_3_PERFCTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 729H, 1833</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 3</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 730H, 1840</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 731H, 1841</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 732H, 1842</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Counter 2 Event Select MSR</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 733H, 1843</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, counter 3 Event Select MSR</td>
<td>MSR_UNC_CBO_3_UNIT_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 735H, 1845</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Unit Status for Counter 0-3</td>
<td>MSR_UNC_CBO_3_PERFCTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 736H, 1846</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 0</td>
<td>MSR_UNC_CBO_3_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 737H, 1847</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 1</td>
<td>MSR_UNC_CBO_3_PERFCTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 738H, 1848</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 2</td>
<td>MSR_UNC_CBO_3_PERFCTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 739H, 1849</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 3</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 740H, 1856</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 741H, 1857</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 742H, 1858</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 2 Event Select MSR</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 743H, 1859</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 3 Event Select MSR</td>
<td>MSR_UNC_CBO_4_UNIT_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 745H, 1861</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Unit status for Counter 0-3</td>
<td>MSR_UNC_CBO_4_PERFCTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 746H, 1862</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 0</td>
<td>MSR_UNC_CBO_4_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 747H, 1863</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 1</td>
<td>MSR_UNC_CBO_4_PERFCTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 748H, 1864</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-22. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors (Contd.)
### Model-Specific Registers (MSRs)

#### 2.11.2 MSRs in the Intel® Xeon® Processor E5 Family Based on Sandy Bridge Microarchitecture

Table 2-23 lists additional model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 Family based on Sandy Bridge microarchitecture. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2DH, and also support MSRs listed in Table 2-20 and Table 2-24.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 749H, 1865</td>
<td>MSR_UNC_CBO_4_PERFCTR3</td>
<td>Uncore C-Box 4, Performance Counter 3</td>
<td>Package</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 17FH, 383</th>
<th>MSR_ERROR_CONTROL</th>
<th>MC Bank Error Configuration (R/W)</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MemError Log Enable (R/W)</td>
<td>When set, enables IMC status bank to log additional info in bits 36:32.</td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 1ADH, 429</th>
<th>MSR_TURBO_RATIO_LIMIT</th>
<th>Maximum Ratio Limit of Turbo Mode</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>Register Address: 286H, 646</td>
<td>IA32_MC6_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 287H, 647</td>
<td>IA32_MC7_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 288H, 648</td>
<td>IA32_MC8_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 289H, 649</td>
<td>IA32_MC9_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 28AH, 650</td>
<td>IA32_MC10_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 28BH, 651</td>
<td>IA32_MC11_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 28CH, 652</td>
<td>IA32_MC12_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 28DH, 653</td>
<td>IA32_MC13_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 28EH, 654</td>
<td>IA32_MC14_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 28FH, 655</td>
<td>IA32_MC15_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 290H, 656</td>
<td>IA32_MC16_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 39CH, 924</td>
<td>MSR_PEBS_NUM_ALT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENABLE_PEBS_NUM_ALT (R/W)</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ENABLE_PEBS_NUM_ALT (R/W) Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs,” and Chapter 17.</td>
<td>IA32_MC5_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 416H, 1046</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>IA32_MC5_MISC</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 417H, 1047</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>IA32_MC5_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 418H, 1048</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>IA32_MC6_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 419H, 1049</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs,” and Chapter 17.</td>
<td>IA32_MC6_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41AH, 1050</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>IA32_MC6_MISC</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41BH, 1051</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>IA32_MC6_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41CH, 1052</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>IA32_MC7_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41DH, 1053</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs,” and Chapter 17.</td>
<td>IA32_MC7_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41EH, 1054</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>IA32_MC7_MISC</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41FH, 1055</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>IA32_MC8_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 420H, 1056</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>IA32_MC8_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 421H, 1057</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs,” and Chapter 17.</td>
<td>IA32_MC8_ADDRESS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 422H, 1058</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>IA32_MC8_MISC</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 423H, 1059</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>IA32_MC9_ADDR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 424H, 1060</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>IA32_MC9_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 425H, 1061</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs,” and Chapter 17.</td>
<td>IA32_MC9_ADDRESS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>IA32_MC9_MISC</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td>IA32_MC10_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td>IA32_MC10_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td>IA32_MC10_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42BH, 1067</td>
<td>IA32_MC10_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42CH, 1068</td>
<td>IA32_MC11_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42DH, 1069</td>
<td>IA32_MC11_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42EH, 1070</td>
<td>IA32_MC11_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42FH, 1071</td>
<td>IA32_MC11_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 430H, 1072</td>
<td>IA32_MC12_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 431H, 1073</td>
<td>IA32_MC12_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 432H, 1074</td>
<td>IA32_MC12_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 433H, 1075</td>
<td>IA32_MC12_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 434H, 1076</td>
<td>IA32_MC13_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 435H, 1077</td>
<td>IA32_MC13_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 436H, 1078</td>
<td>IA32_MC13_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 437H, 1079</td>
<td>IA32_MC13_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 438H, 1080</td>
<td>IA32_MC14_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 439H, 1081</td>
<td>IA32_MC14_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS,” and Chapter 17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43AH, 1082</td>
<td>IA32_MC14_ADDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-23. Additional MSRs Supported by the Intel® Xeon® Processors E5 Family Based on Sandy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43BH, 1083</td>
<td>IA32_MC14_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43CH, 1084</td>
<td>IA32_MC15_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43DH, 1085</td>
<td>IA32_MC15_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43EH, 1086</td>
<td>IA32_MC15_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43FH, 1087</td>
<td>IA32_MC15_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 440H, 1088</td>
<td>IA32_MC16_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 441H, 1089</td>
<td>IA32_MC16_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 442H, 1090</td>
<td>IA32_MC16_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 443H, 1091</td>
<td>IA32_MC16_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 444H, 1092</td>
<td>IA32_MC17_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 445H, 1093</td>
<td>IA32_MC17_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 446H, 1094</td>
<td>IA32_MC17_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 447H, 1095</td>
<td>IA32_MC17_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 448H, 1096</td>
<td>IA32_MC18_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 449H, 1097</td>
<td>IA32_MC18_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRS,&quot; and Chapter 17.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44AH, 1098</td>
<td>IA32_MC18_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44BH, 1099</td>
<td>IA32_MC18_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.11.3  Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 Family

Intel Xeon Processor E5 family is based on the Sandy Bridge microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24. For complete details of the uncore PMU, refer to the Intel Xeon Processor E5 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_2DH.

### Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: C08H, 3080</td>
<td>MSR_U_PMON_UCLK_FIXED_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore U-box UCLK Fixed Counter Control</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: C09H, 3081</td>
<td>MSR_U_PMON_UCLK_FIXED_CTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>Uncore U-box UCLK Fixed Counter</td>
<td>MSR_U_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C10H, 3088</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore U-box Perfmon Event Select for U-box Counter 0</td>
<td>MSR_U_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C11H, 3089</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore U-box Perfmon Event Select for U-box Counter 1</td>
<td>MSR_U_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C16H, 3094</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore U-box Perfmon Counter 0</td>
<td>MSR_U_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C17H, 3095</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore U-box Perfmon Counter 1</td>
<td>MSR_PCU_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C24H, 3108</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon for PCU-box-wide Control</td>
<td>MSR_PCU_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C30H, 3120</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Event Select for PCU Counter 0</td>
<td>MSR_PCU_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C31H, 3121</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Event Select for PCU Counter 1</td>
<td>MSR_PCU_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C32H, 3122</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Event Select for PCU Counter 2</td>
<td>MSR_PCU_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C33H, 3123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Event Select for PCU Counter 3</td>
<td>MSR_PCU_PMON_Box_FILTER</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C34H, 3124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon box-wide Filter</td>
<td>MSR_PCU_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C36H, 3126</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 0</td>
<td>MSR_PCU_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C37H, 3127</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 1</td>
<td>MSR_PCU_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C38H, 3128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 2</td>
<td>MSR_PCU_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C39H, 3129</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 3</td>
<td>MSR_C0_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D04H, 3332</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 0 Perfmon Local Box Wide Control</td>
<td>MSR_C0_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D10H, 3344</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 0</td>
<td>MSR_C0_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D11H, 3345</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 0 Perfmon Event Select for C-box 0 Counter 1</td>
<td>MSR_C0_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D12H, 3346</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------</td>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>D13H, 3347</td>
<td>MSR_CO_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D14H, 3348</td>
<td>MSR_CO_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D16H, 3350</td>
<td>MSR_CO_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D17H, 3351</td>
<td>MSR_CO_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D18H, 3352</td>
<td>MSR_CO_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D19H, 3353</td>
<td>MSR_CO_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D24H, 3364</td>
<td>MSR_C1_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D30H, 3376</td>
<td>MSR_C1_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D31H, 3377</td>
<td>MSR_C1_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D32H, 3378</td>
<td>MSR_C1_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D33H, 3379</td>
<td>MSR_C1_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D34H, 3380</td>
<td>MSR_C1_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D36H, 3382</td>
<td>MSR_C1_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D37H, 3383</td>
<td>MSR_C1_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D38H, 3384</td>
<td>MSR_C1_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D39H, 3385</td>
<td>MSR_C1_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D44H, 3396</td>
<td>MSR_C2_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D50H, 3408</td>
<td>MSR_C2_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D51H, 3409</td>
<td>MSR_C2_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 1</td>
<td>MSR_C2_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D52H, 3410</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 2</td>
<td>MSR_C2_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D53H, 3411</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Event Select for C-box 2 Counter 3</td>
<td>MSR_C2_PMON_BOX_FILTER</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D54H, 3412</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Box Wide Filter</td>
<td>MSR_C2_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D56H, 3414</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Counter 0</td>
<td>MSR_C2_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D57H, 3415</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Counter 1</td>
<td>MSR_C2_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D58H, 3416</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Counter 2</td>
<td>MSR_C2_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D59H, 3417</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 2 Perfmon Counter 3</td>
<td>MSR_C3_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D64H, 3428</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Local Box Wide Control</td>
<td>MSR_C3_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D70H, 3440</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 0</td>
<td>MSR_C3_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D71H, 3441</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 1</td>
<td>MSR_C3_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D72H, 3442</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 2</td>
<td>MSR_C3_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D73H, 3443</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Event Select for C-box 3 Counter 3</td>
<td>MSR_C3_PMON_BOX_FILTER</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D74H, 3444</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Box Wide Filter</td>
<td>MSR_C3_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D76H, 3446</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Counter 0</td>
<td>MSR_C3_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D77H, 3447</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Counter 1</td>
<td>MSR_C3_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D78H, 3448</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Counter 2</td>
<td>MSR_C3_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D79H, 3449</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-box 3 Perfmon Counter 3</td>
<td>MSR_C4_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: DB4H, 3460</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-24. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Information / Bit Fields</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>D90H, 3472</td>
<td>MSR_C4_PMON_EVNTSEL0</td>
<td>Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>D91H, 3473</td>
<td>MSR_C4_PMON_EVNTSEL1</td>
<td>Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>D92H, 3474</td>
<td>MSR_C4_PMON_EVNTSEL2</td>
<td>Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>D93H, 3475</td>
<td>MSR_C4_PMON_EVNTSEL3</td>
<td>Uncore C-box 4 Perfmon Event Select for C-box 4 Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>D95H, 3476</td>
<td>MSR_C4_PMON_BOX_FILTER</td>
<td>Uncore C-box 4 Perfmon Box Wide Filter</td>
<td>Package</td>
</tr>
<tr>
<td>D96H, 3478</td>
<td>MSR_C4_PMON_CTR0</td>
<td>Uncore C-box 4 Perfmon Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>D97H, 3479</td>
<td>MSR_C4_PMON_CTR1</td>
<td>Uncore C-box 4 Perfmon Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>D98H, 3480</td>
<td>MSR_C4_PMON_CTR2</td>
<td>Uncore C-box 4 Perfmon Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>D99H, 3481</td>
<td>MSR_C4_PMON_CTR3</td>
<td>Uncore C-box 4 Perfmon Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>DA4H, 3492</td>
<td>MSR_C5_PMON_BOX_CTL</td>
<td>Uncore C-box 5 Perfmon Local Box Wide Control</td>
<td>Package</td>
</tr>
<tr>
<td>DB0H, 3504</td>
<td>MSR_C5_PMON_EVNTSEL0</td>
<td>Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>DB1H, 3505</td>
<td>MSR_C5_PMON_EVNTSEL1</td>
<td>Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>DB2H, 3506</td>
<td>MSR_C5_PMON_EVNTSEL2</td>
<td>Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>DB3H, 3507</td>
<td>MSR_C5_PMON_EVNTSEL3</td>
<td>Uncore C-box 5 Perfmon Event Select for C-box 5 Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>DB4H, 3508</td>
<td>MSR_C5_PMON_BOX_FILTER</td>
<td>Uncore C-box 5 Perfmon Box Wide Filter</td>
<td>Package</td>
</tr>
<tr>
<td>DB6H, 3510</td>
<td>MSR_C5_PMON_CTR0</td>
<td>Uncore C-box 5 Perfmon Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>DB7H, 3511</td>
<td>MSR_C5_PMON_CTR1</td>
<td>Uncore C-box 5 Perfmon Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>DB8H, 3512</td>
<td>MSR_C5_PMON_CTR2</td>
<td>Uncore C-box 5 Perfmon Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>DB9H, 3513</td>
<td>MSR_C5_PMON_CTR3</td>
<td>Uncore C-box 5 Perfmon Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>D4H, 5324</td>
<td>MSR_C6_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDOH, 3536</td>
<td>MSR_C6_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD1H, 3537</td>
<td>MSR_C6_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD2H, 3538</td>
<td>MSR_C6_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD3H, 3539</td>
<td>MSR_C6_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DD4H, 3540</td>
<td>MSR_C6_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6H, 3542</td>
<td>MSR_C6_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7H, 3543</td>
<td>MSR_C6_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8H, 3544</td>
<td>MSR_C6_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9H, 3545</td>
<td>MSR_C6_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D64H, 3556</td>
<td>MSR_C7_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFOH, 3568</td>
<td>MSR_C7_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF1H, 3569</td>
<td>MSR_C7_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF2H, 3570</td>
<td>MSR_C7_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF3H, 3571</td>
<td>MSR_C7_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF4H, 3572</td>
<td>MSR_C7_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF6H, 3574</td>
<td>MSR_C7_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF7H, 3575</td>
<td>MSR_C7_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7H, 3576</td>
<td>MSR_C7_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7H, 3577</td>
<td>MSR_C7_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.12 MSRS IN THE 3RD GENERATION INTEL® CORE™ PROCESSOR FAMILY BASED ON IVY BRIDGE MICROARCHITECTURE

The 3rd generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v2 product family based on Ivy Bridge microarchitecture support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, and Table 2-25. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3AH.

<table>
<thead>
<tr>
<th>Register Address: CEH, 206</th>
<th>MSR_PLATFORM_INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>Platform Information</td>
<td>Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
</tr>
<tr>
<td>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</td>
<td></td>
</tr>
<tr>
<td>27:16</td>
<td>Reserved.</td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
</tr>
<tr>
<td>When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
</tr>
<tr>
<td>When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates that TDP Limit for Turbo mode is not programmable.</td>
<td></td>
</tr>
<tr>
<td>31:30</td>
<td>Reserved.</td>
</tr>
<tr>
<td>32</td>
<td>Low Power Mode Support (LPM) (R/O)</td>
</tr>
<tr>
<td>When set to 1, indicates that LPM is supported. When set to 0, indicates LPM is not supported.</td>
<td></td>
</tr>
<tr>
<td>34:33</td>
<td>Number of ConfigTDP Levels (R/O)</td>
</tr>
<tr>
<td>00: Only Base TDP level available.</td>
<td></td>
</tr>
<tr>
<td>01: One additional TDP level available.</td>
<td></td>
</tr>
<tr>
<td>02: Two additional TDP level available.</td>
<td></td>
</tr>
<tr>
<td>03: Reserved</td>
<td></td>
</tr>
<tr>
<td>39:35</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture
MODEL-SPECIFIC REGISTERS (MSRS)

Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>55:48</td>
<td>Minimum Operating Ratio (R/O)</td>
<td>Contains the minimum supported operating ratio in units of 100 MHz.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:56</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td>C-State Configuration Control (R/W)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td>When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/WO)</td>
<td>When set, locks bits 15:0 of this register until next reset.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24:16</td>
<td>C3 State Auto Demotion Enable (R/W)</td>
<td>When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>C1 State Auto Demotion Enable (R/W)</td>
<td>When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Enable C3 Undemotion (R/W)</td>
<td>When set, enables undemotion from demoted C3.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-25. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors Based on Ivy Bridge Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Enable C1 Undemotion (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, enables undemotion from demoted C1.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 63:29                         | Reserved.                             |                                   |                 |

| Register Address: 639H, 1593   | MSR_PP0_ENERGY_STATUS                | Package                           |                 |
| PP0 Energy Status (R/O)       | See Section 15.10.4, “PP0/PP1 RAPL Domains.” |

| Register Address: 648H, 1608   | MSR_CONFIG_TDP_NOMINAL               | Package                           |                 |
| Base TDP Ratio (R/O)          | Config_TDP_Base                      |                                   |                 |
|                               | Base TDP level ratio to be used for this specific processor (in units of 100 MHz). |                                   |                 |

| 63:8                          | Reserved.                             |                                   |                 |

| Register Address: 649H, 1609   | MSR_CONFIG_TDP_LEVEL1                | Package                           |                 |
| ConfigTDP Level 1 ratio and power level (R/O) |                                   |                                   |                 |

| 14:0                          | PKG_TDP_LVL1                         | Power setting for ConfigTDP Level 1. |                 |
| 15                            | Reserved.                             |                                   |                 |

| 23:16                         | Config_TDP_LVL1_Ratio                | ConfigTDP level 1 ratio to be used for this specific processor. |                 |
| 31:24                         | Reserved.                             |                                   |                 |

| 46:32                         | PKG_MAX_PWR_LVL1                     | Max Power setting allowed for ConfigTDP Level 1. |                 |
| 47                            | Reserved.                             |                                   |                 |

| 62:48                         | PKG_MIN_PWR_LVL1                     | MIN Power setting allowed for ConfigTDP Level 1. |                 |
| 63                            | Reserved.                             |                                   |                 |

| Register Address: 64AH, 1610   | MSR_CONFIG_TDP_LEVEL2                | Package                           |                 |
| ConfigTDP Level 2 ratio and power level (R/O) |                                   |                                   |                 |

| 14:0                          | PKG_TDP_LVL2                         | Power setting for ConfigTDP Level 2. |                 |
| 15                            | Reserved.                             |                                   |                 |

| 23:16                         | Config_TDP_LVL2_Ratio                | ConfigTDP level 2 ratio to be used for this specific processor. |                 |
| 31:24                         | Reserved.                             |                                   |                 |

| 46:32                         | PKG_MAX_PWR_LVL2                     | Max Power setting allowed for ConfigTDP Level 2. |                 |
| 47                            | Reserved.                             |                                   |                 |
2.12.1 MSRs in the Intel® Xeon® Processor E5 v2 Product Family Based on Ivy Bridge-E Microarchitecture

Table 2-26 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH; see Table 2-1. These processors supports the MSR interfaces listed in Table 2-20 and Table 2-26.

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>62:48</td>
<td>PKG_MIN_Pwr_LVL2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN Power setting allowed for ConfigTDP Level 2.</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 64BH, 1611</td>
<td>MSR_CONFIG_TDP_CONTROL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConfigTDP Control (R/w)</td>
<td>Package</td>
</tr>
<tr>
<td>1:0</td>
<td>TDP_LEVEL (Rw/L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System BIOS can program this field.</td>
<td></td>
</tr>
<tr>
<td>30:2</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Config_TDP_Lock (Rw/L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this bit is set, the content of this register is locked until a reset.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 64CH, 1612</td>
<td>MSR_TURBO_ACTIVATION_RATIO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConfigTDP Control (R/w)</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>MAX_NON_TURBO_RATIO (Rw/L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System BIOS can program this field.</td>
<td></td>
</tr>
<tr>
<td>30:8</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>TURBO_ACTIVATION_RATIO_Lock (Rw/L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this bit is set, the content of this register is locked until a reset.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-20, Table 2-21, and Table 2-22 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3AH.

2.12.1 MSRs in the Intel® Xeon® Processor E5 v2 Product Family Based on Ivy Bridge-E Microarchitecture

Table 2-26 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH; see Table 2-1. These processors supports the MSR interfaces listed in Table 2-20 and Table 2-26.

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4EH, 78</td>
<td>IA32_PPIN_CTL (MSR_PPIN_CTL)</td>
<td>Protected Processor Inventory Number Enable Control (R/W)</td>
</tr>
<tr>
<td>0</td>
<td>LockOut (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable_PPIN (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>4FH, 79</td>
<td>IA32_PPIN (MSR_PPIN)</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Protected Processor Inventory Number (R/O)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>CEH, 206</td>
<td>MSR_PLATFORM_INFO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Platform Information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</td>
<td></td>
</tr>
<tr>
<td>22:16</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>PPIN_CAP (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for a privileged system inventory agent to read PPIN from MSR_PPIN.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.</td>
<td></td>
</tr>
<tr>
<td>27:24</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Programmable TJ OFFSET (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify a temperature offset.</td>
<td></td>
</tr>
<tr>
<td>39:31</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.</td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
</tr>
</tbody>
</table>

C-State Configuration Control (R/W)
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
### Model-Specific Registers (MSRs)

#### Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td>When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/WO)</td>
<td>When set, locks bits 15:0 of this register until next reset.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 179H, 377 IA32_MCG_CAP

Global Machine Check Capability (R/O) Thread

<table>
<thead>
<tr>
<th>7:0</th>
<th>Count</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>MCG_CTL_P</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MCG_EXT_P</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MCP_CMCI_P</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MCG_TES_P</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>MCG_EXT_CNT</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MCG_SER_P</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>MCG_ELOG_P</td>
<td></td>
</tr>
<tr>
<td>63:27</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 17FH, 383 MSR_ERROR_CONTROL

MC Bank Error Configuration (R/W) Package

<p>| 0                              | Reserved.                             | Package |
| 1                              | MemError Log Enable (R/W)             |         |
| 1                              | When set, enables IMC status bank to log additional info in bits 36:32. |
| 63:2                           | Reserved.                             |         |</p>
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 1A2H, 418</td>
<td>MSR_TEMPERATURE_TARGET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temperature Target</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Temperature Target (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>TCC Activation Offset (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only if MSR_PLATFORM_INFO.[30] is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:28</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1AEH, 430</td>
<td>MSR_TURBO_RATIO_LIMIT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0. R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 9C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 9 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 10C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 10 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 11C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 11 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 12C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum turbo ratio limit of 12 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 285H, 645</td>
<td>IA32_MC5_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 286H, 646</td>
<td>IA32_MC6_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 287H, 647</td>
<td>IA32_MC7_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 288H, 648</td>
<td>IA32_MC8_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 289H, 649</td>
<td>IA32_MC9_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28AH, 650</td>
<td>IA32_MC10_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28BH, 651</td>
<td>IA32_MC11_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28CH, 652</td>
<td>IA32_MC12_CTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 280H, 653</td>
<td>IA32_MC13_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28EH, 654</td>
<td>IA32_MC14_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28FH, 655</td>
<td>IA32_MC15_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 290H, 656</td>
<td>IA32_MC16_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 294H, 660</td>
<td>IA32_MC20_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 295H, 661</td>
<td>IA32_MC21_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 296H, 662</td>
<td>IA32_MC22_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 297H, 663</td>
<td>IA32_MC23_CTL2IA32_MC23_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 298H, 664</td>
<td>IA32_MC24_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 299H, 665</td>
<td>IA32_MC25_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 29AH, 666</td>
<td>IA32_MC26_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 29BH, 667</td>
<td>IA32_MC27_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 29CH, 668</td>
<td>IA32_MC28_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC5 reports MC errors from the Intel QPI module.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>416H, 1046</td>
<td>IA32_MC5_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC5_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC5 reports MC errors from the Intel QPI module.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>417H, 1047</td>
<td>IA32_MC5_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC5_MISC</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC5 reports MC errors from the Intel QPI module.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>418H, 1048</td>
<td>IA32_MC6_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC6_CTL</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>419H, 1049</td>
<td>IA32_MC6_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC6_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41AH, 1050</td>
<td>IA32_MC6_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC6_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41BH, 1051</td>
<td>IA32_MC6_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC6_MISC</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41CH, 1052</td>
<td>IA32_MC7_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC7_CTL</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41DH, 1053</td>
<td>IA32_MC7_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC7_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41EH, 1054</td>
<td>IA32_MC7_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC7_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41FH, 1055</td>
<td>IA32_MC7_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC7_MISC</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>420H, 1056</td>
<td>IA32_MC8_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC8_CTL</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>421H, 1057</td>
<td>IA32_MC8_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC8_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>422H, 1058</td>
<td>IA32_MC8_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC8_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC7 and MC 8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_MC8_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC9_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC7 and MC8 report MC errors from the two home agents.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC9_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC9_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MC9_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MCl_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MCl_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MCl_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MCl_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 430H, 1072</td>
<td>IA32_MC12_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 431H, 1073</td>
<td>IA32_MC12_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 432H, 1074</td>
<td>IA32_MC12_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 433H, 1075</td>
<td>IA32_MC12_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 434H, 1076</td>
<td>IA32_MC13_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 435H, 1077</td>
<td>IA32_MC13_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 436H, 1078</td>
<td>IA32_MC13_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 437H, 1079</td>
<td>IA32_MC13_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 438H, 1080</td>
<td>IA32_MC14_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 439H, 1081</td>
<td>IA32_MC14_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43AH, 1082</td>
<td>IA32_MC14_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43BH, 1083</td>
<td>IA32_MC14_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43CH, 1084</td>
<td>IA32_MC15_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 43DH, 1085</td>
<td>IA32_MC15_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43EH, 1086</td>
<td>IA32_MC15_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 43FH, 1087</td>
<td>IA32_MC15_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 440H, 1088</td>
<td>IA32_MC16_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 441H, 1089</td>
<td>IA32_MC16_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 442H, 1090</td>
<td>IA32_MC16_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 443H, 1091</td>
<td>IA32_MC16_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 444H, 1092</td>
<td>IA32_MC17_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 445H, 1093</td>
<td>IA32_MC17_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 446H, 1094</td>
<td>IA32_MC17_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 447H, 1095</td>
<td>IA32_MC17_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC17 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 448H, 1096</td>
<td>IA32_MC18_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 449H, 1097</td>
<td>IA32_MC18_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>44AH, 1098</td>
<td>IA32_MC18_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MC18_CTL MSRs,” through Section 16.3.2.4, “IA32_MC18_MISC MSRs.”</td>
<td>Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>44BH, 1099</td>
<td>IA32_MC18_MISC</td>
<td>See Section 16.3.2.1, “IA32_MC18_CTL MSRs,” through Section 16.3.2.4, “IA32_MC18_MISC MSRs.”</td>
<td>Bank MC18 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td>See Section 16.3.2.1, “IA32_MC19_CTL MSRs,” through Section 16.3.2.4, “IA32_MC19_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>44DH, 1101</td>
<td>IA32_MC19_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MC19_CTL MSRs,” through Section 16.3.2.4, “IA32_MC19_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>44EH, 1102</td>
<td>IA32_MC20_CTL</td>
<td>See Section 16.3.2.1, “IA32_MC20_CTL MSRs,” through Section 16.3.2.4, “IA32_MC20_MISC MSRs.”</td>
<td>Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>44FH, 1103</td>
<td>IA32_MC20_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MC20_CTL MSRs,” through Section 16.3.2.4, “IA32_MC20_MISC MSRs.”</td>
<td>Bank MC20 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>450H, 1104</td>
<td>IA32_MC21_CTL</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>451H, 1105</td>
<td>IA32_MC21_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>452H, 1106</td>
<td>IA32_MC21_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>453H, 1107</td>
<td>IA32_MC21_MISC</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>454H, 1108</td>
<td>IA32_MC21_ADDRESS</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>455H, 1109</td>
<td>IA32_MC21_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
<tr>
<td>456H, 1110</td>
<td>IA32_MC21_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MC21_CTL MSRs,” through Section 16.3.2.4, “IA32_MC21_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>457H, 1111</td>
<td>IA32_MC21_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC21 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>458H, 1112</td>
<td>IA32_MC22_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>459H, 1113</td>
<td>IA32_MC22_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>45AH, 1114</td>
<td>IA32_MC22_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>45BH, 1115</td>
<td>IA32_MC22_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC22 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>45CH, 1116</td>
<td>IA32_MC23_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>45DH, 1117</td>
<td>IA32_MC23_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>45EH, 1118</td>
<td>IA32_MC23_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>45FH, 1119</td>
<td>IA32_MC23_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC23 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>460H, 1120</td>
<td>IA32_MC24_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>461H, 1121</td>
<td>IA32_MC24_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>462H, 1122</td>
<td>IA32_MC24_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>463H, 1123</td>
<td>IA32_MC24_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC24 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 464H, 1124</td>
<td>IA32_MC25_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 465H, 1125</td>
<td>IA32_MC25_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 466H, 1126</td>
<td>IA32_MC25_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 467H, 1127</td>
<td>IA32_MC26_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 468H, 1128</td>
<td>IA32_MC26_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 469H, 1129</td>
<td>IA32_MC26_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 46AH, 1130</td>
<td>IA32_MC26_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 46BH, 1131</td>
<td>IA32_MC26_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 46CH, 1132</td>
<td>IA32_MC27_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 46DH, 1133</td>
<td>IA32_MC27_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 46EH, 1134</td>
<td>IA32_MC27_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 470H, 1135</td>
<td>IA32_MC28_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.12.2 Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family

The Intel® Xeon® processor E7 v2 family (based on Ivy Bridge-E microarchitecture) with a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH supports the MSR interfaces listed in Table 2-20, Table 2-26, and Table 2-27.

Table 2-26. MSRs Supported by the Intel® Xeon® Processor E5 v2 Product Family (Ivy Bridge-E Microarchitecture)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>IA32_MC28_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
</tr>
<tr>
<td>IA32_MC28_ADDR</td>
<td>Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
</tr>
<tr>
<td>IA32_MC28_MISC</td>
<td>Bank MC28 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
</tr>
<tr>
<td>MSR_PKG_PERF_STATUS</td>
<td>Package RAPL Perf Status (R/O)</td>
</tr>
<tr>
<td>MSR_DRAM_POWER_LIMIT</td>
<td>DRAM RAPL Power Limit Control (R/W)</td>
</tr>
<tr>
<td>MSR_DRAM_ENERGY_STATUS</td>
<td>DRAM Energy Status (R/O)</td>
</tr>
<tr>
<td>MSR_DRAM_PERF_STATUS</td>
<td>DRAM Performance Throttling Status (R/O)</td>
</tr>
<tr>
<td>MSR_PP0_ENERGY_STATUS</td>
<td>PPO Energy Status (R/O)</td>
</tr>
</tbody>
</table>

See Table 2-20, for other MSR definitions applicable to Intel Xeon processor E5 v2 with a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH.

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
</tr>
</tbody>
</table>

2-250 Vol. 4
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lock (R/WL)</td>
<td>IA32_MCG_CAP</td>
<td>Thread</td>
</tr>
<tr>
<td>1</td>
<td>Enable VMX Inside SMX Operation (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable VMX Outside SMX Operation (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>SENTER Local Functions Enables (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SENTER Global Functions Enable (R/WL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MCG_CAP</td>
<td>Global Machine Check Capability (R/O)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>7:0</td>
<td>Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MCG_CTL_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MCG_EXT_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MCP_CMCL_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MCG_TES_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>MCG_EXT_CNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MCG_SER_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_MCG_STATUS</td>
<td>Global Machine Check Status (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>RIPV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>EIPV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MCIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LMCE Signaled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_TURBO_RATIO_LIMIT1</td>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 9C</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 10C</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 11C</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 12C</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH (Contd.)
### Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 13C</td>
<td>Maximum turbo ratio limit of 13 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for 14C</td>
<td>Maximum turbo ratio limit of 14 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for 15C</td>
<td>Maximum turbo ratio limit of 15 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>62:56</td>
<td>Semaphore for Turbo Ratio Limit Configuration</td>
<td>If 1, the processor uses override configuration specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Semaphore for Turbo Ratio Limit Configuration</td>
<td>If 1, the processor uses override configuration specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 29DH, 669 IA32_MC29_CTL2
See Table 2-2.

Register Address: 29EH, 670 IA32_MC30_CTL2
See Table 2-2.

Register Address: 29FH, 671 IA32_MC31_CTL2
See Table 2-2.

Register Address: 3F1H, 1009 IA32_PEB_ENABLE (MSR_PEB_ENABLE)
See Section 20.3.1.1.1, “Processor Event Based Sampling (PEBS).”

<table>
<thead>
<tr>
<th>Register Address: 41BH, 1051</th>
<th>IA32_MC6_MISC</th>
<th>Misc MAC Information of Integrated I/O (R/O)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>See Section 16.3.2.4.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 474H, 1140</th>
<th>IA32_MC29_CTL</th>
<th>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
</tr>
</tbody>
</table>

Register Address: 475H, 1141 IA32_MC29_STATUS
See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”
Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.
2.12.3  Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24 and Table 2-28. For complete detail of the uncore PMU, refer to Intel

NOTES:
1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

Table 2-27. Additional MSRs Supported by the Intel® Xeon® Processor E7 v2 Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_3EH (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 476H, 1142</td>
<td>IA32_MC29_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 477H, 1143</td>
<td>IA32_MC29_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC29 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 478H, 1144</td>
<td>IA32_MC30_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 479H, 1145</td>
<td>IA32_MC30_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 47AH, 1146</td>
<td>IA32_MC30_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 47BH, 1147</td>
<td>IA32_MC30_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC30 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 47CH, 1148</td>
<td>IA32_MC31_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 47DH, 1149</td>
<td>IA32_MC31_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC31 reports MC errors from a specific CBo (core broadcast) and its corresponding slice of L3.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-20, Table 2-26 for other MSR definitions applicable to Intel Xeon processor E7 v2 with a CPUID Signature DisplayFamily_DisplayModel value of 06_3AH.

NOTES:
1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.12.3  Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-24 and Table 2-28. For complete detail of the uncore PMU, refer to Intel
Xeon Processor E5 v2 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3EH.

Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: C00H, 3072</td>
<td>MSR_PMON_GLOBAL_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Perfmon Per-Socket Global Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C01H, 3073</td>
<td>MSR_PMON_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Perfmon Per-Socket Global Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C06H, 3078</td>
<td>MSR_PMON_GLOBAL_CONFIG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Perfmon Per-Socket Global Configuration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C15H, 3093</td>
<td>MSR_U_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore U-box Perfmon U-Box Wide Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C35H, 3125</td>
<td>MSR_PCU_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Box Wide Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D1AH, 3354</td>
<td>MSR_C0_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D3AH, 3386</td>
<td>MSR_C1_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D5AH, 3418</td>
<td>MSR_C2_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D7AH, 3450</td>
<td>MSR_C3_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: D9AH, 3482</td>
<td>MSR_C4_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DBAH, 3514</td>
<td>MSR_C5_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DDAH, 3546</td>
<td>MSR_C6_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: DFAH, 3578</td>
<td>MSR_C7_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Box Wide Filter1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E04H, 3588</td>
<td>MSR_C8_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Local Box Wide Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E10H, 3600</td>
<td>MSR_C8_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E11H, 3601</td>
<td>MSR_C8_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E12H, 3602</td>
<td>MSR_C8_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E13H, 3603</td>
<td>MSR_C8_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-28. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 3</td>
<td>MSR_C8_PMON_BOX_FILTER</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E14H, 3604</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Box Wide Filter</td>
<td>MSR_C8_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E16H, 3606</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 0</td>
<td>MSR_C8_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E17H, 3607</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 1</td>
<td>MSR_C8_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E18H, 3608</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 2</td>
<td>MSR_C8_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E19H, 3609</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 3</td>
<td>MSR_C8_PMON_BOX_FILTER1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E1AH, 3610</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Box Wide Filter1</td>
<td>MSR_C9_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E24H, 3620</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Local Box Wide Control</td>
<td>MSR_C9_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E30H, 3632</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 0</td>
<td>MSR_C9_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E31H, 3633</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 1</td>
<td>MSR_C9_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E32H, 3634</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 2</td>
<td>MSR_C9_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E33H, 3635</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-box 9 Counter 3</td>
<td>MSR_C9_PMON_BOX_FILTER</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E34H, 3636</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Box Wide Filter</td>
<td>MSR_C9_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E36H, 3638</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 0</td>
<td>MSR_C9_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E37H, 3639</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 1</td>
<td>MSR_C9_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E38H, 3640</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 2</td>
<td>MSR_C9_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E39H, 3641</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 3</td>
<td>MSR_C9_PMON_BOX_FILTER1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E3AH, 3642</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Box Wide Filter1</td>
<td>MSR_C10_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E44H, 3652</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>E50H, 3664</td>
<td>MSR_C10_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E51H, 3665</td>
<td>MSR_C10_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E52H, 3666</td>
<td>MSR_C10_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E53H, 3667</td>
<td>MSR_C10_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E54H, 3668</td>
<td>MSR_C10_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E56H, 3670</td>
<td>MSR_C10_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E57H, 3671</td>
<td>MSR_C10_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E58H, 3672</td>
<td>MSR_C10_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E59H, 3673</td>
<td>MSR_C10_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E5AH, 3674</td>
<td>MSR_C10_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E64H, 3684</td>
<td>MSR_C11_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E70H, 3696</td>
<td>MSR_C11_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E71H, 3697</td>
<td>MSR_C11_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E72H, 3698</td>
<td>MSR_C11_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E73H, 3699</td>
<td>MSR_C11_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E74H, 3700</td>
<td>MSR_C11_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E76H, 3702</td>
<td>MSR_C11_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E77H, 3703</td>
<td>MSR_C11_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E78H, 3704</td>
<td>MSR_C11_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Name (Former Register Name)</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 11 Perfmon Counter 2</td>
<td>MSR_C11_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E79H, 3705</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 11 Perfmon Counter 3</td>
<td>MSR_C11_PMON_BOX_FILTER1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E7AH, 3706</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 11 Perfmon Box Wide Filter</td>
<td>MSR_C12_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E84H, 3716</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Local Box Wide Control</td>
<td>MSR_C12_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E90H, 3728</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 0</td>
<td>MSR_C12_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E91H, 3729</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 1</td>
<td>MSR_C12_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E92H, 3730</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 2</td>
<td>MSR_C12_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E93H, 3731</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 3</td>
<td>MSR_C12_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E94H, 3732</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Box Wide Filter</td>
<td>MSR_C12_PMON_BOX_FILTER</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E96H, 3734</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 0</td>
<td>MSR_C12_PMON_CTR0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E97H, 3735</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 1</td>
<td>MSR_C12_PMON_CTR1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E98H, 3736</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 2</td>
<td>MSR_C12_PMON_CTR2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E99H, 3737</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 3</td>
<td>MSR_C12_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EA4H, 3748</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 0</td>
<td>MSR_C13_PMON_BOX_FILTER1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EB0H, 3760</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 1</td>
<td>MSR_C13_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EB1H, 3761</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 2</td>
<td>MSR_C13_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EB2H, 3762</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3</td>
<td>MSR_C13_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EB3H, 3763</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3</td>
<td>MSR_C13_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>EB4H, 3764</td>
<td>MSR_C13_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB6H, 3766</td>
<td>MSR_C13_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB7H, 3767</td>
<td>MSR_C13_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB8H, 3768</td>
<td>MSR_C13_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB9H, 3769</td>
<td>MSR_C13_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBAH, 3770</td>
<td>MSR_C14_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC4H, 3780</td>
<td>MSR_C14_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED0H, 3792</td>
<td>MSR_C14_PMON_EVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED1H, 3793</td>
<td>MSR_C14_PMON_EVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED2H, 3794</td>
<td>MSR_C14_PMON_EVTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED3H, 3795</td>
<td>MSR_C14_PMON_EVTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED4H, 3796</td>
<td>MSR_C14_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED6H, 3798</td>
<td>MSR_C14_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED7H, 3799</td>
<td>MSR_C14_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED8H, 3800</td>
<td>MSR_C14_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED9H, 3801</td>
<td>MSR_C14_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDAH, 3802</td>
<td>MSR_C14_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Local Box Wide Control</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Box Wide Filter</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Counter 2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Counter 3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uncore C-Box 14 Perfmon Box Wide Filter1</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
2.13 MSRS IN THE 4TH GENERATION INTEL® CORE™ PROCESSORS BASED ON HASWELL MICROARCHITECTURE

The 4th generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v3 product family (based on Haswell microarchitecture), with a CPUID Signature DisplayFamily_DisplayModel value of 06_3CH, 06_45H, or 06_46H, support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, and Table 2-29. For an MSR listed in Table 2-20 that also appears in Table 2-29, Table 2-29 supersedes Table 2-20.

The MSRs listed in Table 2-29 also apply to processors based on Haswell-E microarchitecture (see Section 2.14).

Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>3BH, 59</td>
<td>IA32_TSC_ADJUST</td>
<td>Per-Logical-Processor TSC ADJUST</td>
<td>Per-Logical-Processor TSC ADJUST (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEH, 206</td>
<td>MSR_PLATFORM_INFO</td>
<td>Platform Information</td>
<td>Platform Information</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27:16</td>
<td>Reserved</td>
<td>15:8 Maximum Non-Turbo Ratio</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
<td>Reserved.</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
<td>Reserved.</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:30</td>
<td>Reserved</td>
<td>32 Low Power Mode Support (LPM) (R/O)</td>
<td>Low Power Mode Support (LPM) (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set to 1, indicates that LPM is supported. When set to 0, indicates LPM is not supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34:33</td>
<td>Number of ConfigTDP Levels (R/O)</td>
<td>39:35 Reserved.</td>
<td>Number of ConfigTDP Levels (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>00: Only Base TDP level available.</td>
<td></td>
<td>00: Only Base TDP level available.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>01: One additional TDP level available.</td>
<td></td>
<td>01: One additional TDP level available.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>02: Two additional TDP level available.</td>
<td></td>
<td>02: Two additional TDP level available.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>03: Reserved.</td>
<td></td>
<td>03: Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>55:48 Minimum Operating Ratio</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.</td>
<td></td>
<td>This is the minimum supported operating ratio in units of 100 MHz.</td>
<td>Package</td>
</tr>
<tr>
<td>63:56</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_PERFEVTSEL0</td>
<td>Performance Event Select for Counter 0 (R/W) Supports all fields described in Table 2-2 and the fields below.</td>
<td>IN_TX: See Section 20.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.</td>
<td>Thread</td>
</tr>
<tr>
<td>IA32_PERFEVTSEL1</td>
<td>Performance Event Select for Counter 1 (R/W) Supports all fields described in Table 2-2 and the fields below.</td>
<td>IN_TX: See Section 20.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.</td>
<td>Thread</td>
</tr>
<tr>
<td>IA32_PERFEVTSEL2</td>
<td>Performance Event Select for Counter 2 (R/W) Supports all fields described in Table 2-2 and the fields below.</td>
<td>IN_TX: See Section 20.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.</td>
<td>Thread</td>
</tr>
<tr>
<td>IA32_PERFEVTSEL3</td>
<td>Performance Event Select for Counter 3 (R/W) Supports all fields described in Table 2-2 and the fields below.</td>
<td>IN_TX: See Section 20.3.6.5.1. When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.</td>
<td>Thread</td>
</tr>
<tr>
<td>MSR_LBR_SELECT</td>
<td>Last Branch Record Filtering Select Register (R/w)</td>
<td>CPL_EQ_0, CPL_NEQ_0, JCC, NEAR_REL_CALL, NEAR_IND_CALL, NEAR_RET, NEAR_IND_JMP, NEAR_REL_JMP</td>
<td>Thread</td>
</tr>
</tbody>
</table>

When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.

When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.

When IN_TXCP=1 & IN_TX=1 and in sampling, a spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large "sample-after" value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.

When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.

When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results.
### Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>FAR_BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>EN_CALL_STACK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D9H, 473</td>
<td>IA32_DEBUGCTL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Debug Control (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LBR: Last Branch Record</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BTF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TR: Branch Trace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BTS: Log Branch Trace Message to BTS Buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BTINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>BTS_OFF_OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BTS_OFF_USER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>FREEZE_LBR_ON_PMI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>FREEZE_PERFMON_ON_PMI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ENABLE_UNCORE_PMI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>FREEZE_WHILE_SMM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>RTM_DEBUG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 491H, 1169</td>
<td>IA32_VMX_VMFUNC</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Function Controls (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 608H, 1548</td>
<td>MSR_PKGC_IRTL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Package C6/C7 Interrupt Response Limit 1 (R/W)</td>
<td>This MSR defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to a C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>Interrupt Response Time Limit (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:10</td>
<td>Time Unit (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Valid (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 60CH, 1548</td>
<td>MSR_PKG_C_IRTL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Package C6/C7 Interrupt Response Limit 2 (R/W)</strong></td>
<td>Package C6/C7 Interrupt Response Limit 2 (R/W)</td>
<td>Defines the interrupt response time limit used by the processor to manage a transition to a package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to a C6 or C7 state.</td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>Interrupt response time limit (R/W)</td>
<td>Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.</td>
<td></td>
</tr>
<tr>
<td>12:10</td>
<td>Time Unit (R/W)</td>
<td>Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-20 for supported time unit encodings.</td>
<td></td>
</tr>
<tr>
<td>14:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Valid (R/W)</td>
<td>Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-state management.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 613H, 1555</td>
<td>MSR_PKG_PERF_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PKG Perf Status (R/O)</strong></td>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 619H, 1561</td>
<td>MSR_DRAM_ENERGY_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DRAM Energy Status (R/O)</strong></td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 618H, 1563</td>
<td>MSR_DRAM_PERF_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DRAM Performance Throttling Status (R/O)</strong></td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 648H, 1608</td>
<td>MSR_CONFIG_TDP_NOMINAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Base TDP Ratio (R/O)</strong></td>
<td>Config_TDP_Base</td>
<td>Base TDP level ratio to be used for this specific processor (in units of 100 MHz).</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Config_TDP_Base</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 649H, 1609</td>
<td>MSR_CONFIG_TDP_LEVEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ConfigTDP Level 1 Ratio and Power Level (R/O)</strong></td>
<td>Configuration of the TDP level 1 ( CONFIG_TDP_LVL1)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>14:0</td>
<td>PKG_TDP_LVL1</td>
<td>Power setting for ConfigTDP Level 1.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Config_TDP_LVL1_Ratio</td>
<td>ConfigTDP level 1 ratio to be used for this specific processor.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-29. Additional MSRs Supported by Processors Based on the Haswell and Haswell-E Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>Reserved.</td>
<td>PKG_MAX_PWR_LVL1</td>
<td>Max Power setting allowed for ConfigTDP Level 1.</td>
<td></td>
</tr>
<tr>
<td>46:32</td>
<td></td>
<td>PKG_MIN_PWR_LVL1</td>
<td>MIN Power setting allowed for ConfigTDP Level 1.</td>
<td></td>
</tr>
<tr>
<td>62:47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64AH, 1610</td>
<td>MSR_CONFIG_TDP_LEVEL2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Level 2 Ratio and Power Level (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:0</td>
<td>PKG_TDP_LVL2</td>
<td>Power setting for ConfigTDP Level 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Config_TDP_LVL2_Ratio</td>
<td>ConfigTDP level 2 ratio to be used for this specific processor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46:32</td>
<td>PKG_MAX_PWR_LVL2</td>
<td>Max Power setting allowed for ConfigTDP Level 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:47</td>
<td>PKG_MIN_PWR_LVL2</td>
<td>MIN Power setting allowed for ConfigTDP Level 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64BH, 1611</td>
<td>MSR_CONFIG_TDP_CONTROL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Control (R/W)</td>
<td>Package</td>
<td>TDP_LEVEL (Rw/L)</td>
<td>System BIOS can program this field.</td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:2</td>
<td></td>
<td>Config_TDP_Lock (Rw/L)</td>
<td>When this bit is set, the content of this register is locked until a reset.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64CH, 1612</td>
<td>MSR_TURBO_ACTIVATION_RATIO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Control (R/w)</td>
<td>Package</td>
<td>MAX_NON_TURBO_RATIO (Rw/L)</td>
<td>System BIOS can program this field.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:8</td>
<td></td>
<td>TURBO_ACTIVATION_RATIO_Lock (Rw/L)</td>
<td>When this bit is set, the content of this register is locked until a reset.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C80H, 3200</td>
<td>IA32_DEBUG_INTERFACE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Silicon Debug Feature Control (R/W)</td>
<td>Package</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.13.1 MSRs in the 4th Generation Intel® Core™ Processor Family Based on Haswell Microarchitecture

Table 2-30 lists model-specific registers (MSRs) that are specific to the 4th generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200 v3 product family (based on Haswell microarchitecture). These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3CH, 06_45H, or 06_46H; see Table 2-1.

**Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td>C-State Configuration Control (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td></td>
<td>Package C-State Limit (R/W)</td>
<td></td>
</tr>
<tr>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The following C-state code name encodings are supported:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000b: C0/C1 (no package C-state support)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001b: C2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010b: C3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011b: C6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100b: C7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101b: C7s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package C states C7 are not available to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3CH.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:4</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>CFG Lock (R/W/O)</td>
<td></td>
</tr>
<tr>
<td>24:16</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>C3 State Auto Demotion Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>C1 State Auto Demotion Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>Enable C3 Undemotion (R/W)</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>Enable C1 Undemotion (R/W)</td>
<td></td>
</tr>
<tr>
<td>63:29</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17DH, 381</td>
<td>MSR_SMM_MCA_CAP</td>
<td>Enhanced SMM Capabilities (SMM-RO)</td>
<td>Thread</td>
</tr>
<tr>
<td>Reports SMM capability Enhancement. Accessible only while in SMM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57:0</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td></td>
<td>SMM_Code_Access_Chk (SMM-RO)</td>
<td></td>
</tr>
<tr>
<td>If set to 1, indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>59 Long_Flow_Indication (SMM-RO)</td>
<td>If set to 1, indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:60</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C Maximum turbo ratio limit of 1 core active.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 2C Maximum turbo ratio limit of 2 core active.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 3C Maximum turbo ratio limit of 3 core active.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 4C Maximum turbo ratio limit of 4 core active.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PMU Global Control</td>
<td>MSR_UNC_PERF_GLOBAL_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Core 0 select.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>1</td>
<td>Core 1 select.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>2</td>
<td>Core 2 select.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>3</td>
<td>Core 3 select.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>18:4</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>29</td>
<td>Enable all uncore counters.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>30</td>
<td>Enable wake on PMI.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>31</td>
<td>Enable Freezing counter when overflow.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore PMU Main Status</td>
<td>MSR_UNC_PERF_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Fixed counter overflowed.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>1</td>
<td>An ARB counter overflowed.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>2</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>3</td>
<td>A CBox counter overflowed (on any slice).</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Fixed Counter Control (R/W)</td>
<td>MSR_UNC_PERF_FIXED_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:0</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>20</td>
<td>Enable overflow propagation.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>21</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>22</td>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td>Enable counting.</td>
<td></td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>395H, 917</td>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td>Uncore Fixed Counter</td>
<td></td>
</tr>
<tr>
<td>47:0</td>
<td>Current count.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>396H, 918</td>
<td>MSR_UNC_CBO_CONFIG</td>
<td>Uncore C-Box Configuration Information (R/O)</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Encoded number of C-Box, derive value by “-1”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3B0H, 946</td>
<td>MSR_UNC_ARB_PERFCTR0</td>
<td>Uncore Arb Unit, Performance Counter 0</td>
<td></td>
</tr>
<tr>
<td>3B1H, 947</td>
<td>MSR_UNC_ARB_PERFCTR1</td>
<td>Uncore Arb Unit, Performance Counter 1</td>
<td></td>
</tr>
<tr>
<td>3B2H, 944</td>
<td>MSR_UNC_ARB_PERFEVTSEL0</td>
<td>Uncore Arb Unit, Counter 0 Event Select MSR</td>
<td></td>
</tr>
<tr>
<td>3B3H, 945</td>
<td>MSR_UNC_ARB_PERFEVTSEL1</td>
<td>Uncore Arb Unit, Counter 1 Event Select MSR</td>
<td></td>
</tr>
<tr>
<td>391H, 913</td>
<td>MSR_UNC_PERF_GLOBAL_CTRL</td>
<td>Uncore PMU Global Control</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Core 0 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Core 1 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Core 2 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Core 3 select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Enable all uncore counters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Enable wake on PMI.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Enable Freezing counter when overflow.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>395H, 917</td>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td>Uncore Fixed Counter</td>
<td></td>
</tr>
<tr>
<td>47:0</td>
<td>Current count.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3B3H, 945</td>
<td>MSR_UNC_ARB_PERFEVTSEL1</td>
<td>Uncore Arb Unit, Counter 1 Event Select MSR</td>
<td></td>
</tr>
<tr>
<td>4E0H, 1248</td>
<td>MSR_SMM_FEATURE_CONTROL</td>
<td>Enhanced SMM Feature Control (SMM-Rw)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reports SMM capability Enhancement. Accessible only while in SMM.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lock (SMM-RWO)</td>
<td>When set to ‘1’ locks this register from further changes.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SMM_Code_Chk_En (SMM-RW)</td>
<td>This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to ‘0’ (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to ‘1’ any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.</td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_SMM_DELAYED</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-1:0</td>
<td>LOG_PROC_STATE (SMM-RO)</td>
<td>Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.</td>
<td></td>
</tr>
<tr>
<td>63:N</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_SMM_BLOCKED</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Reports the blocked state of all logical processors in the package. Available only while in SMM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-1:0</td>
<td>LOG_PROC_STATE (SMM-RO)</td>
<td>Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is OFFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.</td>
<td></td>
</tr>
<tr>
<td>63:N</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_RAPLPOWER_UNIT</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Unit Multipliers Used in RAPL Interfaces (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Power Units</td>
<td>See Section 15.10.1, “RAPL Interfaces.”</td>
<td>Package</td>
</tr>
<tr>
<td>7:4</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>

Register Address: 4E2H, 1250

Register Address: 4E3H, 1251

Register Address: 606H, 1542
Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:8</td>
<td>Energy Status Units</td>
<td>Energy related information (in Joules) is based on the multiplier, $1/2^{\text{ESU}}$; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).</td>
<td>Package</td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>19:16</td>
<td>Time Units</td>
<td>See Section 15.10.1, &quot;RAPL Interfaces.&quot;</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 639H, 1593</td>
<td>MSR_PP0_ENERGY_STATUS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PP0 Energy Status (R/O)</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Register Address: 640H, 1600</td>
<td>MSR_PP1_POWER_LIMIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PP1 RAPL Power Limit Control (R/W)</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Register Address: 641H, 1601</td>
<td>MSR_PP1_ENERGY_STATUS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PP1 Energy Status (R/O)</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Register Address: 642H, 1602</td>
<td>MSR_PP1_POLICY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PP1 Balance Policy (R/W)</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Register Address: 690H, 1680</td>
<td>MSR_CORE_PERF_LIMIT_REASONS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicator of Frequency Clipping in Processor Cores (R/W)</td>
<td>Indicator of Frequency Clipping in Processor Cores (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>(Frequency refers to processor core frequency.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td>When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.</td>
<td>Package</td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal event.</td>
<td>Package</td>
</tr>
<tr>
<td>3:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Graphics Driver Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to Processor Graphics driver override.</td>
<td>Package</td>
</tr>
<tr>
<td>5</td>
<td>Autonomous Utilization-Based Frequency Control Status (R0)</td>
<td>When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.</td>
<td>Package</td>
</tr>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.</td>
<td>Package</td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>8</td>
<td>Electrical Design Point Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Core Power Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to domain-level power limiting.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Package-Level Power Limiting PL1 Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to package-level power limiting PL1.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Package-Level PL2 Power Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to package-level power limiting PL2.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Max Turbo Limit Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to multi-core turbo limits.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Turbo Transition Attenuation Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.</td>
<td></td>
</tr>
<tr>
<td>15:14</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>19:18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Graphics Driver Log</td>
<td>When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Autonomous Utilization-Based Frequency Control Log</td>
<td>When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>VR Therm Alert Log</td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Reserved.</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Electrical Design Point Log</td>
<td>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Core Power Limiting Log</td>
<td>When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Package-Level PL1 Power Limiting Log</td>
<td>When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Package-Level PL2 Power Limiting Log</td>
<td>When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Max Turbo Limit Log</td>
<td>When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Turbo Transition Attenuation Log</td>
<td>When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

**Register Address: 6B0H, 1712**

**MSR_GRAPHICS_PERF_LIMIT_REASONS**

**Indicator of Frequency Clipping in the Processor Graphics (R/W)**
(Frequency refers to processor graphics frequency.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PROCHOT Status (R0) When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.</td>
<td>Package</td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0) When set, frequency is reduced below the operating system request due to a thermal event.</td>
<td></td>
</tr>
<tr>
<td>3:2</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Graphics Driver Status (R0) When set, frequency is reduced below the operating system request due to Processor Graphics driver override.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Autonomous Utilization-Based Frequency Control Status (R0) When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Electrical Design Point Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).</td>
</tr>
<tr>
<td>9</td>
<td>Graphics Power Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to domain-level power limiting.</td>
</tr>
<tr>
<td>10</td>
<td>Package-Level Power Limiting PL1 Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to package-level power limiting PL1.</td>
</tr>
<tr>
<td>11</td>
<td>Package-Level PL2 Power Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to package-level power limiting PL2.</td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
</tr>
<tr>
<td>19:18</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Graphics Driver Log</td>
<td>When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
</tr>
<tr>
<td>21</td>
<td>Autonomous Utilization-Based Frequency Control Log</td>
<td>When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
</tr>
<tr>
<td>22</td>
<td>VR Therm Alert Log</td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
</tr>
<tr>
<td>23</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>24</td>
<td>Electrical Design Point Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Core Power Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Package-Level PL1 Power Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Package-Level PL2 Power Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Max Turbo Limit Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Turbo Transition Attenuation Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 6B1H, 1713
MSR_RING_PERF_LIMIT_REASONS

Indicator of Frequency Clipping in the Ring Interconnect (R/W)
(Frequency refers to ring interconnect in the uncore.)

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to a thermal event.</td>
<td></td>
</tr>
<tr>
<td>5:2</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Electrical Design Point Status (R0)</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name (Former Register Name)</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Package-Level Power Limiting PL1 Status (R0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>request due to package-level power limiting PL1.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Package-Level PL2 Power Limiting Status (R0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, frequency is reduced below the operating system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>request due to package-level power limiting PL2.</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the PROCHOT Status bit has asserted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Thermal Status bit has asserted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>19:18</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Graphics Driver Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Graphics Driver Status bit has</td>
<td></td>
</tr>
<tr>
<td></td>
<td>asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Autonomous Utilization-Based Frequency Control Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has</td>
<td></td>
</tr>
<tr>
<td></td>
<td>asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>VR Therm Alert Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the VR Therm Alert Status bit has</td>
<td></td>
</tr>
<tr>
<td></td>
<td>asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Electrical Design Point Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the EDP Status bit has asserted since the log bit was</td>
<td></td>
</tr>
<tr>
<td></td>
<td>last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Core Power Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was</td>
<td></td>
</tr>
<tr>
<td></td>
<td>last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Package-Level PL1 Power Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-30. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Package-Level PL2 Power Limiting Log</td>
<td>When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td>Package</td>
</tr>
<tr>
<td>28</td>
<td>Max Turbo Limit Log</td>
<td>When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td>Package</td>
</tr>
<tr>
<td>29</td>
<td>Turbo Transition Attenuation Log</td>
<td>When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td>Package</td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 700H, 1792</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 701H, 1793</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 706H, 1798</td>
<td>MSR_UNC_CBO_0_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 707H, 1799</td>
<td>MSR_UNC_CBO_0_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 710H, 1808</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 711H, 1809</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 716H, 1814</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 717H, 1815</td>
<td>MSR_UNC_CBO_2_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 720H, 1824</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Uncore C-Box 0, Counter 0 Event Select MSR

Register Address: 700H, 1792

 MSR_UNC_CBO_0_PERFEVTSEL0

Uncore C-Box 0, Counter 0 Event Select MSR

Register Address: 701H, 1793

 MSR_UNC_CBO_0_PERFEVTSEL1

Uncore C-Box 0, Counter 1 Event Select MSR

Register Address: 706H, 1798

 MSR_UNC_CBO_0_PERFCTR0

Uncore C-Box 0, Performance Counter 0

Register Address: 707H, 1799

 MSR_UNC_CBO_0_PERFCTR1

Uncore C-Box 0, Performance Counter 1

Register Address: 710H, 1808

 MSR_UNC_CBO_1_PERFEVTSEL0

Uncore C-Box 1, Counter 0 Event Select MSR

Register Address: 711H, 1809

 MSR_UNC_CBO_1_PERFEVTSEL1

Uncore C-Box 1, Counter 1 Event Select MSR

Register Address: 716H, 1814

 MSR_UNC_CBO_2_PERFEVTSEL0

Uncore C-Box 0, Performance Counter 0

Register Address: 717H, 1815

 MSR_UNC_CBO_2_PERFCTR1

Uncore C-Box 0, Performance Counter 1

Register Address: 720H, 1824

 MSR_UNC_CBO_3_PERFEVTSEL0

Uncore C-Box 2, Counter 0 Event Select MSR

Register Address: 721H, 1825

 MSR_UNC_CBO_2_PERFEVTSEL1

Uncore C-Box 2, Counter 1 Event Select MSR

Register Address: 726H, 1830

 MSR_UNC_CBO_2_PERFCTR0

Uncore C-Box 2, Performance Counter 0

Register Address: 727H, 1831

 MSR_UNC_CBO_2_PERFCTR1

Uncore C-Box 2, Performance Counter 1

Register Address: 730H, 1840

 MSR_UNC_CBO_3_PERFEVTSEL0

Uncore C-Box 3, Counter 0 Event Select MSR
2.13.2 Additional Residency MSRs Supported in 4th Generation Intel® Core™ Processors

The 4th generation Intel® Core™ processor family (based on Haswell microarchitecture) with a CPUID Signature DisplayFamily_DisplayModel value of 06_45H supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-30, and Table 2-31.

See Table 2-20, Table 2-21, Table 2-22, Table 2-25, and Table 2-29 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 063CH or 06_46H.

Table 2-31. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_45H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td>C-State Configuration Control (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3:0 Package C-State Limit (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The following C-state code name encodings are supported:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000b: C0/C1 (no package C-state support)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001b: C2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010b: C3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011b: C6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100b: C7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101b: C7s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110b: C8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111b: C9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000b: C10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9:4 Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 I/O MWAIT Redirection Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>14:11 Reserved.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 CFG Lock (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>24:16 Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
The Intel® Xeon® processor E5 v3 family and the Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_3F). These processors support the MSR interfaces listed in Table 2-20, Table 2-29, and Table 2-32.

Table 2-20, Table 2-21, Table 2-22, Table 2-29, and Table 2-30 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_45H.

### 2.14 MSRS IN THE INTEL® XEON® PROCESSOR E5 V3 AND E7 V3 PRODUCT FAMILY

The Intel® Xeon® processor E5 v3 family and the Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_3F). These processors support the MSR interfaces listed in Table 2-20, Table 2-29, and Table 2-32.

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>35H, 53</td>
<td>MSR_CORE_THREAD_COUNT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**MODEL-SPECIFIC REGISTERS (MSRS)**

**Configured State of Enabled Processor Core Count and Logical Processor Count (R/O)**

- After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package.
- Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.

<table>
<thead>
<tr>
<th>Register Address: 53H, 83</th>
<th>MSR_THREAD_ID_INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>15:0</td>
<td>THREAD_COUNT (R/O)</td>
</tr>
<tr>
<td>The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.</td>
<td></td>
</tr>
<tr>
<td>31:16</td>
<td>Core_COUNT (R/O)</td>
</tr>
<tr>
<td>The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.</td>
<td></td>
</tr>
</tbody>
</table>

| 63:32 | Reserved. |

**A Hardware Assigned ID for the Logical Processor (R/O)**

<table>
<thead>
<tr>
<th>Register Address: E2H, 226</th>
<th>MSR_PKG_CST_CONFIG_CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7:0</td>
<td>Logical_Processor_ID (R/O)</td>
</tr>
<tr>
<td>An implementation-specific numerical value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.</td>
<td></td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

**C-State Configuration Control (R/W)**

**Note:** C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.


| 2:0 | Package C-State Limit (R/W) |
| Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. |
| The following C-state code name encodings are supported: |
| 000b: C0/C1 (no package C-state support) |
| 001b: C2 |
| 010b: C6 (non-retention) |
| 011b: C6 (retention) |
| 111b: No Package C state limits. All C states supported by the processor are available. |
| 9:3 | Reserved. |
| 10 | I/O MWAIT Redirection Enable (R/W) |
| 14:11 | Reserved. |
| 15 | CFG Lock (R/W/O) |
| 24:16 | Reserved. |
| 25 | C3 State Auto Demotion Enable (R/W) |
| 26 | C1 State Auto Demotion Enable (R/W) |

Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:0</td>
<td>THREAD_COUNT (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>31:16</td>
<td>Core_COUNT (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Logical_Processor_ID (R/O)</td>
<td>Thread</td>
</tr>
<tr>
<td>63:8</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Core</td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/W/O)</td>
<td></td>
</tr>
<tr>
<td>24:16</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>C3 State Auto Demotion Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>C1 State Auto Demotion Enable (R/W)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>28</td>
<td>Enable C3 Undemotion (R/W)</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>Enable C1 Undemotion (R/W)</td>
<td></td>
</tr>
<tr>
<td>63:31</td>
<td>Reserved.</td>
<td>Package C State Demotion Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>63:31 Reserved.</td>
<td>Package C State Undemotion Enable (R/W)</td>
<td></td>
</tr>
<tr>
<td>IA32_MCG_CAP</td>
<td>179H, 377</td>
<td>Register Address: 179H, 377</td>
<td></td>
</tr>
<tr>
<td>IA32_MCG_CAP</td>
<td>377</td>
<td>IA32_MCG_CAP</td>
<td></td>
</tr>
<tr>
<td>Global Machine Check Capability (R/O)</td>
<td>Count</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>MCG_CTL_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MCG_EXT_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MCP_CMCI_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MCG_TES_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>15:12 Reserved.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>MCG_EXT_CNT</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MCG_SER_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>MCG_EM_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>MCG_ELOG_P</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>63:27</td>
<td>Reserved.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>MSR_SMM_MCA_CAP</td>
<td>17DH, 381</td>
<td>MSR_SMM_MCA_CAP</td>
<td>Thread</td>
</tr>
<tr>
<td>Enhanced SMM Capabilities (SMM-RO)</td>
<td>Thread</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Reports SMM capability Enhancement. Accessible only while in SMM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57:0</td>
<td>SMM_Code_Access_Chk (SMM-RO)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>SMM_Code_Access_Chk (SMM-RO)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>SMM_Code_Access_Chk (SMM-RO)</td>
<td>If set to 1, indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>Long_Flow_Indication (SMM-RO)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Long_Flow_Indication (SMM-RO)</td>
<td>If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>63:60</td>
<td>Reserved.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>MSR_ERROR_CONTROL</td>
<td>17FH, 383</td>
<td>MSR_ERROR_CONTROL</td>
<td>Thread</td>
</tr>
<tr>
<td>MC Bank Error Configuration (R/W)</td>
<td>Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MemError Log Enable (R/W)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MemError Log Enable (R/W)</td>
<td>When set, enables IMC status bank to log additional info in bits 36:32.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td>1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td>Package</td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>R/O if MSR_PLATFORM_INFO[28] = 0, and R/W if MSR_PLATFORM_INFO[28] = 1.</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C</td>
<td>Maximum turbo ratio limit of 1 core active.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Maximum Ratio Limit for 2C</td>
<td>Maximum turbo ratio limit of 2 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 3C</td>
<td>Maximum turbo ratio limit of 3 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 4C</td>
<td>Maximum turbo ratio limit of 4 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 5C</td>
<td>Maximum turbo ratio limit of 5 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 6C</td>
<td>Maximum turbo ratio limit of 6 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for 7C</td>
<td>Maximum turbo ratio limit of 7 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for 8C</td>
<td>Maximum turbo ratio limit of 8 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>63:56</td>
<td>MSR_TURBO_RATIO_LIMIT1</td>
<td>Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 9C</td>
<td>Maximum turbo ratio limit of 9 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 10C</td>
<td>Maximum turbo ratio limit of 10 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 11C</td>
<td>Maximum turbo ratio limit of 11 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 12C</td>
<td>Maximum turbo ratio limit of 12 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 13C</td>
<td>Maximum turbo ratio limit of 13 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for 14C</td>
<td>Maximum turbo ratio limit of 14 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for 15C</td>
<td>Maximum turbo ratio limit of 15 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>63:56</td>
<td>MSR_TURBO_RATIO_LIMIT2</td>
<td>Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 17C</td>
<td>Maximum turbo ratio limit of 17 core active.</td>
<td>Package</td>
</tr>
</tbody>
</table>

Register Address: 1AEH, 430

Register Address: 1AFH, 431
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 18C</td>
<td>Maximum turbo ratio limit of 18 core active.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:16</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Semaphore for Turbo Ratio Limit Configuration</td>
<td>If 1, the processor uses override configuration specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1, and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 416H, 1046</td>
<td>IA32_MC5_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 417H, 1047</td>
<td>IA32_MC5_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 418H, 1048</td>
<td>IA32_MC6_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 419H, 1049</td>
<td>IA32_MC6_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41AH, 1050</td>
<td>IA32_MC6_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41BH, 1051</td>
<td>IA32_MC6_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41CH, 1052</td>
<td>IA32_MC7_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41DH, 1053</td>
<td>IA32_MC7_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41EH, 1054</td>
<td>IA32_MC7_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41FH, 1055</td>
<td>IA32_MC7_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 420H, 1056</td>
<td>IA32_MC8_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the home agent HA 1.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 421H, 1057</td>
<td>IA32_MC8_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the home agent HA 1.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 422H, 1058</td>
<td>IA32_MC8_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the home agent HA 1.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 423H, 1059</td>
<td>IA32_MC8_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the home agent HA 1.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 424H, 1060</td>
<td>IA32_MC9_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 425H, 1061</td>
<td>IA32_MC9_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td>IA32_MC9_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td>IA32_MC9_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td>IA32_MC10_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td>IA32_MC10_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td>IA32_MC10_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42BH, 1067</td>
<td>IA32_MC10_MISC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42CH, 1068</td>
<td>IA32_MC11_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42DH, 1069</td>
<td>IA32_MC11_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42EH, 1070</td>
<td>IA32_MC11_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42FH, 1071</td>
<td>IA32_MC11_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 430H, 1072</td>
<td>IA32_MC12_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 431H, 1073</td>
<td>IA32_MC12_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 432H, 1074</td>
<td>IA32_MC12_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 433H, 1075</td>
<td>IA32_MC12_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 434H, 1076</td>
<td>IA32_MC13_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 435H, 1077</td>
<td>IA32_MC13_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 436H, 1078</td>
<td>IA32_MC13_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 437H, 1079</td>
<td>IA32_MC13_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 438H, 1080</td>
<td>IA32_MC14_CTL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 439H, 1081</td>
<td>IA32_MC14_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43AH, 1082</td>
<td>IA32_MC14_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43BH, 1083</td>
<td>IA32_MC14_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43CH, 1084</td>
<td>IA32_MC15_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43DH, 1085</td>
<td>IA32_MC15_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43EH, 1086</td>
<td>IA32_MC15_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43FH, 1087</td>
<td>IA32_MC15_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 440H, 1088</td>
<td>IA32_MC16_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 441H, 1089</td>
<td>IA32_MC16_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 442H, 1090</td>
<td>IA32_MC16_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 443H, 1091</td>
<td>IA32_MC16_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 444H, 1092</td>
<td>IA32_MC17_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 445H, 1093</td>
<td>IA32_MC17_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 446H, 1094</td>
<td>IA32_MC17_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 447H, 1095</td>
<td>IA32_MC17_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 448H, 1096</td>
<td>IA32_MC18_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 449H, 1097</td>
<td>IA32_MC18_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 44AH, 1098</td>
<td>IA32_MC18_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 44BH, 1099</td>
<td>IA32_MC18_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 44DH, 1101</td>
<td>IA32_MC19_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 44EH, 1102</td>
<td>IA32_MC19_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 44FH, 1103</td>
<td>IA32_MC19_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 450H, 1104</td>
<td>IA32_MCI_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 451H, 1105</td>
<td>IA32_MCI_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 452H, 1106</td>
<td>IA32_MCI_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 453H, 1107</td>
<td>IA32_MCI_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 454H, 1108</td>
<td>IA32_MCI21_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC21 reports MC errors from the Intel QPI 2 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 455H, 1109</td>
<td>IA32_MCI21_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC21 reports MC errors from the Intel QPI 2 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 456H, 1110</td>
<td>IA32_MCI21_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC21 reports MC errors from the Intel QPI 2 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 457H, 1111</td>
<td>IA32_MCI21_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC21 reports MC errors from the Intel QPI 2 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 606H, 1542</td>
<td>MSR_RAPL_POWER_UNIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Multipliers Used in RAPL Interfaces (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Power Units</td>
<td>See Section 15.10.1, “RAPL Interfaces.”</td>
<td>Package</td>
</tr>
<tr>
<td>7:4</td>
<td>Reserved.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>12:8</td>
<td>Energy Status Units</td>
<td>Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).</td>
<td>Package</td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>19:16</td>
<td>Time Units</td>
<td>See Section 15.10.1, “RAPL Interfaces.”</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vol. 4 2-285
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 618H, 1560</td>
<td>MSR_DRAM_POWER_LIMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM RAPL Power Limit Control (R/w)</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 619H, 1561</td>
<td>MSR_DRAM_ENERGY_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>DRAM Energy Status (R/O)</td>
<td>Energy Consumed by DRAM devices.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 618H, 1563</td>
<td>MSR_DRAM_PERF_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>DRAM Performance Throttling Status (R/O)</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 61CH, 1564</td>
<td>MSR_DRAM_POWER_INFO</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>DRAM RAPL Parameters (R/w)</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 61EH, 1566</td>
<td>MSR_PCIE_PLL_RATIO</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Configuration of PCIE PLL Relative to BCLK(R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>PCIE Ratio (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>00b: Use 5:5 mapping for 100MHz operation (default).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01b: Use 5:4 mapping for 125MHz operation.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10b: Use 5:3 mapping for 166MHz operation.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11b: Use 5:2 mapping for 250MHz operation.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PLL Select (R/w)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>If 1, use configured setting of PCIE Ratio.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LONG RESET (R/W)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>If 1, wait an additional time-out before re-locking Gen2/Gen3 PLLs.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 620H, 1568</td>
<td>MSR_UNCORE_RATIO_LIMIT</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Ratio Limit (R/W)</td>
<td>Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:0</td>
<td>MAX_RATIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This field is used to limit the max ratio of the LLC/Ring.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>MIN_RATIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Writing to this field controls the minimum possible ratio of the LLC/Ring.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 639H, 1593</td>
<td>MSR_PP0_ENERGY_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Reserved (R/O)</td>
<td>Reads return 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_CORE_PERF_LIMIT_REASONS</td>
<td>Indicator of Frequency Clipping in Processor Cores (R/w)</td>
<td>(Frequency refers to processor core frequency.)</td>
<td>Package</td>
</tr>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td>When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal event.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Power Budget Management Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to PBM limit</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Platform Configuration Services Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to PCS limit</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Autonomous Utilization-Based Frequency Control Status (R0)</td>
<td>When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Electrical Design Point Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Multi-Core Turbo Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits.</td>
<td></td>
</tr>
<tr>
<td>12:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Core Frequency P1 Status (R0)</td>
<td>When set, frequency is reduced below max non-turbo P1.</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Core Max N-Core Turbo Frequency Limiting Status (R0)</td>
<td>When set, frequency is reduced below max n-core turbo frequency.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Core Frequency Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Power Budget Management Log</td>
<td>When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Platform Configuration Services Log</td>
<td>When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Autonomous Utilization-Based Frequency Control Log</td>
<td>When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>VR Therm Alert Log</td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Electrical Design Point Log</td>
<td>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Multi-Core Turbo Log</td>
<td>When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>28:27</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Core Frequency P1 Log</td>
<td>When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Core Max N-Core Turbo Frequency Limiting Log</td>
<td>When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-32. Additional MSRs Supported by the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Core Frequency Limiting Log</td>
<td>When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_QM_EVTSEL</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>7:0</td>
<td>EventID (R/w)</td>
<td>Event encoding:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0: No monitoring.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x1: L3 occupancy monitoring.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other encoding reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41:32</td>
<td>RMID (R/w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:42</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_QM_CTR</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>61:0</td>
<td>Resource Monitored Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Unavailable: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Error: If 1, indicates an unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_PQR_ASSOC</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>

NOTES:
1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

### 2.14.1 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family

The Intel Xeon Processor E5 v3 and E7 v3 families are based on Haswell-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-33. For complete details of the uncore PMU, refer to the Intel Xeon Processor E5 v3 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3FH.
## Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>700H, 1792</td>
<td>MSR_PMON_GLOBAL_CTL</td>
<td>Uncore Perfmon Per-Socket Global Control</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>701H, 1793</td>
<td>MSR_PMON_GLOBAL_STATUS</td>
<td>Uncore Perfmon Per-Socket Global Status</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>702H, 1794</td>
<td>MSR_PMON_GLOBAL_CONFIG</td>
<td>Uncore Perfmon Per-Socket Global Configuration</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>703H, 1795</td>
<td>MSR_U_PMON_UCLK_FIXED_CTL</td>
<td>Uncore U-Box UCLK Fixed Counter Control</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>704H, 1796</td>
<td>MSR_U_PMON_UCLK_FIXED_CTR</td>
<td>Uncore U-Box UCLK Fixed Counter</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>705H, 1797</td>
<td>MSR_U_PMON_EVNTSEL0</td>
<td>Uncore U-Box Perfmon Event Select for U-Box Counter 0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>706H, 1798</td>
<td>MSR_U_PMON_EVNTSEL1</td>
<td>Uncore U-Box Perfmon Event Select for U-Box Counter 1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>707H, 1800</td>
<td>MSR_U_PMON_BOX_STATUS</td>
<td>Uncore U-Box Perfmon U-Box Wide Status</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>709H, 1801</td>
<td>MSR_U_PMON_CTR0</td>
<td>Uncore U-Box Perfmon Counter 0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>70AH, 1802</td>
<td>MSR_U_PMON_CTR1</td>
<td>Uncore U-Box Perfmon Counter 1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>710H, 1808</td>
<td>MSR_PCU_PMON_BOX_CTL</td>
<td>Uncore PCU Perfmon for PCU-Box-Wide Control</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>711H, 1809</td>
<td>MSR_PCU_PMON_EVNTSEL0</td>
<td>Uncore PCU Perfmon Event Select for PCU Counter 0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>712H, 1810</td>
<td>MSR_PCU_PMON_EVNTSEL1</td>
<td>Uncore PCU Perfmon Event Select for PCU Counter 1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>713H, 1811</td>
<td>MSR_PCU_PMON_EVNTSEL2</td>
<td>Uncore PCU Perfmon Event Select for PCU Counter 2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>714H, 1812</td>
<td>MSR_PCU_PMON_EVNTSEL3</td>
<td>Uncore PCU Perfmon Event Select for PCU Counter 3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>715H, 1813</td>
<td>MSR_PCU_PMON_BOX_FILTER</td>
<td>Uncore PCU Perfmon Box-Wide Filter</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>716H, 1814</td>
<td>MSR_PCU_PMON_BOX_STATUS</td>
<td>Uncore PCU Perfmon Box Wide Status</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>717H, 1815</td>
<td>MSR_PCU_PMON_CTR0</td>
<td>Uncore PCU Perfmon Counter 0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>718H, 1816</td>
<td>MSR_PCU_PMON_CTR1</td>
<td>Uncore PCU Perfmon Counter 1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Name</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------</td>
<td>----------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 1</td>
<td>MSR_PCU_PMON_CTR2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 719H, 1817</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 2</td>
<td>MSR_PCU_PMON_CTR3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 71AH, 1818</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PCU Perfmon Counter 3</td>
<td>MSR_S0_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 720H, 1824</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon for SBo 0 Box-Wide Control</td>
<td>MSR_S0_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 721H, 1825</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 0</td>
<td>MSR_S0_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 722H, 1826</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 1</td>
<td>MSR_S0_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 723H, 1827</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 2</td>
<td>MSR_S0_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 724H, 1828</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Event Select for SBo 0 Counter 3</td>
<td>MSR_S0_PMON_BOX_FILTER</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 725H, 1829</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Box-Wide Filter</td>
<td>MSR_S0_PMON_CTR0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 726H, 1830</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Counter 0</td>
<td>MSR_S0_PMON_CTR1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 727H, 1831</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Counter 1</td>
<td>MSR_S0_PMON_CTR2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 728H, 1832</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Counter 2</td>
<td>MSR_S0_PMON_CTR3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 729H, 1833</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 0 Perfmon Counter 3</td>
<td>MSR_S1_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 72AH, 1834</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon for SBo 1 Box-Wide Control</td>
<td>MSR_S1_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 72BH, 1835</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 0</td>
<td>MSR_S1_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 72CH, 1836</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 1</td>
<td>MSR_S1_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 72DH, 1837</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 2</td>
<td>MSR_S1_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 72EH, 1838</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Event Select for SBo 1 Counter 3</td>
<td>MSR_S1_PMON_BOX_FILTER</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 72FH, 1839</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Box-Wide Filter</td>
<td></td>
<td></td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 730H, 1840</td>
<td>MSR_S1_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 731H, 1841</td>
<td>MSR_S1_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 732H, 1842</td>
<td>MSR_S1_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Counter 2</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 733H, 1843</td>
<td>MSR_S1_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 1 Perfmon Counter 3</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 734H, 1844</td>
<td>MSR_S2_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon for SBo 2 Box-Wide Control</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 735H, 1845</td>
<td>MSR_S2_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 736H, 1846</td>
<td>MSR_S2_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 737H, 1847</td>
<td>MSR_S2_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 2</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 738H, 1848</td>
<td>MSR_S2_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Event Select for SBo 2 Counter 3</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 739H, 1849</td>
<td>MSR_S2_PMON_BOX_FILTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Box-Wide Filter</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 73AH, 1850</td>
<td>MSR_S2_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 73BH, 1851</td>
<td>MSR_S2_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 73CH, 1852</td>
<td>MSR_S2_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Counter 2</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 73DH, 1853</td>
<td>MSR_S2_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 2 Perfmon Counter 3</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 73EH, 1854</td>
<td>MSR_S3_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon for SBo 3 Box-Wide Control</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 73FH, 1855</td>
<td>MSR_S3_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 740H, 1856</td>
<td>MSR_S3_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 741H, 1857</td>
<td>MSR_S3_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 2</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 742H, 1858</td>
<td>MSR_S3_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Name</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Event Select for SBo 3 Counter 3</td>
<td>MSR_S3_PMON_BOX_FILTER</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Box-Wide Filter</td>
<td>MSR_S3_PMON_CTR0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Counter 0</td>
<td>MSR_S3_PMON_CTR1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Counter 1</td>
<td>MSR_S3_PMON_CTR2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Counter 2</td>
<td>MSR_S3_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore SBo 3 Perfmon Counter 3</td>
<td>MSR_S3_PMON_BOX_FILTER0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon for Box-Wide Control</td>
<td>MSR_C0_PMON_BOX_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 0</td>
<td>MSR_C0_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 1</td>
<td>MSR_C0_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 2</td>
<td>MSR_C0_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Event Select for C-Box 0 Counter 3</td>
<td>MSR_C0_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Box Wide Filter 0</td>
<td>MSR_C0_PMON_BOX_FILTER0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Box Wide Filter 1</td>
<td>MSR_C0_PMON_BOX_FILTER1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Box Wide Status</td>
<td>MSR_C0_PMON_BOX_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Counter 0</td>
<td>MSR_C0_PMON_CTR0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Counter 1</td>
<td>MSR_C0_PMON_CTR1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Counter 2</td>
<td>MSR_C0_PMON_CTR2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Counter 3</td>
<td>MSR_C0_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0 Perfmon Counter 3</td>
<td>MSR_C0_PMON_BOX_CTL</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)
### Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>E11H, 3601</td>
<td>MSR_C1_PMON_EVNTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>E12H, 3602</td>
<td>MSR_C1_PMON_EVNTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>E13H, 3603</td>
<td>MSR_C1_PMON_EVNTSEL2</td>
<td>Package</td>
</tr>
<tr>
<td>E14H, 3604</td>
<td>MSR_C1_PMON_EVNTSEL3</td>
<td>Package</td>
</tr>
<tr>
<td>E15H, 3605</td>
<td>MSR_C1_PMON_BOX_FILTER0</td>
<td>Package</td>
</tr>
<tr>
<td>E16H, 3606</td>
<td>MSR_C1_PMON_BOX_FILTER1</td>
<td>Package</td>
</tr>
<tr>
<td>E17H, 3607</td>
<td>MSR_C1_PMON_BOX_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>E18H, 3608</td>
<td>MSR_C1_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>E19H, 3609</td>
<td>MSR_C1_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>E20H, 3610</td>
<td>MSR_C1_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>E21H, 3611</td>
<td>MSR_C1_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>E22H, 3612</td>
<td>MSR_C1_PMON_BOX_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>E23H, 3613</td>
<td>MSR_C2_PMON_EVNTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>E24H, 3614</td>
<td>MSR_C2_PMON_EVNTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>E25H, 3615</td>
<td>MSR_C2_PMON_EVNTSEL2</td>
<td>Package</td>
</tr>
<tr>
<td>E26H, 3616</td>
<td>MSR_C2_PMON_EVNTSEL3</td>
<td>Package</td>
</tr>
<tr>
<td>E27H, 3617</td>
<td>MSR_C2_PMON_BOX_FILTER0</td>
<td>Package</td>
</tr>
<tr>
<td>E28H, 3618</td>
<td>MSR_C2_PMON_BOX_FILTER1</td>
<td>Package</td>
</tr>
<tr>
<td>E29H, 3619</td>
<td>MSR_C2_PMON_BOX_STATUS</td>
<td>Package</td>
</tr>
</tbody>
</table>

Model-Specific Registers (MSRs)
## Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-Box 2 Perfmon Box Wide Status</td>
<td>MSR_C2_PMON_CTR0</td>
<td>MSR_C2_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E28H, 3624</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2 Perfmon Counter 0</td>
<td>MSR_C2_PMON_CTR1</td>
<td>MSR_C2_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E29H, 3625</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2 Perfmon Counter 1</td>
<td>MSR_C2_PMON_CTR2</td>
<td>MSR_C2_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2AH, 3626</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2 Perfmon Counter 2</td>
<td>MSR_C2_PMON_CTR3</td>
<td>MSR_C2_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E2BH, 3627</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2 Perfmon Counter 3</td>
<td>MSR_C3_PMON_BOX_CTL</td>
<td>MSR_C3_PMON_BOX_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E30H, 3632</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon for Box-Wide Control</td>
<td>MSR_C3_PMON_EVNTSEL0</td>
<td>MSR_C3_PMON_EVNTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E31H, 3633</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 0</td>
<td>MSR_C3_PMON_EVNTSEL1</td>
<td>MSR_C3_PMON_EVNTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E32H, 3634</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 1</td>
<td>MSR_C3_PMON_EVNTSEL2</td>
<td>MSR_C3_PMON_EVNTSEL2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E33H, 3635</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 2</td>
<td>MSR_C3_PMON_EVNTSEL3</td>
<td>MSR_C3_PMON_EVNTSEL3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E34H, 3636</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Event Select for C-Box 3 Counter 3</td>
<td>MSR_C3_PMON_BOX_FILTER0</td>
<td>MSR_C3_PMON_BOX_FILTER0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E35H, 3637</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Box Wide Filter 0</td>
<td>MSR_C3_PMON_BOX_FILTER1</td>
<td>MSR_C3_PMON_BOX_FILTER1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E36H, 3638</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Box Wide Filter 1</td>
<td>MSR_C3_PMON_BOX_STATUS</td>
<td>MSR_C3_PMON_BOX_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E37H, 3639</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Box Wide Status</td>
<td>MSR_C3_PMON_CTR0</td>
<td>MSR_C3_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E38H, 3640</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Counter 0</td>
<td>MSR_C3_PMON_CTR1</td>
<td>MSR_C3_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E39H, 3641</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Counter 1</td>
<td>MSR_C3_PMON_CTR2</td>
<td>MSR_C3_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E3AH, 3642</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Counter 2</td>
<td>MSR_C3_PMON_CTR3</td>
<td>MSR_C3_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E3BH, 3643</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3 Perfmon Counter 3</td>
<td>MSR_C4_PMON_BOX_CTL</td>
<td>MSR_C4_PMON_BOX_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E40H, 3648</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4 Perfmon for Box-Wide Control</td>
<td>MSR_C4_PMON_EVNTSEL0</td>
<td>MSR_C4_PMON_EVNTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E41H, 3649</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_C4_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_BOX_FILTER0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_BOX_FILTER1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_BOX_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_CTR0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_CTR1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_CTR2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C4_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_BOX_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_BOX_FILTER0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_BOX_FILTER1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_BOX_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C5_PMON_CTR0</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: E42H, 3650
Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 1
Package

Register Address: E43H, 3651
Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 2
Package

Register Address: E44H, 3652
Uncore C-Box 4 Perfmon Event Select for C-Box 4 Counter 3
Package

Register Address: E45H, 3653
Uncore C-Box 4 Perfmon Box Wide Filter 0
Package

Register Address: E46H, 3654
Uncore C-Box 4 Perfmon Box Wide Filter 1
Package

Register Address: E47H, 3655
Uncore C-Box 4 Perfmon Box Wide Status
Package

Register Address: E48H, 3656
Uncore C-Box 4 Perfmon Counter 0
Package

Register Address: E49H, 3657
Uncore C-Box 4 Perfmon Counter 1
Package

Register Address: E4AH, 3658
Uncore C-Box 4 Perfmon Counter 2
Package

Register Address: E4BH, 3659
Uncore C-Box 4 Perfmon Counter 3
Package

Register Address: E50H, 3664
Uncore C-Box 5 Perfmon for Box-Wide Control
Package

Register Address: E51H, 3665
Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 0
Package

Register Address: E52H, 3666
Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 1
Package

Register Address: E53H, 3667
Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 2
Package

Register Address: E54H, 3668
Uncore C-Box 5 Perfmon Event Select for C-Box 5 Counter 3
Package

Register Address: E55H, 3669
Uncore C-Box 5 Perfmon Box Wide Filter 0
Package

Register Address: E56H, 3670
Uncore C-Box 5 Perfmon Box Wide Filter 1
Package

Register Address: E57H, 3671
Uncore C-Box 5 Perfmon Box Wide Status
Package

Register Address: E58H, 3672
Uncore C-Box 5 Perfmon Counter 0
<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-Box 5 Perfmon Counter 0</td>
<td>MSR_C5_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E59H, 3673</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5 Perfmon Counter 1</td>
<td>MSR_C5_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E5AH, 3674</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5 Perfmon Counter 2</td>
<td>MSR_C5_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E5BH, 3675</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5 Perfmon Counter 3</td>
<td>MSR_C6_PMON_BOX_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E60H, 3680</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon for Box-Wide Control</td>
<td>MSR_C6_PMON_EVNTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E61H, 3681</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 0</td>
<td>MSR_C6_PMON_EVNTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E62H, 3682</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 1</td>
<td>MSR_C6_PMON_EVNTSEL2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E63H, 3683</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 2</td>
<td>MSR_C6_PMON_EVNTSEL3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E64H, 3684</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Event Select for C-Box 6 Counter 3</td>
<td>MSR_C6_PMON_BOX_FILTER0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E65H, 3685</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Box Wide Filter 0</td>
<td>MSR_C6_PMON_BOX_FILTER1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E66H, 3686</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Box Wide Filter 1</td>
<td>MSR_C6_PMON_BOX_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E67H, 3687</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Box Wide Status</td>
<td>MSR_C6_PMON_CTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E68H, 3688</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Counter 0</td>
<td>MSR_C6_PMON_CTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E69H, 3689</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Counter 1</td>
<td>MSR_C6_PMON_CTR2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E6AH, 3690</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Counter 2</td>
<td>MSR_C6_PMON_CTR3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E6BH, 3691</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6 Perfmon Counter 3</td>
<td>MSR_C7_PMON_BOX_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E70H, 3696</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon for Box-Wide Control</td>
<td>MSR_C7_PMON_EVNTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E71H, 3697</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 0</td>
<td>MSR_C7_PMON_EVNTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E72H, 3698</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: E73H, 3699</td>
<td>MSR_C7_PMON_EVTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 2</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E74H, 3700</td>
<td>MSR_C7_PMON_EVTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Event Select for C-Box 7 Counter 3</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E75H, 3701</td>
<td>MSR_C7_PMON_BOX_FILTER0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Box Wide Filter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E76H, 3702</td>
<td>MSR_C7_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Box Wide Filter 1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E77H, 3703</td>
<td>MSR_C7_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Box Wide Status</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E78H, 3704</td>
<td>MSR_C7_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Counter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E79H, 3705</td>
<td>MSR_C7_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Counter 1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7AH, 3706</td>
<td>MSR_C7_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Counter 2</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7BH, 3707</td>
<td>MSR_C7_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7 Perfmon Counter 3</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E80H, 3712</td>
<td>MSR_C8_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Local Box Wide Control</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E81H, 3713</td>
<td>MSR_C8_PMON_EVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E82H, 3714</td>
<td>MSR_C8_PMON_EVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E83H, 3715</td>
<td>MSR_C8_PMON_EVTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 2</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E84H, 3716</td>
<td>MSR_C8_PMON_EVTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Event Select for C-Box 8 Counter 3</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E85H, 3717</td>
<td>MSR_C8_PMON_BOX_FILTER0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Box Wide Filter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E86H, 3718</td>
<td>MSR_C8_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Box Wide Filter 1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E87H, 3719</td>
<td>MSR_C8_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Box Wide Status</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E88H, 3720</td>
<td>MSR_C8_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E89H, 3721</td>
<td>MSR_C8_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 1</td>
<td>MSR_C8_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E8AH, 3722</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 2</td>
<td>MSR_C8_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E8BH, 3723</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8 Perfmon Counter 3</td>
<td>MSR_C9_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E90H, 3728</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Local Box Wide Control</td>
<td>MSR_C9_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E91H, 3729</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 0</td>
<td>MSR_C9_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E92H, 3730</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 1</td>
<td>MSR_C9_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E93H, 3731</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 2</td>
<td>MSR_C9_PMON_EVNTSEL3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E94H, 3732</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Event Select for C-Box 9 Counter 3</td>
<td>MSR_C9_PMON_BOX_FILTER0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E95H, 3733</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Box Wide Filter 0</td>
<td>MSR_C9_PMON_BOX_FILTER1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E96H, 3734</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Box Wide Filter 1</td>
<td>MSR_C9_PMON_BOX_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E97H, 3735</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Box Wide Status</td>
<td>MSR_C9_PMON_CTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E98H, 3736</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 0</td>
<td>MSR_C9_PMON_CTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E99H, 3737</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 1</td>
<td>MSR_C9_PMON_CTR2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E9AH, 3738</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 2</td>
<td>MSR_C9_PMON_CTR3</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: E9BH, 3739</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 9 Perfmon Counter 3</td>
<td>MSR_C10_PMON_BOX_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EA0H, 3744</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 10 Perfmon Local Box Wide Control</td>
<td>MSR_C10_PMON_EVNTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EA1H, 3745</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 0</td>
<td>MSR_C10_PMON_EVNTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EA2H, 3746</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 1</td>
<td>MSR_C10_PMON_EVNTSEL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EA3H, 3747</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 2</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Register Name</td>
<td>Bit Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------</td>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>EA4H, 3748</td>
<td>MSR_C10_PMON_EVTSEL3</td>
<td>Uncore C-Box 10 Perfmon Event Select for C-Box 10 Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>EAH, 3749</td>
<td>MSR_C10_PMON_BOX_FILTER0</td>
<td>Uncore C-Box 10 Perfmon Box Wide Filter 0</td>
<td>Package</td>
</tr>
<tr>
<td>EAGH, 3750</td>
<td>MSR_C10_PMON_BOX_FILTER1</td>
<td>Uncore C-Box 10 Perfmon Box Wide Filter 1</td>
<td>Package</td>
</tr>
<tr>
<td>EA7H, 3751</td>
<td>MSR_C10_PMON_BOX_STATUS</td>
<td>Uncore C-Box 10 Perfmon Box Wide Status</td>
<td>Package</td>
</tr>
<tr>
<td>EA8H, 3752</td>
<td>MSR_C10_PMON_CTR0</td>
<td>Uncore C-Box 10 Perfmon Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>EA9H, 3753</td>
<td>MSR_C10_PMON_CTR1</td>
<td>Uncore C-Box 10 Perfmon Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>EAAH, 3754</td>
<td>MSR_C10_PMON_CTR2</td>
<td>Uncore C-Box 10 Perfmon Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>EABH, 3755</td>
<td>MSR_C10_PMON_CTR3</td>
<td>Uncore C-Box 10 Perfmon Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>EB0H, 3760</td>
<td>MSR_C11_PMON_BOX_CTL</td>
<td>Uncore C-Box 11 Perfmon Local Box Wide Control</td>
<td>Package</td>
</tr>
<tr>
<td>EB1H, 3761</td>
<td>MSR_C11_PMON_EVTSEL0</td>
<td>Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>EB2H, 3762</td>
<td>MSR_C11_PMON_EVTSEL1</td>
<td>Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>EB3H, 3763</td>
<td>MSR_C11_PMON_EVTSEL2</td>
<td>Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>EB4H, 3764</td>
<td>MSR_C11_PMON_EVTSEL3</td>
<td>Uncore C-Box 11 Perfmon Event Select for C-Box 11 Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>EB5H, 3765</td>
<td>MSR_C11_PMON_BOX_FILTER0</td>
<td>Uncore C-Box 11 Perfmon Box Wide Filter 0</td>
<td>Package</td>
</tr>
<tr>
<td>EB6H, 3766</td>
<td>MSR_C11_PMON_BOX_FILTER1</td>
<td>Uncore C-Box 11 Perfmon Box Wide Filter 1</td>
<td>Package</td>
</tr>
<tr>
<td>EB7H, 3767</td>
<td>MSR_C11_PMON_BOX_STATUS</td>
<td>Uncore C-Box 11 Perfmon Box Wide Status</td>
<td>Package</td>
</tr>
<tr>
<td>EB8H, 3768</td>
<td>MSR_C11_PMON_CTR0</td>
<td>Uncore C-Box 11 Perfmon Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>EB9H, 3769</td>
<td>MSR_C11_PMON_CTR1</td>
<td>Uncore C-Box 11 Perfmon Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>EB1AH, 3770</td>
<td>MSR_C11_PMON_CTR2</td>
<td>Uncore C-Box 11 Perfmon Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Register Name</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>------------------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 11 Perfmon Counter 2</td>
<td>MSR_C11_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EBBH, 3771</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 11 Perfmon Counter 3</td>
<td>MSR_C12_PMON_BOX_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ECOH, 3776</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Local Box Wide Control</td>
<td>MSR_C12_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC1H, 3777</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 0</td>
<td>MSR_C12_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC2H, 3778</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 1</td>
<td>MSR_C12_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC3H, 3779</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 2</td>
<td>MSR_C12_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC4H, 3780</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Event Select for C-Box 12 Counter 3</td>
<td>MSR_C12_PMON_BOX_FILTER0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC5H, 3781</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Box Wide Filter 0</td>
<td>MSR_C12_PMON_BOX_FILTER1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC6H, 3782</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Box Wide Filter 1</td>
<td>MSR_C12_PMON_BOX_STATUS</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC7H, 37833783</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Box Wide Status</td>
<td>MSR_C12_PMON_CTR0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC8H, 3784</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 0</td>
<td>MSR_C12_PMON_CTR1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EC9H, 3785</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 1</td>
<td>MSR_C12_PMON_CTR2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ECAH, 3786</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 2</td>
<td>MSR_C12_PMON_CTR3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ECBH, 3787</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 12 Perfmon Counter 3</td>
<td>MSR_C13_PMON_BOX_CTL</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: EDOH, 3792</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon local box wide control</td>
<td>MSR_C13_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ED1H, 3793</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 0</td>
<td>MSR_C13_PMON_EVNTSEL1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ED2H, 3794</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 1</td>
<td>MSR_C13_PMON_EVNTSEL2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ED3H, 3795</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 2</td>
<td>MSR_C13_PMON_EVNTSEL3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ED4H, 3796</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 13 Perfmon Event Select for C-Box 13 Counter 3</td>
<td>MSR_C13_PMON_EVNTSEL0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: ED5H, 3797</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-33. Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: ED5H, 3797</td>
<td>MSR_C13_PMON_BOX_FILTER0</td>
<td>Uncore C-Box 13 Perfmon Box Wide Filter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: ED6H, 3798</td>
<td>MSR_C13_PMON_BOX_FILTER1</td>
<td>Uncore C-Box 13 Perfmon Box Wide Filter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: ED7H, 3799</td>
<td>MSR_C13_PMON_BOX_STATUS</td>
<td>Uncore C-Box 13 Perfmon Box Wide Status</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: ED8H, 3800</td>
<td>MSR_C13_PMON_CTR0</td>
<td>Uncore C-Box 13 Perfmon Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: ED9H, 3801</td>
<td>MSR_C13_PMON_CTR1</td>
<td>Uncore C-Box 13 Perfmon Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EDAH, 3802</td>
<td>MSR_C13_PMON_CTR2</td>
<td>Uncore C-Box 13 Perfmon Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EDBH, 3803</td>
<td>MSR_C13_PMON_CTR3</td>
<td>Uncore C-Box 13 Perfmon Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE0H, 3808</td>
<td>MSR_C14_PMON_BOX_CTL</td>
<td>Uncore C-Box 14 Perfmon Local Box Wide Control</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE1H, 3809</td>
<td>MSR_C14_PMON_EVNTSEL0</td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE2H, 3810</td>
<td>MSR_C14_PMON_EVNTSEL1</td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE3H, 3811</td>
<td>MSR_C14_PMON_EVNTSEL2</td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE4H, 3812</td>
<td>MSR_C14_PMON_EVNTSEL3</td>
<td>Uncore C-Box 14 Perfmon Event Select for C-Box 14 Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE5H, 3813</td>
<td>MSR_C14_PMON_BOX_FILTER</td>
<td>Uncore C-Box 14 Perfmon Box Wide Filter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE6H, 3814</td>
<td>MSR_C14_PMON_BOX_FILTER1</td>
<td>Uncore C-Box 14 Perfmon Box Wide Filter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE7H, 3815</td>
<td>MSR_C14_PMON_BOX_STATUS</td>
<td>Uncore C-Box 14 Perfmon Box Wide Status</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE8H, 3816</td>
<td>MSR_C14_PMON_CTR0</td>
<td>Uncore C-Box 14 Perfmon Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EE9H, 3817</td>
<td>MSR_C14_PMON_CTR1</td>
<td>Uncore C-Box 14 Perfmon Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EEAH, 3818</td>
<td>MSR_C14_PMON_CTR2</td>
<td>Uncore C-Box 14 Perfmon Counter 2</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: EEBH, 3819</td>
<td>MSR_C14_PMON_CTR3</td>
<td>Uncore C-Box 14 Perfmon Counter 3</td>
<td>Package</td>
</tr>
<tr>
<td>Register Name</td>
<td>Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_BOX_CTL</td>
<td>Uncore C-Box 14 Perfmon Counter 3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_EVNTSEL0</td>
<td>Uncore C-Box 15 Perfmon Local Box Wide Control</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_EVNTSEL1</td>
<td>Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_EVNTSEL2</td>
<td>Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_EVNTSEL3</td>
<td>Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_BOX_FILTER0</td>
<td>Uncore C-Box 15 Perfmon Event Select for C-Box 15 Counter 3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_BOX_FILTER1</td>
<td>Uncore C-Box 15 Perfmon Box Wide Filter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_BOX_FILTER2</td>
<td>Uncore C-Box 15 Perfmon Box Wide Filter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_BOX_STATUS</td>
<td>Uncore C-Box 15 Perfmon Box Wide Status</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_CTR0</td>
<td>Uncore C-Box 15 Perfmon Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_CTR1</td>
<td>Uncore C-Box 15 Perfmon Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_CTR2</td>
<td>Uncore C-Box 15 Perfmon Counter 2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C15_PMON_CTR3</td>
<td>Uncore C-Box 15 Perfmon Counter 3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C16_PMON_BOX_CTL</td>
<td>Uncore C-Box 16 Perfmon for Box-Wide Control</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C16_PMON_EVNTSEL0</td>
<td>Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C16_PMON_EVNTSEL1</td>
<td>Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C16_PMON_EVNTSEL2</td>
<td>Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 2</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C16_PMON_EVNTSEL3</td>
<td>Uncore C-Box 16 Perfmon Event Select for C-Box 16 Counter 3</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>MSR_C16_PMON_BOX_FILTER0</td>
<td>Uncore C-Box 16 Perfmon Box Wide Filter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Register Address: F06H, 3846</td>
<td>MSR_C16_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 16 Perfmon Box Wide Filter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F07H, 3847</td>
<td>MSR_C16_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 16 Perfmon Box Wide Status</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F08H, 3848</td>
<td>MSR_C16_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 16 Perfmon Counter 0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F09H, 3849</td>
<td>MSR_C16_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 16 Perfmon Counter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F0AH, 3850</td>
<td>MSR_C16_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 16 Perfmon Counter 2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F0BH, 3851</td>
<td>MSR_C16_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 16 Perfmon Counter 3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F10H, 3856</td>
<td>MSR_C17_PMON_BOX_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon for Box-Wide Control</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F11H, 3857</td>
<td>MSR_C17_PMON_EVNTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F12H, 3858</td>
<td>MSR_C17_PMON_EVNTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F13H, 3859</td>
<td>MSR_C17_PMON_EVNTSEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F14H, 3860</td>
<td>MSR_C17_PMON_EVNTSEL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Event Select for C-Box 17 Counter 3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F15H, 3861</td>
<td>MSR_C17_PMON_BOX_FILTER0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Box Wide Filter 0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F16H, 3862</td>
<td>MSR_C17_PMON_BOX_FILTER1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Box Wide Filter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F17H, 3863</td>
<td>MSR_C17_PMON_BOX_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Box Wide Status</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F18H, 3864</td>
<td>MSR_C17_PMON_CTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Counter 0</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F19H, 3865</td>
<td>MSR_C17_PMON_CTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Counter 1</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F1AH, 3866</td>
<td>MSR_C17_PMON_CTR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Counter 2</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: F1BH, 3867</td>
<td>MSR_C17_PMON_CTR3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 17 Perfmon Counter 3</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
2.15 MSRS IN THE INTEL® CORE™ M PROCESSORS AND THE 5TH GENERATION INTEL® CORE™ PROCESSORS

The Intel® Core™ M-5xxx processors, 5th generation Intel® Core™ Processors, and the Intel® Xeon® Processor E3-1200 v4 family are based on Broadwell microarchitecture. The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_3DH. The Intel® Xeon® Processor E3-1200 v4 family and 5th generation Intel® Core™ Processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_47H. Processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3DH or 06_47H support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, Table 2-34, and Table 2-35. For an MSR listed in Table 2-35 that also appears in the model-specific tables of prior generations, Table 2-35 supersedes prior generation tables.

Table 2-34 lists MSRs that are common to processors based on the Broadwell microarchitectures (including CPUID Signature DisplayFamily_DisplayModel values of 06_3DH, 06_47H, 06_4FH, and 06_56H).

Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td>See Table 2-2 and Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>Ovf_PMC0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ovf_PMC1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ovf_PMC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ovf_PMC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Ovf_FixedCtr0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Ovf_FixedCtr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Ovf_FixedCtr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54:35</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Trace_ToPA_PMI</td>
<td>See Section 33.2.7.2, “Table of Physical Addresses (ToPA).”</td>
<td></td>
</tr>
<tr>
<td>60:56</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Ovf_Uncore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Ovf_BufDSSAVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>CondChgd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 390H, 912</td>
<td>IA32_PERF_GLOBAL_OVF_CTRL</td>
<td>See Table 2-2 and Section 20.6.2.2, “Global Counter Control Facilities.”</td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>Set 1 to clear Ovf_PMC0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Set 1 to clear Ovf_PMC1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Set 1 to clear Ovf_PMC2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Set 1 to clear Ovf_PMC3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to clear Ovf_FixedCtr0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to clear Ovf_FixedCtr1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to clear Ovf_FixedCtr2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>54:35</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Set 1 to clear Trace_toPA_PMI. See Section 33.2.7.2, “Table of Physical Addresses (ToPA).”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60:56</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Set 1 to clear Ovf_Uncore.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Set 1 to clear Ovf_BufDSSAVE.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Set 1 to clear CondChgd.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 560H, 1376</td>
<td>IA32_RTIT_OUTPUT_BASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:0</td>
<td>Reserved</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>MAXPHYADDR1-1:7</td>
<td>Base physical address.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>63:MAXPHYADDR</td>
<td>Reserved</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 561H, 1377</td>
<td>IA32_RTIT_OUTPUT_MASK_PTRS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>6:0</td>
<td>Reserved</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>31:7</td>
<td>MaskOrTableOffset</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>63:32</td>
<td>Output Offset.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 570H, 1392</td>
<td>IA32_RTIT_CTL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>TraceEn</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>1</td>
<td>Reserved, must be zero.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>3</td>
<td>User</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved, must be zero.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>7</td>
<td>CR3Filter</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>8</td>
<td>ToPA</td>
<td>Writing 0 will #GP if also setting TraceEn.</td>
<td>Thread</td>
</tr>
<tr>
<td>9</td>
<td>Reserved, must be zero.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>10</td>
<td>TSCEn</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>11</td>
<td>DisRETC</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>12</td>
<td>Reserved, must be zero.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>13</td>
<td>Reserved; writing 0 will #GP if also setting TraceEn.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>63:14</td>
<td>Reserved, must be zero.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 571H, 1393</td>
<td>IA32_RTIT_STATUS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>Reserved, writes ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ContextEn, writes ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TriggerEn, writes ignored.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-306 Vol. 4
### Table 2-34. Additional MSRs Common to Processors Based on Broadwell Microarchitectures

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Error (R/W)</td>
</tr>
<tr>
<td>5</td>
<td>Stopped</td>
</tr>
<tr>
<td>63:6</td>
<td>Reserved, must be zero.</td>
</tr>
</tbody>
</table>

Register Address: 572H, 1394

**IA32_RTIT_CR3_MATCH**

Trace Filter CR3 Match Register (R/W)

Thread

| 4:0                           | Reserved.     |

Register Address: 620H, 1568

**MSR_UNCORE_RATIO_LIMIT**

Uncore Ratio Limit (R/W)

Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.

| 6:0                           | MAX_RATIO     |
|                              | This field is used to limit the max ratio of the LLC/Ring. |
| 7                             | Reserved.     |
| 14:8                          | MIN_RATIO     |
|                              | Writing to this field controls the minimum possible ratio of the LLC/Ring. |
| 63:15                         | Reserved.     |

**NOTES:**

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 2-35 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

### Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td>C-State Configuration Control (R/W)</td>
</tr>
</tbody>
</table>

Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Core
### Table 2-35. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10</td>
<td></td>
</tr>
<tr>
<td>9:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/W/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>C3 State Auto Demotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>C1 State Auto Demotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Enable C3 Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Enable C1 Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Enable Package C-State Auto-Demotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Enable Package C-State Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:31</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td>Maximum Ratio Limit of Turbo Mode R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C</td>
<td>Maximum turbo ratio limit of 1 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 2C</td>
<td>Maximum turbo ratio limit of 2 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 3C</td>
<td>Maximum turbo ratio limit of 3 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 4C</td>
<td>Maximum turbo ratio limit of 4 core active.</td>
<td>Package</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 5C</td>
<td>Maximum turbo ratio limit of 5 core active.</td>
<td>Package</td>
</tr>
</tbody>
</table>
2.16 MSRS IN THE INTEL® XEON® PROCESSOR E5 V4 FAMILY

The MSRs listed in Table 2-36 are available and common to the Intel® Xeon® Processor D Product Family (CPUID Signature DisplayFamily_DisplayModel value of 06_56H) and to the Intel Xeon processors E5 v4 and E7 v4 families (CPUID Signature DisplayFamily_DisplayModel value of 06_4FH). These processors are based on Broadwell microarchitecture.

See Section 2.16.1 for lists of tables of MSRs that are supported by the Intel® Xeon® Processor D Family.

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_PPIN_CTL (MSR_PPIN_CTL)</td>
<td>Protected Processor Inventory Number Enable Control (R/W)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LockOut (R/WO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable_PPIN (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_PPIN (MSR_PPIN)</td>
<td>Protected Processor Inventory Number (R/O)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Protected Processor Inventory Number (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>MSR_PLATFORM_INFO</td>
<td>Platform Information</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>C6:6</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>22:16</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-20, Table 2-21, Table 2-22, Table 2-25, Table 2-29, Table 2-30, and Table 2-34 for other MSR definitions applicable to processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_3DH.
Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>PPIN_CAP (R/O)</td>
<td>See Table 2-26.</td>
<td>Package</td>
</tr>
<tr>
<td>27:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
<td>See Table 2-26.</td>
<td>Package</td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
<td>See Table 2-26.</td>
<td>Package</td>
</tr>
<tr>
<td>30</td>
<td>Programmable TJ OFFSET (R/O)</td>
<td>See Table 2-26.</td>
<td>Package</td>
</tr>
<tr>
<td>39:31</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>See Table 2-26.</td>
<td>Package</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-State Configuration Control (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.</td>
<td></td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/wO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Automatic C-State Conversion Enable (R/W)</td>
<td>If 1, the processor will convert HALT or MWAIT(C1) to MWAIT(C6).</td>
<td></td>
</tr>
<tr>
<td>24:17</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>C3 State Auto Demotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>C1 State Auto Demotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Enable C3 Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Enable C1 Undemotion (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Package C State Demotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
<td>Package C State Undemotion Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:31</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td></td>
<td>IA32_MCG_CAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Machine Check Capability (R/O)</td>
<td>Count</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>MCG_CTL_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>MCG_EXT_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>MCP_CMCI_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>MCG_TES_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>MCG_EXT_CNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>MCG_SER_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>MCG_EM_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>MCG_ELOG_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:27</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17DH, 381</td>
<td></td>
<td>MSR_SMM_MCA_CAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enhanced SMM Capabilities (SMM-RO)</td>
<td>Reports SMM capability Enhancement. Accessible only while in SMM.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57:0</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td></td>
<td>SMM_Code_Access_Chk (SMM-RO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td>Long_FLOW_Indication (SMM-RO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:60</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td></td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Thermal Status (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Thermal Status Log (R/WC0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>PROTCHT # or FORCEPR# Status (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>PROTCHT # or FORCEPR# Log (R/WC0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register Information / Bit Fields**

- **30**: Package C State Undemotion Enable (R/W)
- **63:31**: Reserved.
- **7:0**: Count
- **8**: MCG_CTL_P
- **9**: MCG_EXT_P
- **10**: MCP_CMCI_P
- **11**: MCG_TES_P
- **15:12**: Reserved
- **23:16**: MCG_EXT_CNT
- **24**: MCG_SER_P
- **25**: MCG_EM_P
- **26**: MCG_ELOG_P
- **63:27**: Reserved.

**Register Address**

- **179H, 377**: IA32_MCG_CAP
- **17DH, 381**: MSR_SMM_MCA_CAP
- **19CH, 412**: IA32_THERM_STATUS

**Bit Description**

- **Count**: Thread
- **MCG_CTL_P**: Thread
- **MCG_EXT_P**: Thread
- **MCP_CMCI_P**: Thread
- **MCG_TES_P**: Thread
- **Reserved**: Thread
- **SMM_Code_Access_Chk (SMM-RO)**: If set to 1, indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
- **Long_FLOW_Indication (SMM-RO)**: If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>4</td>
<td>Critical Temperature Status (R/O)</td>
</tr>
<tr>
<td>5</td>
<td>Critical Temperature Status Log (R/WC0)</td>
</tr>
<tr>
<td>6</td>
<td>Thermal Threshold #1 Status (R/O)</td>
</tr>
<tr>
<td>7</td>
<td>Thermal Threshold #1 Log (R/WC0)</td>
</tr>
<tr>
<td>8</td>
<td>Thermal Threshold #2 Status (R/O)</td>
</tr>
<tr>
<td>9</td>
<td>Thermal Threshold #2 Log (R/WC0)</td>
</tr>
<tr>
<td>10</td>
<td>Power Limitation Status (R/O)</td>
</tr>
<tr>
<td>11</td>
<td>Power Limitation Log (R/WC0)</td>
</tr>
<tr>
<td>12</td>
<td>Current Limit Status (R/O)</td>
</tr>
<tr>
<td>13</td>
<td>Current Limit Log (R/WC0)</td>
</tr>
<tr>
<td>14</td>
<td>Cross Domain Limit Status (R/O)</td>
</tr>
<tr>
<td>15</td>
<td>Cross Domain Limit Log (R/WC0)</td>
</tr>
<tr>
<td>22:16</td>
<td>Digital Readout (R/O)</td>
</tr>
<tr>
<td>26:23</td>
<td>Reserved.</td>
</tr>
<tr>
<td>30:27</td>
<td>Resolution in Degrees Celsius (R/O)</td>
</tr>
<tr>
<td>31</td>
<td>Reading Valid (R/O)</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: 1A2H, 418</strong></td>
<td><strong>MSR_TEMPERATURE_TARGET</strong></td>
</tr>
<tr>
<td><strong>Temperature Target</strong></td>
<td><strong>Package</strong></td>
</tr>
<tr>
<td>15:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>23:16</td>
<td>Temperature Target (R/O)</td>
</tr>
<tr>
<td>27:24</td>
<td>TCC Activation Offset (R/W)</td>
</tr>
</tbody>
</table>
Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>63:28</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 2C</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 3C</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 4C</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 5C</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for 6C</td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for 7C</td>
</tr>
<tr>
<td>63:56</td>
<td>Maximum Ratio Limit for 8C</td>
</tr>
<tr>
<td>Register Address: 1AEH, 430</td>
<td>MSR_TURBO_RATIO_LIMIT1</td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode</td>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
</tr>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 9C</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 10C</td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 11C</td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 12C</td>
</tr>
<tr>
<td>39:32</td>
<td>Maximum Ratio Limit for 13C</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Ratio Limit for 14C</td>
</tr>
<tr>
<td>55:48</td>
<td>Maximum Ratio Limit for 15C</td>
</tr>
<tr>
<td>63:56</td>
<td>Maximum Ratio Limit for 16C</td>
</tr>
<tr>
<td>Register Address: 606H, 1542</td>
<td>MSR_RAPL_POWER_UNIT</td>
</tr>
<tr>
<td>Unit Multipliers Used in RAPL Interfaces (R/O)</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Power Units</td>
</tr>
<tr>
<td>7:4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>12:8</td>
<td>Energy Status Units</td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
</tr>
<tr>
<td>19:16</td>
<td>Time Units</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 618H, 1560</td>
<td>MSR_DRAM_POWER_LIMIT</td>
</tr>
</tbody>
</table>
### MODEL-SPECIFIC REGISTERS (MSRS)

#### DRAM RAPL Power Limit Control (R/W)
See Section 15.10.5, “DRAM RAPL Domain.”

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM RAPL Power Limit Control (R/w)</td>
<td>MSR_DRAM_ENERGY_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Energy consumed by DRAM devices.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 619H, 1561

#### DRAM Energy Status (R/O)

Register Address: 61BH, 1563

#### DRAM Performance Throttling Status (R/O)
See Section 15.10.5, “DRAM RAPL Domain.”

Register Address: 61CH, 1564

#### DRAM RAPL Parameters (R/W)
See Section 15.10.5, “DRAM RAPL Domain.”

Register Address: 620H, 1568

#### Uncore Ratio Limit (R/W)
Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.

Register Address: 639H, 1593

#### MSR_PP0_ENERGY_STATUS
Reserved (R/O)
Reads return 0.

Register Address: 690H, 1680

#### MSR_CORE_PERF_LIMIT_REASONS
Indicator of Frequency Clipping in Processor Cores (R/W)
(Frequency refers to processor core frequency.)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
</tr>
<tr>
<td>2</td>
<td>Power Budget Management Status (R0)</td>
</tr>
</tbody>
</table>

Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Platform Configuration Services Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to PCS limit.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Autonomous Utilization-Based Frequency Control Status (R0)</td>
<td>When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Electrical Design Point Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g., maximum electrical current consumption).</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Multi-Core Turbo Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Core Frequency P1 Status (R0)</td>
<td>When set, frequency is reduced below max non-turbo P1.</td>
<td></td>
</tr>
<tr>
<td>12:11</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Core Max N-Core Turbo Frequency Limiting Status (R0)</td>
<td>When set, frequency is reduced below max n-core turbo frequency.</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Core Frequency Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Power Budget Management Log</td>
<td>When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Platform Configuration Services Log</td>
<td>When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Reserved.</td>
<td>Autonomic Utilization-Based Frequency Control Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Reserved.</td>
<td>VR Therm Alert Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Reserved.</td>
<td>Electrical Design Point Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the EDP Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Reserved.</td>
<td>Multi-Core Turbo Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Reserved.</td>
<td>Core Frequency P1 Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Reserved.</td>
<td>Core Max N-Core Turbo Frequency Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Reserved.</td>
<td>Core Frequency Limiting Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td>IA32_P_M_ENABLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 770H, 1904</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Section 15.4.2, “Enabling HWP.”</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
<td>IA32_HWP_CAPABILITIES</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 771H, 1905</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Section 15.4.3, “HWP Performance Range and Dynamic Capabilities.”</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Reserved.</td>
<td>IA32_HWP_REQUEST</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register Address: 774H, 1908</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Section 15.4.4, “Managing HWP.”</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minimum Performance (R/W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maximum Performance (R/W)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>23:16</td>
<td>Desired Performance (R/W)</td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: 777H, 1911</strong></td>
<td>IA32_HWP_STATUS</td>
</tr>
<tr>
<td>See Section 15.4.5, “HWP Feedback.”</td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>2</td>
<td>Excursion to Minimum (R/O)</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: C8DH, 3213</strong></td>
<td>IA32_QM_EVTSEL</td>
</tr>
<tr>
<td>Monitoring Event Select Register (R/W)</td>
<td></td>
</tr>
<tr>
<td>If CPUID.(EAX=07H, ECX=0);EBX.RDT-M[bit 12] = 1.</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>EventID (R/W)</td>
</tr>
<tr>
<td>Event encoding:</td>
<td></td>
</tr>
<tr>
<td>0x00: No monitoring.</td>
<td></td>
</tr>
<tr>
<td>0x01: L3 occupancy monitoring.</td>
<td></td>
</tr>
<tr>
<td>0x02: Total memory bandwidth monitoring.</td>
<td></td>
</tr>
<tr>
<td>0x03: Local memory bandwidth monitoring.</td>
<td></td>
</tr>
<tr>
<td>All other encoding reserved.</td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
</tr>
<tr>
<td>41:32</td>
<td>RMID (R/W)</td>
</tr>
<tr>
<td>63:42</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: CBFH, 3215</strong></td>
<td>IA32_PQR_ASSOC</td>
</tr>
<tr>
<td>Resource Association Register (R/W)</td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>RMID</td>
</tr>
<tr>
<td>31:10</td>
<td>Reserved.</td>
</tr>
<tr>
<td>51:32</td>
<td>COS (R/W)</td>
</tr>
<tr>
<td>63:52</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: C90H, 3216</strong></td>
<td>IA32_L3_QOS_MASK_0</td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 0 (R/W)</td>
<td></td>
</tr>
<tr>
<td>If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] &gt;=0.</td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 0 enforcement.</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: C91H, 3217</strong></td>
<td>IA32_L3_QOS_MASK_1</td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 1 (R/W)</td>
<td></td>
</tr>
<tr>
<td>If CPUID.(EAX=10H, ECX=1);EDX.COS_MAX[15:0] &gt;=1.</td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 1 enforcement.</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: C92H, 3218</strong></td>
<td>IA32_L3_QOS_MASK_2</td>
</tr>
</tbody>
</table>
### Table 2-36. Additional MSRs Common to the Intel® Xeon® Processor D and the Intel® Xeon® Processor E5 v4 Family Based on Broadwell Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 Class Of Service Mask - COS 2 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=2.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 2 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C93H, 3219</td>
<td>IA32_L3_QOS_MASK_3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 3 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=3.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 3 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C94H, 3220</td>
<td>IA32_L3_QOS_MASK_4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 4 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=4.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 4 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C95H, 3221</td>
<td>IA32_L3_QOS_MASK_5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 5 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=5.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 5 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C96H, 3222</td>
<td>IA32_L3_QOS_MASK_6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C97H, 3223</td>
<td>IA32_L3_QOS_MASK_7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 7 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=7.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 7 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C98H, 3224</td>
<td>IA32_L3_QOS_MASK_8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 8 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=8.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 8 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C99H, 3225</td>
<td>IA32_L3_QOS_MASK_9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The MSRs listed in Table 2-37 are available to Intel® Xeon® Processor D Product Family (CPUID Signature DisplayFamily_DisplayModel value of 06_56H). The Intel® Xeon® processor D product family is based on Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-20, Table 2-29, Table 2-34, Table 2-36, and Table 2-37.
Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_56H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 1ACH, 428</td>
<td>MSR_TURBO_RATIO_LIMIT3</td>
<td>Config Ratio Limit of Turbo Mode</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>R/W if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td>62:0 Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>63 Semaphore for Turbo Ratio Limit Configuration</td>
<td>If 1, the processor uses override configuration specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 286H, 646</td>
<td>IA32_MC6_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 287H, 647</td>
<td>IA32_MC7_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 289H, 649</td>
<td>IA32_MC9_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 28AH, 650</td>
<td>IA32_MC10_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 418H, 1048</td>
<td>IA32_MC6_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 419H, 1049</td>
<td>IA32_MC6_STATUS</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41AH, 1050</td>
<td>IA32_MC6_ADDR</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41BH, 1051</td>
<td>IA32_MC6_MISC</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.” Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 41CH, 1052</td>
<td>IA32_MC7_CTL</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.” Bank MC7 reports MC errors from the home agent HA 0.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

#### Register Address: Hex, Decimal

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 41DH, 1053</td>
<td>IA32_MC7_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41EH, 1054</td>
<td>IA32_MC7_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41FH, 1055</td>
<td>IA32_MC7_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC7 reports MC errors from the home agent HA 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 424H, 1060</td>
<td>IA32_MC9_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 425H, 1061</td>
<td>IA32_MC9_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td>IA32_MC9_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td>IA32_MC9_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td>IA32_MC10_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td>IA32_MC10_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td>IA32_MC10_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 42BH, 1067</td>
<td>IA32_MC10_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC9 through MC 10 report MC errors from each channel of the integrated memory controllers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 444H, 1092</td>
<td>IA32_MC17_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 445H, 1093</td>
<td>IA32_MC17_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-37. Additional MSRs Supported by Intel® Xeon® Processor D with a CPUID Signature DisplayFamily_DisplayModel Value of 06_56H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Regiser Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>446H, 1094</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC17_ADDR</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>Package</td>
</tr>
<tr>
<td>447H, 1095</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC17_MISC</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>Package</td>
</tr>
<tr>
<td>448H, 1096</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC18_CTL</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>449H, 1097</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC18_STATUS</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>44AH, 1098</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC18_ADDR</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>44BH, 1099</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC18_MISC</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
</tr>
<tr>
<td>44CH, 1100</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC19_CTL</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
<tr>
<td>44DH, 1101</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC19_STATUS</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
<tr>
<td>44EH, 1102</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC19_ADDR</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
<tr>
<td>44FH, 1103</td>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>IA32_MC19_MISC</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
</tr>
</tbody>
</table>
The MSRs listed in Table 2-37 are available to the Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID Signature DisplayFamily_DisplayModel value of 06_4FH). The Intel® Xeon® processor E5 v4 family is based on Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-34, Table 2-36, and Table 2-38.

NOTES:
1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 1ACH, 428</td>
<td>MSR_TURBO_RATIO_LIMIT3</td>
<td>Config Ratio Limit of Turbo Mode</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1.</td>
<td></td>
</tr>
<tr>
<td>62:0</td>
<td>Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63</td>
<td>Semaphore for Turbo Ratio Limit Configuration</td>
<td>If 1, the processor uses override configuration specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1, and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 285H, 645</td>
<td>IA32_MC5_CTL2</td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>

See Table 2-2.

Register Address: 286H, 646 IA32_MC6_CTL2
See Table 2-2.

Register Address: 287H, 647 IA32_MC7_CTL2
See Table 2-2.

Register Address: 288H, 648 IA32_MC8_CTL2
See Table 2-2.

Register Address: 289H, 649 IA32_MC9_CTL2
See Table 2-2.

Register Address: 28AH, 650 IA32_MC10_CTL2
See Table 2-2.

2.16.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families

The MSRs listed in Table 2-37 are available to the Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID Signature DisplayFamily_DisplayModel value of 06_4FH). The Intel® Xeon® processor E5 v4 family is based on Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-29, Table 2-34, Table 2-36, and Table 2-38.
### Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 28BH, 651</td>
<td>IA32_MC11_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28CH, 652</td>
<td>IA32_MC12_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28DH, 653</td>
<td>IA32_MC13_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28EH, 654</td>
<td>IA32_MC14_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 28FH, 655</td>
<td>IA32_MC15_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 290H, 656</td>
<td>IA32_MC16_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 294H, 660</td>
<td>IA32_MC20_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 295H, 661</td>
<td>IA32_MC21_CTL2</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MC5_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MC5_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 416H, 1046</td>
<td>IA32_MC5_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 417H, 1047</td>
<td>IA32_MC5_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC5 reports MC errors from the Intel QPI 0 module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 418H, 1048</td>
<td>IA32_MC6_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>IA32_MC6_STATUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 419H, 1049</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC6_ADDR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41AH, 1050</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC6_MISC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41BH, 1051</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC7_CTL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41CH, 1052</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC7 reports MC errors from the home agent HA 0.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC7_STATUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41DH, 1053</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC7 reports MC errors from the home agent HA 0.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC7_ADDR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41EH, 1054</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC7 reports MC errors from the home agent HA 0.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC7_MISC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 41FH, 1055</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC7 reports MC errors from the home agent HA 0.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Register Address: 420H, 1056</strong></td>
<td><strong>IA32_MC8_CTL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC8 reports MC errors from the home agent HA 1.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC8_STATUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 421H, 1057</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC8 reports MC errors from the home agent HA 1.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC8_ADDR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 422H, 1058</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC8 reports MC errors from the home agent HA 1.</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC8_MISC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 423H, 1059</td>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Bank MC8 reports MC errors from the home agent HA 1.</td>
<td>Package</td>
</tr>
<tr>
<td><strong>Register Address: 424H, 1060</strong></td>
<td><strong>IA32_MC9_CTL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot; Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>IA32_MC9_STATUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Register Name</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------</td>
<td>---------------</td>
<td>-------</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td>IA32_MC9_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td>IA32_MC9_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td>IA32_MC10_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td>IA32_MC10_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td>IA32_MC10_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42BH, 1067</td>
<td>IA32_MC10_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42CH, 1068</td>
<td>IA32_MC11_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42DH, 1069</td>
<td>IA32_MC11_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42EH, 1070</td>
<td>IA32_MC11_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 42FH, 1071</td>
<td>IA32_MC11_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 430H, 1072</td>
<td>IA32_MC12_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 431H, 1073</td>
<td>IA32_MC12_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 432H, 1074</td>
<td>IA32_MC12_ADDR</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC12_CTL MSRs,” through Section 16.3.2.4, “IA32_MC12_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 433H, 1075</td>
<td>IA32_MC12_MISC</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC12_CTL MSRs,” through Section 16.3.2.4, “IA32_MC12_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 434H, 1076</td>
<td>IA32_MC13_CTL</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC13_CTL MSRs,” through Section 16.3.2.4, “IA32_MC13_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 435H, 1077</td>
<td>IA32_MC13_STATUS</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC13_CTL MSRs,” through Section 16.3.2.4, “IA32_MC13_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 436H, 1078</td>
<td>IA32_MC13_ADDR</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC13_CTL MSRs,” through Section 16.3.2.4, “IA32_MC13_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 437H, 1079</td>
<td>IA32_MC13_MISC</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC13_CTL MSRs,” through Section 16.3.2.4, “IA32_MC13_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 438H, 1080</td>
<td>IA32_MC14_CTL</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC14_CTL MSRs,” through Section 16.3.2.4, “IA32_MC14_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 439H, 1081</td>
<td>IA32_MC14_STATUS</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC14_CTL MSRs,” through Section 16.3.2.4, “IA32_MC14_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43AH, 1082</td>
<td>IA32_MC14_ADDR</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC14_CTL MSRs,” through Section 16.3.2.4, “IA32_MC14_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43BH, 1083</td>
<td>IA32_MC14_MISC</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC14_CTL MSRs,” through Section 16.3.2.4, “IA32_MC14_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43CH, 1084</td>
<td>IA32_MC15_CTL</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC15_CTL MSRs,” through Section 16.3.2.4, “IA32_MC15_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43DH, 1085</td>
<td>IA32_MC15_STATUS</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC15_CTL MSRs,” through Section 16.3.2.4, “IA32_MC15_MISC MSRs.” Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43EH, 1086</td>
<td>IA32_MC15_ADDR</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

**Table 2-38. Additional MSRs Supported by Intel® Xeon® Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_4FH**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>IA32_MC15_MISC</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 43FH, 1087</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>IA32_MC16_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 440H, 1088</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>IA32_MC16_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 441H, 1089</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>IA32_MC16_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 442H, 1090</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>IA32_MC17_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 443H, 1091</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Banks MC9 through MC 16 report MC errors from each channel of the integrated memory controllers.</td>
<td>IA32_MC17_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 444H, 1092</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>IA32_MC17_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 445H, 1093</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>IA32_MC17_MISC</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 446H, 1094</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>IA32_MC18_CTL</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 447H, 1095</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC17 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.</td>
<td>IA32_MC18_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 448H, 1096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>IA32_MC18_ADDR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 449H, 1097</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44AH, 1098</td>
<td>IA32_MC18_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC18 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44BH, 1099</td>
<td>IA32_MC18_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44DH, 1101</td>
<td>IA32_MC19_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44EH, 1102</td>
<td>IA32_MC19_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC19 reports MC errors from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44FH, 1103</td>
<td>IA32_MC19_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 450H, 1104</td>
<td>IA32_MC20_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 451H, 1105</td>
<td>IA32_MC20_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 452H, 1106</td>
<td>IA32_MC20_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 453H, 1107</td>
<td>IA32_MC20_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCI_MISC MSRs.&quot;</td>
<td>Bank MC20 reports MC errors from the Intel QPI 1 module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 454H, 1108</td>
<td>IA32_MC21_CTL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.17 MSRS IN THE 6TH—13TH GENERATION INTEL® CORE™ PROCESSORS, 1ST—5TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILIES, INTEL® CORE™ ULTRA 7 PROCESSORS, 8TH GENERATION INTEL® CORE™ I3 PROCESSORS, AND INTEL® XEON® E PROCESSORS

6th generation Intel® Core™ processors are based on Skylake microarchitecture and have a CPUID Signature DisplayFamily_DisplayModel value of 06_4EH or 06_5EH.

The Intel® Xeon® Scalable Processor Family based on the Skylake microarchitecture, the 2nd generation Intel® Xeon® Scalable Processor Family based on the Cascade Lake product, and the 3rd generation Intel® Xeon® Scalable Processor Family based on the Cooper Lake product all have a CPUID Signature DisplayFamily_DisplayModel value of 06_55H.

7th generation Intel® Core™ processors are based on the Kaby Lake microarchitecture, 8th generation and 9th generation Intel® Core™ processors, and Intel® Xeon® E processors are based on Coffee Lake microarchitecture; these processors have a CPUID Signature DisplayFamily_DisplayModel value of 06_8EH or 06_9EH.

8th generation Intel® Core™ i3 processors are based on Cannon Lake microarchitecture and have a CPUID Signature DisplayFamily_DisplayModel value of 06_A5H or 06_A6H.

10th generation Intel® Core™ processors are based on Comet Lake microarchitecture (with a CPUID Signature DisplayFamily_DisplayModel value of 06_A5H or 06_A6H) and Ice Lake microarchitecture (with a CPUID Signature DisplayFamily_DisplayModel value of 06_7DH or 06_7EH).
11th generation Intel® Core™ processors are based on Tiger Lake microarchitecture and have a CPUID Signature DisplayFamily_DisplayModel value of 06_8CH or 06_8DH.

The 3rd generation Intel® Xeon® Scalable Processor Family is based on Ice Lake microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_6AH or 06_6CH.

12th generation Intel® Core™ processors supporting the Alder Lake performance hybrid architecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_97H or 06_9AH.

13th generation Intel® Core™ processors supporting the Raptor Lake performance hybrid architecture have a CPUID Signature DisplayFamily_DisplayModel value of 06_BAH, 06_B7H, or 06_BFH.

The 4th generation Intel® Xeon® Scalable Processor Family is based on Sapphire Rapids microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_8FH.

The 5th generation Intel® Xeon® Scalable Processor Family is based on Emerald Rapids microarchitecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_CFH.

The Intel® Core™ Ultra 7 processor is based on Meteor Lake hybrid architecture and has a CPUID Signature DisplayFamily_DisplayModel value of 06_AAH.

These processors support the MSR interfaces listed in Table 2-20, Table 2-21, Table 2-25, Table 2-29, Table 2-35, and Table 2-39. For an MSR listed in Table 2-39 that also appears in the model-specific tables of prior generations, Table 2-39 supersedes prior generation tables.

Tables 2-40 through 2-52 list additional supported MSR interfaces for specific processors; see each table for additional details.

The notation of “Platform” in the Scope column (with respect to MSR_PLATFORM_ENERGY_COUNTER and MSR_PLATFORM_POWER_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor’s implementation.

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td>Control Features in Intel 64 Processor (R/W)</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>Control Features in Intel 64 Processor (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td>IA32_MTRRCAP</td>
<td>MTRR Capability (R/O, Architectural)</td>
<td>See Table 2-2.</td>
<td>Thread</td>
</tr>
<tr>
<td>MTRR Capability (R/O, Architectural)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td>Thermal Monitor Status (R/W)</td>
<td>See Table 2-2.</td>
<td>Core</td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Thermal Status (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status Log (R/WC0)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PROTCHOT # or FORCEPR# Status (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. MSRs at the following addresses are not supported in the 12th generation Intel Core processor E-core: 3F7H. MSRs at the following addresses are not supported in the 12th generation Intel Core processor E-core or P-core: 652H, 653H, 655H, 656H, DB0H, DB1H, DB2H, and D90H.
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
</table>
| 3 | MSR_TURBO_RATIO_LIMIT | Maximum Ratio Limit of Turbo Mode  
R/O if MSR_PLATFORM_INFO.[28] = 0, and R/W if MSR_PLATFORM_INFO.[28] = 1  
Package |
| 4 | PROCHOT # or FORCEPR# Log (R/WC0)  
See Table 2-2. |
| 5 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 6 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 7 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 8 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 9 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 10 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 11 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 12 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 13 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 14 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 15 | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |
| 22:16 | Digital Readout (R/O)  
See Table 2-2. |
| 26:23 | Reserved. |
| 30:27 | Resolution in Degrees Celsius (R/O)  
See Table 2-2. |
| 31 | Reading Valid (R/O)  
See Table 2-2. |
| 63:32 | Reserved. |
Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Maximum Ratio Limit for 1C</td>
<td>Maximum turbo ratio limit of 1 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for 2C</td>
<td>Maximum turbo ratio limit of 2 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Maximum Ratio Limit for 3C</td>
<td>Maximum turbo ratio limit of 3 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Maximum Ratio Limit for 4C</td>
<td>Maximum turbo ratio limit of 4 core active.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C9H, 457</td>
<td>MSR_LASTBRANCH_TOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1FCH, 508</td>
<td>MSR_POWER_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C1E Enable (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>18:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Disable Energy Efficiency Optimization (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Disable Race to Halt Optimization (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:21</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300H, 768</td>
<td>MSR_SGXOWNEREPOCH0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>301H, 769</td>
<td>MSR_SGXOWNEREPOCH1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last Branch Record Stack TOS (R/W)
Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.

Register Address: 1C9H, 457
MSR_LASTBRANCH_TOS

Last Branch Record Stack TOS (R/W)
Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.

Register Address: 1FCH, 508
MSR_POWER_CTL

Power Control Register

Register Address: 300H, 768
MSR_SGXOWNEREPOCH0

Lower 64 Bit CR_SGXOWNEREPOCH (W)
Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.

Register Address: 301H, 769
MSR_SGXOWNEREPOCH1
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>Upper 64 Bit CR_SGXOWNEREPOCH (w)</td>
<td></td>
</tr>
<tr>
<td>Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.</td>
</tr>
<tr>
<td>Register Address: 38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
</tr>
</tbody>
</table>

See Table 2-2 and Section 20.2.4, “Architectural Performance Monitoring Version 4.”

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>0</td>
<td>Ovf_PMC0</td>
</tr>
<tr>
<td>1</td>
<td>Ovf_PMC1</td>
</tr>
<tr>
<td>2</td>
<td>Ovf_PMC2</td>
</tr>
<tr>
<td>3</td>
<td>Ovf_PMC3</td>
</tr>
<tr>
<td>4</td>
<td>Ovf_PMC4 (if CPUID.0AH:EAX[15:8] &gt; 4)</td>
</tr>
<tr>
<td>5</td>
<td>Ovf_PMC5 (if CPUID.0AH:EAX[15:8] &gt; 5)</td>
</tr>
<tr>
<td>6</td>
<td>Ovf_PMC6 (if CPUID.0AH:EAX[15:8] &gt; 6)</td>
</tr>
<tr>
<td>7</td>
<td>Ovf_PMC7 (if CPUID.0AH:EAX[15:8] &gt; 7)</td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td>Ovf_FixedCtr0</td>
</tr>
<tr>
<td>33</td>
<td>Ovf_FixedCtr1</td>
</tr>
<tr>
<td>34</td>
<td>Ovf_FixedCtr2</td>
</tr>
<tr>
<td>54:35</td>
<td>Reserved</td>
</tr>
<tr>
<td>55</td>
<td>Trace_ToPA_PMI</td>
</tr>
<tr>
<td>57:56</td>
<td>Reserved</td>
</tr>
<tr>
<td>58</td>
<td>LBR_Frz</td>
</tr>
<tr>
<td>59</td>
<td>CTR_Frz</td>
</tr>
<tr>
<td>60</td>
<td>ASCI</td>
</tr>
<tr>
<td>61</td>
<td>Ovf_Uncore</td>
</tr>
<tr>
<td>62</td>
<td>Ovf_BufDSSAVE</td>
</tr>
<tr>
<td>63</td>
<td>CondChgd</td>
</tr>
</tbody>
</table>

See Table 2-2 and Section 20.2.4, “Architectural Performance Monitoring Version 4.”
Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] &gt; 7).</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Set 1 to clear Ovf_FixedCtr0.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Set 1 to clear Ovf_FixedCtr1.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Set 1 to clear Ovf_FixedCtr2.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54:35</td>
<td>Reserved.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>Set 1 to clear Trace_ToPA_PMI.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57:56</td>
<td>Reserved.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Set 1 to clear LBR_Frz.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>Set 1 to clear CTR_Frz.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>Set 1 to clear ASCI.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Set 1 to clear Ovf_Uncore.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Set 1 to clear Ovf_BufDSSAVE.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Set 1 to clear CondChgd.</td>
<td>Thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 2-2 and Section 20.2.4, “Architectural Performance Monitoring Version 4.”
Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Reserved.</td>
<td>63 Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 392H, 914</td>
<td>IA32_PERF_GLOBAL_INUSE</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F7H, 1015</td>
<td>MSR_PEBS_FRONTEND</td>
<td>FrontEnd Precise Event Condition Select (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Event Code Select</td>
<td>2:0 Event Code Select</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td>3 Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Event Code Select High</td>
<td>4 Event Code Select High</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:5</td>
<td>Reserved</td>
<td>7:5 Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:8</td>
<td>IDQ_Bubble_Length Specifier</td>
<td>19:8 IDQ_Bubble_Length Specifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:20</td>
<td>IDQ_Bubble_Width Specifier</td>
<td>22:20 IDQ_Bubble_Width Specifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved</td>
<td>63:23 Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 500H, 1280</td>
<td>IA32_SGX_SVN_STATUS</td>
<td>Status and SVN Threshold of SGX Support for ACM (R/O)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>Lock</td>
<td>0 Lock</td>
<td>See Section 39.11.3, “Interactions with Authenticated Code Modules (ACMs).”</td>
<td></td>
</tr>
<tr>
<td>15:1</td>
<td>Reserved</td>
<td>15:1 Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved</td>
<td>63:24 Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 560H, 1376</td>
<td>IA32_RTIT_OUTPUT_BASE</td>
<td>Trace Output Base Register (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 561H, 1377</td>
<td>IA32_RTIT_OUTPUT_MASK_PTRS</td>
<td>Trace Output Mask Pointers Register (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 570H, 1392</td>
<td>IA32_RTIT_CTL</td>
<td>Trace Control Register (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>TraceEn</td>
<td>0 TraceEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CYCEn</td>
<td>1 CYCEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td>2 OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>User</td>
<td>3 User</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved, must be zero.</td>
<td>6:4 Reserved, must be zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CR3Filter</td>
<td>7 CR3Filter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>8</td>
<td>ToPA</td>
</tr>
<tr>
<td>9</td>
<td>MTCEn</td>
</tr>
<tr>
<td>10</td>
<td>TSCEn</td>
</tr>
<tr>
<td>11</td>
<td>DisRETC</td>
</tr>
<tr>
<td>12</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td>13</td>
<td>BranchEn</td>
</tr>
<tr>
<td>17:14</td>
<td>MTCFreq</td>
</tr>
<tr>
<td>18</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td>22:19</td>
<td>CycThresh</td>
</tr>
<tr>
<td>23</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td>27:24</td>
<td>PSBFreq</td>
</tr>
<tr>
<td>31:28</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td>35:32</td>
<td>ADDRO_CFG</td>
</tr>
<tr>
<td>39:36</td>
<td>ADDR1_CFG</td>
</tr>
<tr>
<td>63:40</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td><strong>Register Address: 571H, 1393</strong></td>
<td><strong>IA32_RTIT_STATUS</strong></td>
</tr>
<tr>
<td><strong>Tracing Status Register (R/w)</strong></td>
<td><strong>Thread</strong></td>
</tr>
<tr>
<td>0</td>
<td>FilterEn, writes ignored.</td>
</tr>
<tr>
<td>1</td>
<td>ContextEn, writes ignored.</td>
</tr>
<tr>
<td>2</td>
<td>TriggerEn, writes ignored.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Error (R/W)</td>
</tr>
<tr>
<td>5</td>
<td>Stopped</td>
</tr>
<tr>
<td>31:6</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td>48:32</td>
<td>PacketByteCnt</td>
</tr>
<tr>
<td>63:49</td>
<td>Reserved, must be zero.</td>
</tr>
<tr>
<td><strong>Register Address: 572H, 1394</strong></td>
<td><strong>IA32_RTIT_CR3_MATCH</strong></td>
</tr>
<tr>
<td><strong>Trace Filter CR3 Match Register (R/w)</strong></td>
<td><strong>Thread</strong></td>
</tr>
<tr>
<td>4:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>63:5</td>
<td>CR3[63:5] value to match</td>
</tr>
<tr>
<td><strong>Register Address: 580H, 1408</strong></td>
<td><strong>IA32_RTIT_ADDRO_A</strong></td>
</tr>
<tr>
<td><strong>Region 0 Start Address (R/w)</strong></td>
<td><strong>Thread</strong></td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td><strong>Register Address: 581H, 1409</strong></td>
<td><strong>IA32_RTIT_ADDRO_B</strong></td>
</tr>
<tr>
<td><strong>Region 0 End Address (R/w)</strong></td>
<td><strong>Thread</strong></td>
</tr>
</tbody>
</table>
Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 582H, 1410</td>
<td>IA32_RTIT_ADDR1_A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region 1 Start Address (R/W)</td>
<td></td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 583H, 1411</td>
<td>IA32_RTIT_ADDR1_B</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Region 1 End Address (R/W)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 639H, 1593</td>
<td>MSR_PP0_ENERGY_STATUS</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>PP0 Energy Status (R/O)</td>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64DH, 1613</td>
<td>MSR_PLATFORM_ENERGY_COUNTER</td>
<td>Platform Energy Counter (R/O)</td>
<td>This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.</td>
<td>Platform</td>
</tr>
<tr>
<td>Platform Energy Counter (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.ENERGY_STATUS.Unit.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64EH, 1614</td>
<td>MSR_PPERF</td>
<td>Productive Performance Count (R/O)</td>
<td>Hardware’s view of workload scalability. See Section 15.4.5.1.</td>
<td>Thread</td>
</tr>
<tr>
<td>Productive Performance Count (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Hardware’s view of workload scalability. See Section 15.4.5.1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 64FH, 1615</td>
<td>MSR_CORE_PERF_LIMIT_REASONS</td>
<td>Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)</td>
<td>Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)</td>
<td>Package</td>
</tr>
<tr>
<td>Indicator of Frequency Clipping in Processor Cores (R/W) (Frequency refers to processor core frequency.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>PROCHOT Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thermal Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal event.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Residency State Regulation Status (R0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Residency State Regulation Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to residency state regulation limit.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Running Average Thermal Limit Status (R0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Running Average Thermal Limit Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VR Therm Design Current Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to VR thermal design current limit.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Other Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to electrical or other constraints.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Package/Platform-Level Power Limiting PL1 Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Package/Platform-Level PL2 Power Limiting Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Max Turbo Limit Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to multi-core turbo limits.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Turbo Transition Attenuation Status (R0)</td>
<td>When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.</td>
<td></td>
</tr>
<tr>
<td>15:14</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>19:18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Residency State Regulation Log</td>
<td>When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Running Average Thermal Limit Log</td>
<td>When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>22</td>
<td>VR Therm Alert Log</td>
</tr>
<tr>
<td>23</td>
<td>VR Thermal Design Current Log</td>
</tr>
<tr>
<td>24</td>
<td>Other Log</td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
</tr>
<tr>
<td>26</td>
<td>Package/Platform-Level PL1 Power Limiting Log</td>
</tr>
<tr>
<td>27</td>
<td>Package/Platform-Level PL2 Power Limiting Log</td>
</tr>
<tr>
<td>28</td>
<td>Max Turbo Limit Log</td>
</tr>
<tr>
<td>29</td>
<td>Turbo Transition Attenuation Log</td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

**HDC Configuration (R/W)**

| Register Address: 652H, 1618 | MSR_PKG_HDC_CONFIG |

**Register Information / Bit Fields**

<table>
<thead>
<tr>
<th>2:0</th>
<th>PKG_Cx_Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY.</td>
<td></td>
</tr>
</tbody>
</table>

**Core HDC Idle Residency (R/O)**

| Register Address: 653H, 1619 | MSR_CORE_HDC_RESIDENCY |

| 63:0 | Core_Cx_Duty_Cycle_Cnt |

**Register Address: 655H, 1621 | MSR_PKG_HDC_SHALLOW_RESIDENCY**
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Pkg_C2_Duty_Cycle_Cnt</td>
<td>Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 656H, 1622</td>
<td>MSR_PKG_HDC_DEEP_RESIDENCY</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Package Cx HDC Idle Residency (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Pkg_Cx_Duty_Cycle_Cnt</td>
<td>Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 658H, 1624</td>
<td>MSR_WEIGHTED_CORE_C0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Core-count Weighted C0 Residency (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td></td>
<td>Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 659H, 1625</td>
<td>MSR_ANY_GFXE_C0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Any Core C0 Residency (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>MSR_CORE_GFXE_OVERLAP_C0</td>
<td>Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 65AH, 1626</td>
<td>MSR_ANY_GFXE_C0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Any Graphics Engine C0 Residency (R/O)</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>MSR_PLATFORM_POWER_LIMIT</td>
<td>Platform</td>
<td></td>
</tr>
<tr>
<td>Platform Power Limit Control (R/W-L)</td>
<td>Platform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.</td>
<td>Platform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:0</td>
<td>Platform Power Limit #1</td>
<td>Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Enable Platform Power Limit #1</td>
<td>When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #1 over the time window specified by Power Limit #1 Time Window.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Platform Clamping Limitation #1</td>
<td>When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #1 value. This bit is writeable only when CPUID (EAX=6):EAX[4] is set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:17</td>
<td>Time Window for Platform Power Limit #1</td>
<td>Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation: ( \text{Time Window} = (\text{float}) \left( (1+(X/4))*2^Y \right) ), where: ( X = \text{POWER_LIMIT_1_TIME}[23:22] ) ( Y = \text{POWER_LIMIT_1.Time}[21:17] ) The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is ODH, and the unit is specified in MSR_RAPL_POWER_UNIT[Time Unit].</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46:32</td>
<td>Platform Power Limit #2</td>
<td>Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e., Platform Power Limit # 1).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Enable Platform Power Limit #2</td>
<td>When set, enables the processor to apply control policy such that the platform power does not exceed Platform power limit #2 over the Short Duration time window.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Platform Clamping Limitation #2</td>
<td>When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62:49</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>Lock. Setting this bit will lock all other bits of this MSR until system RESET.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Thread**

**Register Address: 690H, 1680**  
MSR_LASTBRANCH_16_FROM_IP

**Last Branch Record 16 From IP (R/W)**  
One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also:  
• Last Branch Record Stack TOS at 1C9H.  
• Section 18.12.

**Register Address: 691H, 1681**  
MSR_LASTBRANCH_17_FROM_IP

**Last Branch Record 17 From IP (R/W)**  
See description of MSR_LASTBRANCH_0_FROM_IP.

**Register Address: 692H, 1682**  
MSR_LASTBRANCH_18_FROM_IP
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>693H, 1683</td>
<td>MSR_LASTBRANCH_19_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>694H, 1684</td>
<td>MSR_LASTBRANCH_20_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>695H, 1685</td>
<td>MSR_LASTBRANCH_21_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>696H, 1686</td>
<td>MSR_LASTBRANCH_22_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>697H, 1687</td>
<td>MSR_LASTBRANCH_23_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>698H, 1688</td>
<td>MSR_LASTBRANCH_24_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>699H, 1689</td>
<td>MSR_LASTBRANCH_25_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>69AH, 1690</td>
<td>MSR_LASTBRANCH_26_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>69BH, 1691</td>
<td>MSR_LASTBRANCH_27_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>69CH, 1692</td>
<td>MSR_LASTBRANCH_28_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>69DH, 1693</td>
<td>MSR_LASTBRANCH_29_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>69EH, 1694</td>
<td>MSR_LASTBRANCH_30_FROM_IP</td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 69FH, 1695</strong></td>
<td>MSR_LASTBRANCH_31.From_IP</td>
<td><strong>Bit Description</strong></td>
<td><strong>Scope</strong></td>
</tr>
<tr>
<td>Last Branch Record 31 From IP (R/W)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0.From_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 6B0H, 1712</strong></td>
<td>MSR_GRAPHICS_PERF_LIMIT_REASONS</td>
<td><strong>Bit Description</strong></td>
<td><strong>Scope</strong></td>
</tr>
<tr>
<td>Indicator of Frequency Clipping in the Processor Graphics (R/W)</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>(Frequency refers to processor graphics frequency.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td>When set, frequency is reduced due to assertion of external PROCHOT.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td>When set, frequency is reduced due to a thermal event.</td>
<td></td>
</tr>
<tr>
<td>4:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Running Average Thermal Limit Status (R0)</td>
<td>When set, frequency is reduced due to running average thermal limit.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VR Thermal Design Current Status (R0)</td>
<td>When set, frequency is reduced due to VR TDC limit.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Other Status (R0)</td>
<td>When set, frequency is reduced due to electrical or other constraints.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Package/Platform-Level Power Limiting PL1 Status (R0)</td>
<td>When set, frequency is reduced due to package/platform-level power limiting PL1.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Package/Platform-Level PL2 Power Limiting Status (R0)</td>
<td>When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Inefficient Operation Status (R0)</td>
<td>When set, processor graphics frequency is operating below target frequency.</td>
<td></td>
</tr>
<tr>
<td>15:13</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>20:18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>21</td>
<td>Running Average Thermal Limit Log</td>
</tr>
<tr>
<td>22</td>
<td>VR Therm Alert Log</td>
</tr>
<tr>
<td>23</td>
<td>VR Thermal Design Current Log</td>
</tr>
<tr>
<td>24</td>
<td>Other Log</td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
</tr>
<tr>
<td>26</td>
<td>Package/Platform-Level PL1 Power Limiting Log</td>
</tr>
<tr>
<td>27</td>
<td>Package/Platform-Level PL2 Power Limiting Log</td>
</tr>
<tr>
<td>28</td>
<td>Inefficient Operation Log</td>
</tr>
<tr>
<td>63:29</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 6B1H, 1713  
**MSR_RING_PERF_LIMIT_REASONS**  
Indicator of Frequency Clipping in the Ring Interconnect (R/W)  
(Frequency refers to ring interconnect in the uncore.)  
Scope: Package  

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PROCHOT Status (RO)</td>
<td>When set, frequency is reduced due to assertion of external PROCHOT.</td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (RO)</td>
<td>When set, frequency is reduced due to a thermal event.</td>
</tr>
<tr>
<td>4:2</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Running Average Thermal Limit Status (RO)</td>
<td>When set, frequency is reduced due to running average thermal limit.</td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name *Bit Description**</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0x06 (10) VR Therm Alert Status (R0)</td>
<td>When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0x07 (11) VR Thermal Design Current Status (R0)</td>
<td>When set, frequency is reduced due to VR TDC limit.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0x08 (12) Other Status (R0)</td>
<td>When set, frequency is reduced due to electrical or other constraints.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0x09 (13) Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0x0A (14) Package/Platform-Level Power Limiting PL1 Status (R0)</td>
<td>When set, frequency is reduced due to package/Platform-level power limiting PL1.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0x0B (15) Package/Platform-Level PL2 Power Limiting Status (R0)</td>
<td>When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>0x0F-0D (25:18) Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0x0E (17) PROCHOT Log</td>
<td>When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0x0F (18) Thermal Log</td>
<td>When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>20:18</td>
<td>0x12-0C (20:18) Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0x13 (19) Running Average Thermal Limit Log</td>
<td>When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0x14 (20) VR Therm Alert Log</td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0x15 (21) VR Thermal Design Current Log</td>
<td>When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0x16 (22) Other Log</td>
<td>When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>0x17 (23) Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th–13th Generation Intel® Core™ Processors, 1st–5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>26</td>
<td>Package/Platform-Level PL1 Power Limiting Log</td>
</tr>
<tr>
<td>27</td>
<td>Package/Platform-Level PL2 Power Limiting Log</td>
</tr>
<tr>
<td>63:28</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 6D0H, 1744</td>
<td>MSR_LASTBRANCH_16_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 16 To IP (R/W)</td>
<td>One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction. See also:</td>
</tr>
<tr>
<td></td>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
</tr>
<tr>
<td></td>
<td>• Section 18.12.</td>
</tr>
<tr>
<td>Register Address: 6D1H, 1745</td>
<td>MSR_LASTBRANCH_17_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 17 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D2H, 1746</td>
<td>MSR_LASTBRANCH_18_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 18 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D3H, 1747</td>
<td>MSR_LASTBRANCH_19_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 19 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D4H, 1748</td>
<td>MSR_LASTBRANCH_20_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 20 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D5H, 1749</td>
<td>MSR_LASTBRANCH_21_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 21 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D6H, 1750</td>
<td>MSR_LASTBRANCH_22_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 22 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D7H, 1751</td>
<td>MSR_LASTBRANCH_23_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 23 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
<tr>
<td>Register Address: 6D8H, 1752</td>
<td>MSR_LASTBRANCH_24_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record 24 To IP (R/W)</td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
</tr>
</tbody>
</table>
### MODEL-SPECIFIC REGISTERS (MSRs)

**Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 6D9H, 1753</td>
<td>MSR_LASTBRANCH_25_TO_IP</td>
<td>Last Branch Record 25 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DAH, 1754</td>
<td>MSR_LASTBRANCH_26_TO_IP</td>
<td>Last Branch Record 26 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DBH, 1755</td>
<td>MSR_LASTBRANCH_27_TO_IP</td>
<td>Last Branch Record 27 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DCH, 1756</td>
<td>MSR_LASTBRANCH_28_TO_IP</td>
<td>Last Branch Record 28 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DDH, 1757</td>
<td>MSR_LASTBRANCH_29_TO_IP</td>
<td>Last Branch Record 29 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DEH, 1758</td>
<td>MSR_LASTBRANCH_30_TO_IP</td>
<td>Last Branch Record 30 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6DFH, 1759</td>
<td>MSR_LASTBRANCH_31_TO_IP</td>
<td>Last Branch Record 31 To IP (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See description of MSR_LASTBRANCH_0_TO_IP.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 770H, 1904</td>
<td>IA32_PM_ENABLE</td>
<td>See Section 15.4.2, “Enabling HWP.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 771H, 1905</td>
<td>IA32_HWP_CAPABILITIES</td>
<td>See Section 15.4.3, “HWP Performance Range and Dynamic Capabilities.”</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 772H, 1906</td>
<td>IA32_HWP_REQUEST_PKG</td>
<td>See Section 15.4.4, “Managing HWP.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 773H, 1907</td>
<td>IA32_HWP_INTERRUPT</td>
<td>See Section 15.4.6, “HWP Notifications.”</td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 774H, 1908</td>
<td>IA32_HWP_REQUEST</td>
<td>See Section 15.4.4, “Managing HWP.”</td>
<td>Thread</td>
</tr>
</tbody>
</table>

- **7:0** Minimum Performance (R/w)
- **15:8** Maximum Performance (R/w)
- **23:16** Desired Performance (R/w)
- **31:24** Energy/Performance Preference (R/w)
- **41:32** Activity Window (R/w)
- **42** Package Control (R/w)
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:43</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>777H, 1911</td>
<td>IA32_HWP_STATUS</td>
<td>See Section 15.4.5, “HWP Feedback.”</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>D90H, 3472</td>
<td>IA32_BNDCFGS</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DA0H, 3488</td>
<td>IA32_XSS</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DB0H, 3504</td>
<td>IA32_PKG_HDC_CTL</td>
<td>See Section 15.5.2, “Package level Enabling HDC.”</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>DB1H, 3505</td>
<td>IA32_PM_CTL1</td>
<td>See Section 15.5.3, “Logical-Processor Level HDC Control.”</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DB2H, 3506</td>
<td>IA32_THREAD_STALL</td>
<td>See Section 15.5.4.1, “IA32_THREAD_STALL.”</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC0H, 3520</td>
<td>MSR_LBR_INFO_0</td>
<td>Last Branch Record 0 Additional Information (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC1H, 3521</td>
<td>MSR_LBR_INFO_1</td>
<td>One of 32 triplet of last branch record registers on the last branch record stack. This part of the stack contains flag, TSX-related and elapsed cycle information. See also:</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC2H, 3522</td>
<td>MSR_LBR_INFO_2</td>
<td>• Last Branch Record Stack TOS at 1C9H.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC3H, 3523</td>
<td>MSR_LBR_INFO_3</td>
<td>• Section 18.9.1, “LBR Stack.”</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC4H, 3524</td>
<td>MSR_LBR_INFO_4</td>
<td>Last Branch Record 3 Additional Information (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC5H, 3525</td>
<td>MSR_LBR_INFO_5</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC6H, 3526</td>
<td>MSR_LBR_INFO_6</td>
<td>Last Branch Record 5 Additional Information (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>DC7H, 3527</td>
<td>MSR_LBR_INFO_7</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td>Thread</td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-350 Vol. 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-39. Additional MSRs Supported by the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: DD4H, 3540</td>
<td>MSR_LBR_INFO_20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 20 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DD5H, 3541</td>
<td>MSR_LBR_INFO_21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 21 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DD6H, 3542</td>
<td>MSR_LBR_INFO_22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 22 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DD7H, 3543</td>
<td>MSR_LBR_INFO_23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 23 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DD8H, 3544</td>
<td>MSR_LBR_INFO_24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 24 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DD9H, 3545</td>
<td>MSR_LBR_INFO_25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 25 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DDAH, 3546</td>
<td>MSR_LBR_INFO_26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 26 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DDBH, 3547</td>
<td>MSR_LBR_INFO_27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 27 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DDCH, 3548</td>
<td>MSR_LBR_INFO_28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 28 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DDDH, 3549</td>
<td>MSR_LBR_INFO_29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 29 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DDEH, 3550</td>
<td>MSR_LBR_INFO_30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 30 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: DDFH, 3551</td>
<td>MSR_LBR_INFO_31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 31 Additional Information (R/W)</td>
<td>See description of MSR_LBR_INFO_0.</td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
Table 2-40 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_4EH, 06_5EH, 06_8EH, 06_9EH, or 06_66H.

### Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 394H, 916</td>
<td>MSR_UNC_PERF_FIXED_CTRL</td>
<td>Uncore Fixed Counter Control (R/W) Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>19:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Enable overflow propagation.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Enable counting.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 395H, 917</td>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td>Uncore Fixed Counter Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>43:0</td>
<td>Current count.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:44</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 396H, 918</td>
<td>MSR_UNC_CBO_CONFIG</td>
<td>Uncore C-Box Configuration Information (R/O) Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>Specified the number of C-Box units with programmable counters (including processor cores and processor graphics).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B0H, 946</td>
<td>MSR_UNC_ARB_PERFCTR0</td>
<td>Uncore Arb Unit, Performance Counter 0 Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B1H, 947</td>
<td>MSR_UNC_ARB_PERFCTR1</td>
<td>Uncore Arb Unit, Performance Counter 1 Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B2H, 944</td>
<td>MSR_UNC_ARB_PERFEVTSEL0</td>
<td>Uncore Arb Unit, Counter 0 Event Select MSR Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B3H, 945</td>
<td>MSR_UNC_ARB_PERFEVTSEL1</td>
<td>Uncore Arb Unit, Counter 1 Event Select MSR Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 700H, 1792</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL0</td>
<td>Uncore C-Box 0, Counter 0 Event Select MSR Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 701H, 1793</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL1</td>
<td>Uncore C-Box 0, Counter 1 Event Select MSR Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 706H, 1798</td>
<td>MSR_UNC_CBO_0_PERFCTR0</td>
<td>Uncore C-Box 0, Performance Counter 0 Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 707H, 1799</td>
<td>MSR_UNC_CBO_0_PERFCTR1</td>
<td>Uncore C-Box 0, Performance Counter 1 Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 710H, 1808</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL0</td>
<td>Uncore C-Box 1, Counter 0 Event Select MSR Package</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 711H, 1809</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL1</td>
<td>Uncore C-Box 1, Counter 0 Event Select MSR Package</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-Box 1, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_1_PERFCTR0</td>
<td>Address: 716H, 1814</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 0</td>
<td>MSR_UNC_CBO_1_PERFCTR1</td>
<td>Address: 717H, 1815</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 1</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL0</td>
<td>Address: 720H, 1824</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL1</td>
<td>Address: 721H, 1825</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Performance Counter 0</td>
<td>MSR_UNC_CBO_2_PERFCTR0</td>
<td>Address: 726H, 1830</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Performance Counter 1</td>
<td>MSR_UNC_CBO_2_PERFCTR1</td>
<td>Address: 727H, 1831</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL0</td>
<td>Address: 731H, 1841</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 0</td>
<td>MSR_UNC_CBO_3_PERFCTR0</td>
<td>Address: 736H, 1846</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 1</td>
<td>MSR_UNC_CBO_3_PERFCTR1</td>
<td>Address: 737H, 1847</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Uncore PMU Global Control</td>
<td>MSR_UNC_PERF_GLOBAL_CTRL</td>
<td>Address: E01H, 3585</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Slice 0 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Slice 1 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Slice 2 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Slice 3 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Slice 4 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:5</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Enable all uncore counters.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Enable wake on PMI.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Enable Freezing counter when overflow.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore PMU Main Status</td>
<td>MSR_UNC_PERF_GLOBAL_STATUS</td>
<td>Address: E02H, 3586</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Fixed counter overflowed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>An ARB counter overflowed.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-40. Uncore PMU MSRs Supported by 6th Generation, 7th Generation, and 8th Generation Intel® Core™ Processors, and 8th generation Intel® Core™ i3 Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Scope</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSR_TRACE_HUB_STH_ACPBAR_BASE</td>
<td>NPK Address Used by AET Messages (R/W)</td>
<td>Package</td>
<td>Lock Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If set, this MSR cannot be re-written anymore. Lock bit has to be set in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>order for the AET packets to be directed to NPK MMIO.</td>
</tr>
<tr>
<td>17:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:18</td>
<td>ACPIBAR_BASE_ADDRESS</td>
<td></td>
<td></td>
<td>AET target address in NPK MMIO space.</td>
</tr>
<tr>
<td>Register Address: 1F4H, 500</td>
<td>MSR_PRMRR_PHYS_BASE</td>
<td>Processor Reserved Memory Range Register - Physical Base Control Register (R/W)</td>
<td>Core</td>
<td>MemType</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PRMRR BASE MemType.</td>
</tr>
<tr>
<td>2:0</td>
<td></td>
<td></td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>11:3</td>
<td></td>
<td></td>
<td></td>
<td>Base</td>
</tr>
<tr>
<td>45:12</td>
<td></td>
<td></td>
<td></td>
<td>PRMRR Base Address.</td>
</tr>
<tr>
<td>63:46</td>
<td></td>
<td></td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1F5H, 501</td>
<td>MSR_PRMRR_PHYS_MASK</td>
<td>Processor Reserved Memory Range Register - Physical Mask Control Register (R/W)</td>
<td>Core</td>
<td>Reserved.</td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>Lock</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lock bit for the PRMRR.</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>VLD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Enable bit for the PRMRR.</td>
</tr>
<tr>
<td>45:12</td>
<td></td>
<td></td>
<td></td>
<td>Mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PRMRR MASK bits.</td>
</tr>
</tbody>
</table>

#### 2.17.1 MSRs Introduced in 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

Table 2-41 lists additional MSRs for 7th generation and 8th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_8EH or 06_9EH. For an MSR listed in Table 2-41 that also appears in the model-specific tables of prior generations, Table 2-41 supersedes prior generation tables.

### Table 2-41. Additional MSRs Supported by the 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Scope</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSR_TRACE_HUB_STH_ACPBAR_BASE</td>
<td>NPK Address Used by AET Messages (R/W)</td>
<td>Package</td>
<td>Lock Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If set, this MSR cannot be re-written anymore. Lock bit has to be set in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>order for the AET packets to be directed to NPK MMIO.</td>
</tr>
<tr>
<td>17:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:18</td>
<td>ACPIBAR_BASE_ADDRESS</td>
<td></td>
<td></td>
<td>AET target address in NPK MMIO space.</td>
</tr>
<tr>
<td>Register Address: 1F4H, 500</td>
<td>MSR_PRMRR_PHYS_BASE</td>
<td>Processor Reserved Memory Range Register - Physical Base Control Register (R/W)</td>
<td>Core</td>
<td>MemType</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PRMRR BASE MemType.</td>
</tr>
<tr>
<td>2:0</td>
<td></td>
<td></td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>11:3</td>
<td></td>
<td></td>
<td></td>
<td>Base</td>
</tr>
<tr>
<td>45:12</td>
<td></td>
<td></td>
<td></td>
<td>PRMRR Base Address.</td>
</tr>
<tr>
<td>63:46</td>
<td></td>
<td></td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1F5H, 501</td>
<td>MSR_PRMRR_PHYS_MASK</td>
<td>Processor Reserved Memory Range Register - Physical Mask Control Register (R/W)</td>
<td>Core</td>
<td>Reserved.</td>
</tr>
<tr>
<td>9:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>Lock</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lock bit for the PRMRR.</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>VLD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Enable bit for the PRMRR.</td>
</tr>
<tr>
<td>45:12</td>
<td></td>
<td></td>
<td></td>
<td>Mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PRMRR MASK bits.</td>
</tr>
</tbody>
</table>
### Table 2-41. Additional MSRs Supported by the 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:46</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1FBH, 507</td>
<td>MSR_PRMRR_VALID_CONFIG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid PRMRR Configurations (R/W)</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1M supported MEE size.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>32M supported MEE size.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>64M supported MEE size.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>128M supported MEE size.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2F4H, 756</td>
<td>MSR_UNCORE_PRMRR_PHYS_BASE</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>(R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The PRMRR range is used to protect the processor reserved memory from unauthorized reads and writes. Any IO access to this range is aborted. This register controls the location of the PRMRR range by indicating its starting address. It functions in tandem with the PRMRR mask register.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAWIDTH-1:12</td>
<td>Range Base</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This field corresponds to bits PAWIDTH-1:12 of the base address memory range which is allocated to PRMRR memory.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:PAWIDTH</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2F5H, 757</td>
<td>MSR_UNCORE_PRMRR_PHYS_MASK</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>(R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This register controls the size of the PRMRR range by indicating which address bits must match the PRMRR base register value.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Setting this bit locks all writeable settings in this register, including itself.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Range_En</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicates whether the PRMRR range is enabled and valid.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38:12</td>
<td>Range_Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This field indicates which address bits must match PRMRR base in order to qualify as an PRMRR access.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:39</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 620H, 1568</td>
<td>MSR_RING_RATIO_LIMIT</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Ring Ratio Limit (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This register provides Min/Max Ratio Limits for the LLC and Ring.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:0</td>
<td>MAX_Ratio</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This field is used to limit the max ratio of the LLC/Ring.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.17.2 MSRs Specific to 8th Generation Intel® Core™ i3 Processors

Table 2-42 lists additional MSRs for 8th generation Intel Core i3 processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_66H. For an MSR listed in Table 2-42 that also appears in the model-specific tables of prior generations, Table 2-42 supersede prior generation tables.

Table 2-41. Additional MSRs Supported by the 7th Generation and 8th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture and Coffee Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>0x0</td>
<td>T0.10 Lock (R/WL)</td>
<td>Thread</td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>1</td>
<td>T0.00 Enable VMX Inside SMX Operation (R/WL)</td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>2</td>
<td>T0.01 Enable VMX Outside SMX Operation (R/WL)</td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>14:8</td>
<td>SENTER Local Functions Enables (R/WL)</td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>15</td>
<td>SENTER Global Functions Enable (R/WL)</td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>17</td>
<td>SGX Launch Control Enable (R/WL)</td>
<td></td>
</tr>
<tr>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>18</td>
<td>SGX Global Functions Enable (R/WL)</td>
<td></td>
</tr>
<tr>
<td>MSR_BR_DETECT_CTRL</td>
<td></td>
<td>63:21</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>MSR_BR_DETECT_CTRL</td>
<td></td>
<td>0</td>
<td>EnMonitoring Global enable for branch monitoring.</td>
<td></td>
</tr>
<tr>
<td>MSR_BR_DETECT_CTRL</td>
<td></td>
<td>1</td>
<td>EnExcept Enable branch monitoring event signaling on threshold trip. The branch monitoring event handler is signaled via the existing PMI signaling mechanism as programmed from the corresponding local APIC LVT entry.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. This MSR is specific to 7th generation and 8th generation Intel® Core™ processors.
### Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>EnLBFRz</td>
<td>Enable LBR freeze on threshold trip. This will cause the LBR frozen bit 58 to be set in IA32_PERF_GLOBAL_STATUS when a triggering condition occurs and this bit is enabled.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DisableInGuest</td>
<td>When set to '1', branch monitoring, event triggering and LBR freeze actions are disabled when operating at VMX non-root operation.</td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:8</td>
<td>WindowSize</td>
<td>Window size defined by WindowCntSel. Values 0 – 1023 are supported. Once the Window counter reaches the WindowSize count both the Window Counter and all Branch Monitoring Counters are cleared.</td>
<td></td>
</tr>
<tr>
<td>23:18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25:24</td>
<td>WindowCntSel</td>
<td>Window event count select: '00 = Instructions retired. '01 = Branch instructions retired '10 = Return instructions retired. '11 = Indirect branch instructions retired.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>CntAndMode</td>
<td>When set to '1', the overall branch monitoring event triggering condition is true only if all enabled counters’ threshold conditions are true. When '0', the threshold tripping condition is true if any enabled counters’ threshold is true.</td>
<td></td>
</tr>
<tr>
<td>63:27</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 351H, 849</strong></td>
<td><strong>MSR_BR_DETECT_STATUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch Monitoring Global Status (R/W)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Branch Monitoring Event Signaled</td>
<td>When set to ‘1’, Branch Monitoring event signaling is blocked until this bit is cleared by software.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LBRsValid</td>
<td>This status bit is set to ‘1’ if the LBR state is considered valid for sampling by branch monitoring software.</td>
<td></td>
</tr>
<tr>
<td>7:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CntrHit0</td>
<td>Branch monitoring counter #0 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>CntrHit1</td>
<td>Branch monitoring counter #1 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:10</td>
<td>Reserved.</td>
<td>Reserved for additional branch monitoring counters threshold hit status.</td>
<td></td>
</tr>
<tr>
<td>25:16</td>
<td>CountWindow</td>
<td>The current value of the window counter. The count value is frozen on a valid branch monitoring triggering condition. This is a 10-bit unsigned value.</td>
<td></td>
</tr>
<tr>
<td>31:26</td>
<td>Reserved.</td>
<td>Reserved for future extension of CountWindow.</td>
<td></td>
</tr>
<tr>
<td>39:32</td>
<td>Count0</td>
<td>The current value of counter 0 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit0 will also be set). This is an 8-bit signed value (2’s complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).</td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Count1</td>
<td>The current value of counter 1 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit1 will also be set). This is an 8-bit signed value (2’s complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).</td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 354H–355H, 852–853 | MSR_BR_DETECT_COUNTER_CONFIG_i | Branch Monitoring Detect Counter Configuration (R/W) |

<table>
<thead>
<tr>
<th></th>
<th>CntrEn</th>
<th>Enable counter.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CntrEvSel</td>
<td>Event select (other values #GP)</td>
<td></td>
</tr>
<tr>
<td>7:1</td>
<td></td>
<td>‘00000000 = RETs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘00000001 = RET-CALL bias.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘00000010 = RET mispredicts.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘00000011 = Branch (all) mispredicts.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘00000100 = Indirect branch mispredicts.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘00000101 = Far branch instructions.</td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>CntrThreshold</td>
<td>Threshold (an unsigned value of 0 to 127 supported). The value 0 of counter threshold will result in event signaled after every instruction. #GP if threshold is &lt; 2.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-43 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_66H.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>MispredEventCnt</td>
<td>Mispredict events counting behavior:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>'0 = Mispredict events are counted in a window.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>'1 = Mispredict events are counted based on a consecutive occurrence. CntrThreshold is treated as # of consecutive mispredicts. This control bit only applies to events specified by CntrEvSel that involve a prediction (0000010, 0000011, 0000100). Setting this bit for other events is ignored.</td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F8H, 1016</td>
<td>MSR_PKG_C3_RESIDENCY</td>
<td>Package C3 Residency Counter (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>Package C3 Residency Counter (R/O)</td>
<td></td>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td>Package</td>
</tr>
<tr>
<td>63:0</td>
<td>MAX_Ratio</td>
<td>This field is used to limit the max ratio of the LLC/Ring.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>MIN_Ratio</td>
<td>Writing to this field controls the minimum possible ratio of the LLC/Ring.</td>
<td></td>
</tr>
<tr>
<td>63:15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 660H, 1632</td>
<td>MSR_CORE_C1_RESIDENCY</td>
<td>Core C1 Residency Counter (R/O)</td>
<td>Core</td>
</tr>
<tr>
<td>Core C1 Residency Counter (R/O)</td>
<td></td>
<td>Value since last reset for the Core C1 residency. Counter rate is the Max Non-Turbo frequency (same as TSC). This counter counts in case both of the core’s threads are in an idle state and at least one of the core’s thread residency is in a C1 state or in one of its sub states. The counter is updated only after a core C state exit. Note: Always reads 0 if core C1 is unsupported. A value of zero indicates that this processor does not support core C1 or never entered core C1 level state.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 662H, 1634</td>
<td>MSR_CORE_C3_RESIDENCY</td>
<td>Core C3 Residency Counter (R/O)</td>
<td>Core</td>
</tr>
<tr>
<td>Core C3 Residency Counter (R/O)</td>
<td></td>
<td>Will always return 0.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2-42. Additional MSRs Supported by the 8th Generation Intel® Core™ i3 Processors Based on Cannon Lake Microarchitecture (Contd.)
### Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 394H, 916</td>
<td>MSR_UNC_PERF_FIXED_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Fixed Counter Control (R/W)</td>
<td>19:0 Reserved.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>20</td>
<td>Enable overflow propagation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Enable counting.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 395H, 917</td>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Fixed Counter</td>
<td>47:0 Current count.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 396H, 918</td>
<td>MSR_UNC_CBO_CONFIG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box Configuration Information (R/O)</td>
<td>3:0 Report the number of C-Box units with performance counters, including processor cores and processor graphics.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B0H, 946</td>
<td>MSR_UNC_ARB_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Arb Unit, Performance Counter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B1H, 947</td>
<td>MSR_UNC_ARB_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Arb Unit, Performance Counter 1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B2H, 944</td>
<td>MSR_UNC_ARB_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Arb Unit, Counter 0 Event Select MSR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B3H, 945</td>
<td>MSR_UNC_ARB_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Arb Unit, Counter 1 Event Select MSR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 700H, 1792</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0, Counter 0 Event Select MSR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 701H, 1793</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0, Counter 1 Event Select MSR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 702H, 1794</td>
<td>MSR_UNC_CBO_0_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0, Performance Counter 0</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 703H, 1795</td>
<td>MSR_UNC_CBO_0_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 0, Performance Counter 1</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 708H, 1800</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Counter 0 Event Select MSR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 709H, 1801</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Counter 1 Event Select MSR</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 70AH, 1802</td>
<td>MSR_UNC_CBO_1_PERFCTR0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-Box 1, Performance Counter 0</td>
<td>MSR_UNC_CBO_1_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 70BH, 1803</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 1, Performance Counter 1</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 710H, 1808</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_2_PERFCTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 711H, 1809</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 712H, 1810</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Performance Counter 0</td>
<td>MSR_UNC_CBO_2_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 713H, 1811</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 2, Performance Counter 1</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 718H, 1816</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_3_PERFCTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 719H, 1817</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 71AH, 1818</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 0</td>
<td>MSR_UNC_CBO_3_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 71BH, 1819</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 3, Performance Counter 1</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 720H, 1824</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_4_PERFCTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 721H, 1825</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 722H, 1826</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 0</td>
<td>MSR_UNC_CBO_4_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 723H, 1827</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 1</td>
<td>MSR_UNC_CBO_5_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 728H, 1832</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Counter 0 Event Select MSR</td>
<td>MSR_UNC_CBO_5_PERFCTR0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 729H, 1833</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Counter 1 Event Select MSR</td>
<td>MSR_UNC_CBO_5_PERFEVTSEL1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 72AH, 1834</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Performance Counter 0</td>
<td>MSR_UNC_CBO_5_PERFCTR1</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 72BH, 1835</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Performance Counter 1</td>
<td>MSR_UNC_CBO_6_PERFEVTSEL0</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 730H, 1840</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6, Counter 0 Event Select MSR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-43. Uncore PMU MSRs Supported by Intel® Core™ Processors Based on Cannon Lake Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 731H, 1841</td>
<td>MSR_UNC_CBO_6_PERFEVTSEL1</td>
<td>Uncore C-Box 6, Counter 1 Event Select MSR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 732H, 1842</td>
<td>MSR_UNC_CBO_6_PERFCTR0</td>
<td>Uncore C-Box 6, Performance Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 733H, 1843</td>
<td>MSR_UNC_CBO_6_PERFCTR1</td>
<td>Uncore C-Box 6, Performance Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 738H, 1848</td>
<td>MSR_UNC_CBO_7_PERFEVTSEL0</td>
<td>Uncore C-Box 7, Counter 0 Event Select MSR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 739H, 1849</td>
<td>MSR_UNC_CBO_7_PERFEVTSEL1</td>
<td>Uncore C-Box 7, Counter 1 Event Select MSR</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 73AH, 1850</td>
<td>MSR_UNC_CBO_7_PERFCTR0</td>
<td>Uncore C-Box 7, Performance Counter 0</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 73BH, 1851</td>
<td>MSR_UNC_CBO_7_PERFCTR1</td>
<td>Uncore C-Box 7, Performance Counter 1</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: E01H, 3585</td>
<td>MSR_UNC_PERF_GLOBAL_CTRL</td>
<td>Uncore PMU Global Control</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Slice 0 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Slice 1 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Slice 2 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Slice 3 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Slice 4 select.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18:5</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Enable all uncore counters.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Enable wake on PMI.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Enable Freezing counter when overflow.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E02H, 3586</td>
<td>MSR_UNC_PERF_GLOBAL_STATUS</td>
<td>Uncore PMU Main Status</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Fixed counter overflowed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>An ARB counter overflowed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A CBox counter overflowed (on any slice).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2.17.3 MSRs Introduced in 10th Generation Intel® Core™ Processors

Table 2-44 lists additional MSRs for 10th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_7DH or 06_7EH. For an MSR listed in Table 2-44 that also appears in the model-specific tables of prior generations, Table 2-44 supersedes prior generation tables.
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 33H, 51</td>
<td>MSR_MEMORY_CTRL</td>
<td>Memory Control Register</td>
<td>Reserved.</td>
<td>Core</td>
</tr>
<tr>
<td>28:0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>SPLIT_LOCK_DISABLE</td>
<td>If set to 1, a split lock will cause an #AC(0) exception. See Section 9.1.2.3, “Features to Disable Bus Locks.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 48H, 72</td>
<td>IA32_SPEC_CTRL</td>
<td>See Table 2-2.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Register Address: 49H, 73</td>
<td>IA32_PREDICT_CMD</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 8CH, 140</td>
<td>IA32_SGXLEPUBKEYHASH0</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 8DH, 141</td>
<td>IA32_SGXLEPUBKEYHASH1</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 8EH, 142</td>
<td>IA32_SGXLEPUBKEYHASH2</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: 8FH, 143</td>
<td>IA32_SGXLEPUBKEYHASH3</td>
<td>See Table 2-2.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: A0H, 160</td>
<td>MSR_BIOS_MCU_ERRORCODE</td>
<td>BIOS MCU ERRORCODE (R/O)</td>
<td>This MSR indicates if WRMSR 0x79 failed to configure PRM memory and gives a hint to debug BIOS.</td>
<td>Package</td>
</tr>
<tr>
<td>15:0</td>
<td>Error Codes (R/O)</td>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>30:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>MCU Partial Success (R/O)</td>
<td>When set to 1, WRMSR 0x79 skipped part of the functionality during BIOS.</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>Register Address: A5H, 165</td>
<td>MSR_FIT_BIOS_ERROR</td>
<td>FIT BIOS ERROR (R/W)</td>
<td>Report error codes for debug in case the processor failed to parse the Firmware Table in BIOS. Can also be used to log BIOS information.</td>
<td>Thread</td>
</tr>
<tr>
<td>7:0</td>
<td>Error Codes (R/W)</td>
<td>Error codes for debug.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Entry Type (R/W)</td>
<td>Failed FIT entry type.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>FIT MCU Entry (R/W)</td>
<td>FIT contains MCU entry.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>

62:17 | Reserved. | Thread |
63 | LOCK (R/w) When set to 1, writes to this MSR will be skipped. | Thread |

Register Address: 10BH, 267 IA32_FLUSH_CMD

See Table 2-2. Thread

Register Address: 151H, 337 MSR_BIOS_DONE

BIOS Done (R/W)

0 | BIOS Done Indication (R/WO) Set by BIOS when it finishes programming the processor and wants to lock the memory configuration from changes by software that is running on this thread. Writes to the bit will be ignored if EAX[0] is 0. | Thread |
1 | Package BIOS Done Indication (R/O) When set to 1, all threads in the package have bit 0 of this MSR set. | Package |
31:2 | Reserved. | Thread |

Register Address: 1F1H, 497 MSR_CRASHLOG_CONTROL

Write Data to a Crash Log Configuration

0 | CDDIS: CrashDump_Disable If set, indicates that Crash Dump is disabled. | Thread |
63:1 | Reserved. | Thread |

Register Address: 2A0H, 672 MSR_PRMRR_BASE_0

Processor Reserved Memory Range Register - Physical Base Control Register (R/W)

2:0 | MEMTYPE: PRMRR BASE Memory Type. | Core |
3 | CONFIGURED: PRMRR BASE Configured. | Core |
11:4 | Reserved. | Core |
51:12 | BASE: PRMRR Base Address. | Core |
63:52 | Reserved. | Core |

Register Address: 30CH, 780 IA32_FIXED_CTR3

Fixed-Function Performance Counter Register 3 (R/W) Bit definitions are the same as found in IA32_FIXED_CTR0, offset 309H. See Table 2-2.

Register Address: 329H, 809 MSR_PERF_METRICS

Performance Metrics (R/w) Reports metrics directly. Software can check (and/or expose to its guests) the availability of PERF_METRICS feature using IA32_PERF_CAPABILITIES.PERF_METRICS_AVAILABLE (bit 15).

7:0 | Retiring. Percent of utilized slots by uops that eventually retire (commit). | Thread |
15:8 | Bad Speculation. Percent of wasted slots due to incorrect speculation, covering utilized by uops that do not retire, or recovery bubbles (unutilized slots). | Thread |
23:16 | Frontend Bound. Percent of unutilized slots where front-end did not deliver a uop while back-end is ready. | Thread |
### Table 2-44. MSRs Supported by the 10th Generation Intel® Core™ Processors (Ice Lake Microarchitecture) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td></td>
<td>Backend Bound. Percent of unutilized slots where a uop was not delivered to back-end due to lack of back-end resources.</td>
<td></td>
</tr>
<tr>
<td>63:25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F2H, 1010</td>
<td>MSR_PEBS_DATA_CFG</td>
<td>PEBS Data Configuration (R/W) Provides software the capability to select data groups of interest and thus reduce the record size in memory and record generation latency. Hence, a PEBS record’s size and layout vary based on the selected groups. The MSR also allows software to select LBR depth for branch data records.</td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td>Memory Info. Setting this bit will capture memory information such as the linear address, data source and latency of the memory access in the PEBS record.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>GPRs. Setting this bit will capture the contents of the General Purpose registers in the PEBS record.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>XMMs. Setting this bit will capture the contents of the XMM registers in the PEBS record.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LBRs. Setting this bit will capture LBR TO, FROM, and INFO in the PEBS record.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>LBR Entries. Set the field to the desired number of entries - 1. For example, if the LBR_entries field is 0, a single entry will be included in the record. To include 32 LBR entries, set the LBR_entries field to 31 (0x1F). To ensure all PEBS records are 16-byte aligned, software can use LBR_entries that is multiple of 3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 541H, 1345</td>
<td>MSR_CORE_UARCH_CTL Core Microarchitecture Control MSR (R/W)</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>0</td>
<td>L1 Scrubbing Enable When set to 1, enable L1 scrubbing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 657H, 1623</td>
<td>MSR_FAST_UNCORE_MSRS_CTL Fast WRMSR/RDMSR Control MSR (R/W)</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>3:0</td>
<td>FAST_ACCESS_ENABLE: Bit 0: When set to ‘1’, provides a hint for the hardware to enable fast access mode for the IA32_HWP_REQUEST MSR. This bit is sticky and is cleaned by the hardware only during reset time. This bit is valid only if FAST_UNCORE_MSRS_CAPABILITY[0] is set. Setting this bit will cause CPUID[6].EAX[18] to be set.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 65EH, 1630</td>
<td>MSR_FAST_UNCORE_MSRS_STATUS Indication of Uncore MSRs, Post Write Activates</td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
### 2.17.4 MSRs Introduced in the 11th Generation Intel® Core™ Processors based on Tiger Lake Microarchitecture

Table 2-45 lists additional MSRs for 11th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_8CH or 06_8DH. The MSRs listed in Table 2-44 are also supported by these processors. For an MSR listed in Table 2-44 that also appears in the model-specific tables of prior generations, Table 2-45 supersedes prior generation tables.

#### Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: A0H, 160</td>
<td>MSR_BIOS_MCU_ERRORCODE</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>BIOS MCU ERRORCODE (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>Error Codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:16</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: A7H, 167</td>
<td>MSR_BIOS_DEBUG</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>BIOS DEBUG (R/O)</td>
<td>This MSR indicates if WRMSR 79H failed to configure PRM memory and gives a hint to debug BIOS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:0</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>MCU Partial Success</td>
<td>Bit Description</td>
<td>When set to 1, WRMSR 79H skipped part of the functionality during BIOS.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: CFH, 207</td>
<td>IA32_CORE_CAPABILITIES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32 Core Capabilities Register (R/O)</td>
<td></td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>This MSR provides an architectural enumeration function for model-specific behavior.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FUSA_SUPPORTED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RSM_IN_CPL0_ONLY</td>
<td></td>
<td>When set to 1, the RSM instruction is only allowed in CPL0 (#GP triggered in any CPL != 0). When set to 0, then any CPL may execute the RSM instruction.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SPLIT_LOCK_DISABLE_SUPPORTED</td>
<td></td>
<td>When read as 1, software can set bit 29 of MSR_MEMORY_CTRL (MSR address 33H).</td>
<td></td>
</tr>
<tr>
<td>31:6</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 492H, 1170</td>
<td>IA32_VMX_PROCBASED_CTLS3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_VMX_PROCBASED_CTLS3</td>
<td></td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>This MSR enumerates the allowed 1-settings of the third set of processor-based controls. Specifically, VM entry allows bit X of the tertiary processor-based VM-execution controls to be 1 if and only if bit X of the MSR is set to 1. If bit X of the MSR is cleared to 0, VM entry fails if control X and the “activate tertiary controls” primary processor-based VM-execution control are both 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LOADIwKEY</td>
<td></td>
<td>This control determines whether executions of LOADIwKEY cause VM exits.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 601H, 1537</td>
<td>MSR_VR_CURRENT_CONFIG</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Power Limit 4 (PL4)</td>
<td></td>
<td></td>
<td>Package-level maximum power limit (in Watts). It is a proactive, instantaneous limit.</td>
<td></td>
</tr>
<tr>
<td>12:0</td>
<td>PL4 Value</td>
<td></td>
<td>PL4 value in 0.125 A increments. This field is locked by VR_CURRENT_CONFIG[LOCK]. When the LOCK bit is set to 1b, this field becomes Read Only.</td>
<td></td>
</tr>
<tr>
<td>30:13</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lock Indication (LOCK)</td>
<td></td>
<td>This bit will lock the CURRENT_LIMIT settings in this register and will also lock this setting. This means that once set to 1b, the CURRENT_LIMIT setting and this bit become Read Only until the next Warm Reset.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>62:32</td>
<td>Bit Description Not in use.</td>
</tr>
<tr>
<td>63</td>
<td>Bit Description Reserved.</td>
</tr>
<tr>
<td>Register Address: 6A0H, 1696</td>
<td>IA32_U_CET</td>
</tr>
<tr>
<td>Configure User Mode CET (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 6A2H, 1698</td>
<td>IA32_S_CET</td>
</tr>
<tr>
<td>Configure Supervisor Mode CET (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 6A4H, 1700</td>
<td>IA32_PL0_SSP</td>
</tr>
<tr>
<td>Linear address to be loaded into SSP on transition to privilege level 0. (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 6A5H, 1701</td>
<td>IA32_PL1_SSP</td>
</tr>
<tr>
<td>Linear address to be loaded into SSP on transition to privilege level 1. (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 6A6H, 1702</td>
<td>IA32_PL2_SSP</td>
</tr>
<tr>
<td>Linear address to be loaded into SSP on transition to privilege level 2. (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 6A7H, 1703</td>
<td>IA32_PL3_SSP</td>
</tr>
<tr>
<td>Linear address to be loaded into SSP on transition to privilege level 3. (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 6A8H, 1704</td>
<td>IA32_INTERRUPT_SSP_TABLE_ADDR</td>
</tr>
<tr>
<td>Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 981H, 2433</td>
<td>IA32_TME_CAPABILITY</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 982H, 2434</td>
<td>IA32_TME_ACTIVATE</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 983H, 2435</td>
<td>IA32_TME_EXCLUDE_MASK</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 984H, 2436</td>
<td>IA32_TME_EXCLUDE_BASE</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 990H, 2448</td>
<td>IA32_COPY_STATUS(^1)</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 991H, 2449</td>
<td>IA32_IWKEYBACKUP_STATUS(^1)</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C82H, 3202</td>
<td>IA32_L2_QOS_CFG</td>
</tr>
</tbody>
</table>
Table 2-45. Additional MSRs Supported by the 11th Generation Intel® Core™ Processors Based on Tiger Lake Microarchitecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_CR_L2_QOS_CFG</td>
<td>This MSR provides software an enumeration of the parameters that L2 QoS (Intel RDT) support in any particular implementation.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CDP_ENABLE</td>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, it will enable the code and data prioritization for the L2 CAT/Intel RDT feature.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 0, code and data prioritization is disabled for L2 CAT/Intel RDT. See Chapter 18, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features,” for further details on CDP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_L2_QOS_MASK_[0-7]</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 18, “Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:0</td>
<td>WAYS_MASK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Setting a 1 in this bit X allows threads with CLOS &lt;n&gt; (where N is [0-7]) to allocate to way X in the MLC. Ones are only allowed to be written to ways that physically exist in the MLC (CPUID.4.2:EBX[31:22] will indicate this).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Writing a 1 to a value beyond the highest way or a non-contiguous set of 1s will cause a #GP on the WRMSR to this MSR.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_COPY_LOCAL_TO_PLATFORM1</td>
<td>Thread</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_COPY_PLATFORM_TO_LOCAL1</td>
<td>Thread</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. Further details on Key Locker and usage of this MSR can be found here:

2.17.5 MSRs Introduced in the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

Table 2-46 lists additional MSRs for 12th and 13th generation Intel Core processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_97H, 06_9AH, 06_BAH, 06_B7H, or 06_BFH. Table 2-47 lists the MSRs unique to the processor P-core. Table 2-48 lists the MSRs unique to the processor E-core.

The MSRs listed in Table 2-44 and Table 2-45 are also supported by these processors. For an MSR listed in Table 2-46, Table 2-47, or Table 2-48 that also appears in the model-specific tables of prior generations, Table 2-46, Table 2-47, and Table 2-48 supersede prior generation tables.

1. MSRs at the following addresses are not supported in the 12th and 13th generation Intel Core processor E-core: 30CH, 329H, 541H, and 657H. The MSR at address 657H is not supported in the 12th and 13th generation Intel Core processor P-core.
### Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 33H, 51</td>
<td>Memory Control Register</td>
<td>MSR_MEMORY_CTRL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>26:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>UC_STORE_THROTTLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If set to 1, when enabled, the processor will only allow one in-progress UC store at a time.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>UC_LOCK_DISABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If set to 1, a UC lock will cause a #GP(0) exception. See Section 9.1.2.3, “Features to Disable Bus Locks.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>SPLIT_LOCK_DISABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If set to 1, a split lock will cause an #AC(0) exception. See Section 9.1.2.3, “Features to Disable Bus Locks.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: BCH, 188</td>
<td>IA32_MISC_PACKAGE_CTLS</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Power Filtering Control (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_ARCH_CAPABILITIES[bit 10] enumeration support for this MSR. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C7H, 199</td>
<td>IA32_PMC6</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>General Performance Counter 6 (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C8H, 200</td>
<td>IA32_PMC7</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>General Performance Counter 7 (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: CFH, 207</td>
<td>IA32_CORE_CAPABILITIES</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>IA32 Core Capabilities Register (R/O)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This MSR provides an architectural enumeration function for model-specific behavior.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>STL_QOS_SUPPORTED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, the STLB QoS feature is supported and the STLB QoS MSRs (1A8FH -1A97H) are accessible. When set to 0, access to these MSRs will #GP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FUSA_SUPPORTED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RSM_IN_CPL0_ONLY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set to 1, the RSM instruction is only allowed in CPL0 (#GP triggered in any CPL != 0). When set to 0, then any CPL may execute the RSM instruction.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>UC_LOCK_DISABLE_SUPPORTED</td>
<td>When read as 1, software can set bit 28 of MSR_MEMORY_CTRL (MSR address 33H).</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>SPLIT_LOCK_DISABLE_SUPPORTED</td>
<td>When read as 1, software can set bit 29 of MSR_MEMORY_CTRL.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>SNOOP_FILTER_QOS_SUPPORTED</td>
<td>When set to 1, the Snoop Filter Qos Mask MSRs are supported. When set to 0, access to these MSRs will #GP.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>UC_STORE_THROTTLING_SUPPORTED</td>
<td>When set 1, UC Store throttle capability exist through MSR_MEMORY_CTRL (33H) bit 27.</td>
<td></td>
</tr>
<tr>
<td>31:8</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1EH, 225</td>
<td>IA32_UMWAIT_CONTROL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10AH, 266</td>
<td>IA32_ARCH_CAPABILITIES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18CH, 396</td>
<td>IA32_PERFEVTSEL6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18DH, 397</td>
<td>IA32_PERFEVTSEL7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195H, 405</td>
<td>IA32_OVERCLOCKING_STATUS</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>1ADH, 429</td>
<td>MSR_PRIMARY_TURBO_RATIO_LIMIT</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>

#### Description
- **Register Address: E1H, 225**
  - **IA32_UMWAIT_CONTROL**
  - UMWAIT Control (R/W)
  - See Table 2-2.

- **Register Address: 10AH, 266**
  - **IA32_ARCH_CAPABILITIES**
  - Enumeration of Architectural Features (R/O)
  - See Table 2-2.

- **Register Address: 18CH, 396**
  - **IA32_PERFEVTSEL6**
  - See Table 2-20.

- **Register Address: 18DH, 397**
  - **IA32_PERFEVTSEL7**
  - See Table 2-20.

- **Register Address: 195H, 405**
  - **IA32_OVERCLOCKING_STATUS**
  - Overclocking Status (R/O)
  - **IA32_ARCH_CAPABILITIES[bit 23]** enumerates support for this MSR. See Table 2-2.

- **Register Address: 1ADH, 429**
  - **MSR_PRIMARY_TURBO_RATIO_LIMIT**
  - Primary Maximum Turbo Ratio Limit (R/W)
  - Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.

| 7:0   | MAX_TURBO_GROUP_0: Maximum turbo ratio limit with 1 core active. |
| 15:8  | MAX_TURBO_GROUP_1: Maximum turbo ratio limit with 2 cores active. |
| 23:16 | MAX_TURBO_GROUP_2: Maximum turbo ratio limit with 3 cores active. |
| 31:24 | MAX_TURBO_GROUP_3: Maximum turbo ratio limit with 4 cores active. |
| 39:32 | MAX_TURBO_GROUP_4: Maximum turbo ratio limit with 5 cores active. |
### Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>47:40</td>
<td>MAX_TURBO_GROUP_5: Maximum turbo ratio limit with 6 cores active.</td>
</tr>
<tr>
<td>55:48</td>
<td>MAX_TURBO_GROUP_6: Maximum turbo ratio limit with 7 cores active.</td>
</tr>
<tr>
<td>63:56</td>
<td>MAX_TURBO_GROUP_7: Maximum turbo ratio limit with 8 cores active.</td>
</tr>
<tr>
<td>Register Address: 493H, 1171</td>
<td>IA32_VMX_EXIT_CTLS2</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C7H, 1223</td>
<td>IA32_A_PMC6</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 4C8H, 1224</td>
<td>IA32_A_PMC7</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 650H, 1616</td>
<td>MSR_SECONDARY_TURBO_RATIO_LIMIT</td>
</tr>
<tr>
<td>Software can configure these limits when MSR_PLATFORM_INFO[28] = 1.</td>
<td>Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.</td>
</tr>
<tr>
<td>7:0</td>
<td>MAX_TURBO_GROUP_0: Maximum turbo ratio limit with 1 core active.</td>
</tr>
<tr>
<td>15:8</td>
<td>MAX_TURBO_GROUP_1: Maximum turbo ratio limit with 2 cores active.</td>
</tr>
<tr>
<td>23:16</td>
<td>MAX_TURBO_GROUP_2: Maximum turbo ratio limit with 3 cores active.</td>
</tr>
<tr>
<td>31:24</td>
<td>MAX_TURBO_GROUP_3: Maximum turbo ratio limit with 4 cores active.</td>
</tr>
<tr>
<td>39:32</td>
<td>MAX_TURBO_GROUP_4: Maximum turbo ratio limit with 5 cores active.</td>
</tr>
<tr>
<td>47:40</td>
<td>MAX_TURBO_GROUP_5: Maximum turbo ratio limit with 6 cores active.</td>
</tr>
<tr>
<td>55:48</td>
<td>MAX_TURBO_GROUP_6: Maximum turbo ratio limit with 7 cores active.</td>
</tr>
<tr>
<td>63:56</td>
<td>MAX_TURBO_GROUP_7: Maximum turbo ratio limit with 8 cores active.</td>
</tr>
<tr>
<td>Register Address: 664H, 1636</td>
<td>MSR_MC6_RESIDENCY_COUNTER</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-46. Additional MSRs Supported by the 12th and 13th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>63:0</td>
<td>Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.</td>
</tr>
<tr>
<td>Register Address: 6E1H, 1761</td>
<td>IA32_PKRS</td>
</tr>
<tr>
<td>Specifies the PK permissions associated with each protection domain for supervisor pages (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 776H, 1910</td>
<td>IA32_HwP_CTL</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 981H, 2433</td>
<td>IA32_TME_CAPABILITY</td>
</tr>
<tr>
<td>Memory Encryption Capability MSR See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1200H—121FH, 4608—4639</td>
<td>IA32_LBR_x_INFO</td>
</tr>
<tr>
<td>Last Branch Record Entry X Info Register (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 14CEH, 5326</td>
<td>IA32_LBR_CTL</td>
</tr>
<tr>
<td>Last Branch Record Enabling and Configuration Register (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 14CFH, 5327</td>
<td>IA32_LBR_DEPTH</td>
</tr>
<tr>
<td>Last Branch Record Maximum Stack Depth Register (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1500H—151FH, 5376—5407</td>
<td>IA32_LBR_x_FROM_IP</td>
</tr>
<tr>
<td>Last Branch Record Entry X Source IP Register (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1600H—161FH, 5632—5663</td>
<td>IA32_LBR_x_TO_IP</td>
</tr>
<tr>
<td>Last Branch Record Entry X Destination IP Register (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17D2H, 6098</td>
<td>IA32_THREAD_FEEDBACK_CHAR</td>
</tr>
<tr>
<td>Thread Feedback Characteristics (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17D4H, 6100</td>
<td>IA32_Hw_FEEDBACK_THREAD_CONFIG</td>
</tr>
<tr>
<td>Hardware Feedback Thread Configuration (R/W) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17DAH, 6106</td>
<td>IA32_HRESET_ENABLE</td>
</tr>
<tr>
<td>History Reset Enable (R/W) See Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>

The MSRs listed in Table 2-47 are unique to the 12th and 13th generation Intel Core processor P-core. These MSRs are not supported on the processor E-core.
### Table 2-47. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor P-core

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefetch Disable Bits (R/W)</td>
<td>MSR_PREFETCH_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>L2_HARDWARE_PREFETCHER_DISABLE</td>
<td>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L2_ADJACENT_CACHE_LINE_PREFETCHER_DISABLE</td>
<td>If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DCU_HARDWARE_PREFETCHER_DISABLE</td>
<td>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DCU_IP_PREFETCHER_DISABLE</td>
<td>If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>AMP_PREFETCHER_DISABLE</td>
<td>If 1, disables the L2 Adaptive Multipath Probability (AMP) prefetcher.</td>
<td></td>
</tr>
<tr>
<td>63:6</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrontEnd Precise Event Condition Select (R/W)</td>
<td>MSR_PEBBS_FRONTEND</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>See Table 2-39.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread Microarchitectural Control (R/W)</td>
<td>MSR_THREAD_UARCH_CTL</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>WB_MEM_STRM_LD_DISABLE</td>
<td>Disable streaming behavior for MOVNTDQA loads to WB memory type. If set, these accesses will be treated like regular cacheable loads (Data will be cached).</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core Microarchitecture Control MSR (R/W)</td>
<td>MSR_CORE_UARCH_CTL</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>See Table 2-44.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_L2_QOS_MASK_[0-7]</td>
<td>IA32_CR_L2_QOS_MASK_[0-7]</td>
<td>Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 1B, &quot;Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features.&quot;</td>
<td>Core</td>
</tr>
</tbody>
</table>
The MSRs listed in Table 2-48 are unique to the 12th and 13th generation Intel Core processor E-core. These MSRs are not supported on the processor P-core.

### Table 2-48. MSRs Supported by 12th and 13th Generation Intel® Core™ Processor E-core

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>19:0</td>
<td>WAYS_MASK</td>
<td>Setting a 1 in this bit X allows threads with CLOS &lt;n&gt; (where N is [0-7]) to allocate to way X in the MLC. Ones are only allowed to be written to ways that physically exist in the MLC (CPUID.4.2:EBX[31:22] will indicate this). Writing a 1 to a value beyond the highest way or a non-contiguous set of 1s will cause a #GP on the WRMSR to this MSR.</td>
<td></td>
</tr>
<tr>
<td>31:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: D10H—D1FH, 3220—3359

| IA32_L2_QOS_MASK_[0-15] | Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 18, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features." |

Register Address: 1309H—130BH, 4873—4875

<table>
<thead>
<tr>
<th>MSR_RELOAD_FIXED_CTRx</th>
<th>Reload value for IA32_FIXED_CTRx (R/w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:0</td>
<td>Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEB_ENABLE, and FIXED_CTRx is overflowed.</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 14C1H—14C6H, 5313—5318

<table>
<thead>
<tr>
<th>MSR_RELOAD_PMCx</th>
<th>Reload value for IA32_PMCx (R/w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:0</td>
<td>Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEB_ENABLE, and PMCx is overflowed.</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Table 2-49 lists the MSRs of uncore PMU for Intel processors with a CPUID Signature DisplayFamily_DisplayModel value of 06_97H, 06_9AH, 06_BAH, 06_B7H, or 06_BFH.
### Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 396H, 918</td>
<td>MSR_UNC_CBO_CONFIG</td>
<td>Uncore C-Box Configuration Information (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>3:0</td>
<td>Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics).</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 2000H, 8192</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL0</td>
<td>Uncore C-Box 0, Counter 0 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2001H, 8193</td>
<td>MSR_UNC_CBO_0_PERFEVTSEL1</td>
<td>Uncore C-Box 0, Counter 1 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2002H, 8194</td>
<td>MSR_UNC_CBO_0_PERFCTR0</td>
<td>Uncore C-Box 0, Performance Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2003H, 8195</td>
<td>MSR_UNC_CBO_0_PERFCTR1</td>
<td>Uncore C-Box 0, Performance Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2008H, 8200</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL0</td>
<td>Uncore C-Box 1, Counter 0 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2009H, 8201</td>
<td>MSR_UNC_CBO_1_PERFEVTSEL1</td>
<td>Uncore C-Box 1, Counter 1 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 200AH, 8202</td>
<td>MSR_UNC_CBO_1_PERFCTR0</td>
<td>Uncore C-Box 1, Performance Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2010H, 8203</td>
<td>MSR_UNC_CBO_1_PERFCTR1</td>
<td>Uncore C-Box 1, Performance Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2011H, 8208</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL0</td>
<td>Uncore C-Box 2, Counter 0 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2012H, 8209</td>
<td>MSR_UNC_CBO_2_PERFEVTSEL1</td>
<td>Uncore C-Box 2, Counter 1 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2018H, 8210</td>
<td>MSR_UNC_CBO_2_PERFCTR0</td>
<td>Uncore C-Box 2, Performance Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2019H, 8211</td>
<td>MSR_UNC_CBO_2_PERFCTR1</td>
<td>Uncore C-Box 2, Performance Counter 1</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2018H, 8216</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL0</td>
<td>Uncore C-Box 3, Counter 0 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2019H, 8217</td>
<td>MSR_UNC_CBO_3_PERFEVTSEL1</td>
<td>Uncore C-Box 3, Counter 1 Event Select MSR</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 201AH, 8218</td>
<td>MSR_UNC_CBO_3_PERFCTR0</td>
<td>Uncore C-Box 3, Performance Counter 0</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 201BH, 8219</td>
<td>MSR_UNC_CBO_3_PERFCTR1</td>
<td>Uncore C-Box 3, Performance Counter 1</td>
<td>Package</td>
</tr>
</tbody>
</table>
Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 2020H, 8224</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 0 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2021H, 8225</td>
<td>MSR_UNC_CBO_4_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Counter 1 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2022H, 8226</td>
<td>MSR_UNC_CBO_4_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2023H, 8227</td>
<td>MSR_UNC_CBO_4_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 4, Performance Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2028H, 8232</td>
<td>MSR_UNC_CBO_5_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Counter 0 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2029H, 8233</td>
<td>MSR_UNC_CBO_5_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Counter 1 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 202AH, 8234</td>
<td>MSR_UNC_CBO_5_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Performance Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 202BH, 8235</td>
<td>MSR_UNC_CBO_5_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 5, Performance Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2030H, 8240</td>
<td>MSR_UNC_CBO_6_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6, Counter 0 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2031H, 8241</td>
<td>MSR_UNC_CBO_6_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6, Counter 1 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2032H, 8242</td>
<td>MSR_UNC_CBO_6_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6, Performance Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2033H, 8243</td>
<td>MSR_UNC_CBO_6_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 6, Performance Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2038H, 8248</td>
<td>MSR_UNC_CBO_7_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7, Counter 0 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2039H, 8249</td>
<td>MSR_UNC_CBO_7_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7, Counter 1 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 203AH, 8250</td>
<td>MSR_UNC_CBO_7_PERFCTR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7, Performance Counter 0</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 203BH, 8251</td>
<td>MSR_UNC_CBO_7_PERFCTR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 7, Performance Counter 1</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2040H, 8256</td>
<td>MSR_UNC_CBO_8_PERFEVTSEL0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8, Counter 0 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2041H, 8257</td>
<td>MSR_UNC_CBO_8_PERFEVTSEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore C-Box 8, Counter 1 Event Select MSR</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 2042H, 8258</td>
<td>MSR_UNC_CBO_8_PERFCTR0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncore C-Box 8, Performance Counter 0</td>
<td>Register Address: 2043H, B259</td>
<td>MSR_UNC_CBO_8_PERFCTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-Box 8, Performance Counter 1</td>
<td>Register Address: 2048H, B264</td>
<td>MSR_UNC_CBO_9_PERFEVTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-Box 9, Counter 0 Event Select MSR</td>
<td>Register Address: 2049H, B265</td>
<td>MSR_UNC_CBO_9_PERFEVTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-Box 9, Counter 1 Event Select MSR</td>
<td>Register Address: 204AH, B266</td>
<td>MSR_UNC_CBO_9_PERFCTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore C-Box 9, Performance Counter 0</td>
<td>Register Address: 204BH, B267</td>
<td>MSR_UNC_CBO_9_PERFCTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 0, Counter 0 Event Select MSR</td>
<td>Register Address: 2FD0H, 12240</td>
<td>MSR_UNC_ARB_0_PERFEVTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 0, Counter 1 Event Select MSR</td>
<td>Register Address: 2FD1H, 12241</td>
<td>MSR_UNC_ARB_0_PERFEVTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 0, Performance Counter 0</td>
<td>Register Address: 2FD2H, 12242</td>
<td>MSR_UNC_ARB_0_PERFCTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 0, Performance Counter 1</td>
<td>Register Address: 2FD3H, 12243</td>
<td>MSR_UNC_ARB_0_PERFCTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 0, Performance Status</td>
<td>Register Address: 2FD4H, 12244</td>
<td>MSR_UNC_ARB_0_PERF_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 0, Performance Control</td>
<td>Register Address: 2FD5H, 12245</td>
<td>MSR_UNC_ARB_0_PERF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 1, Counter 0 Event Select MSR</td>
<td>Register Address: 2FD9H, 12249</td>
<td>MSR_UNC_ARB_1_PERFEVTSEL0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 1, Counter 1 Event Select MSR</td>
<td>Register Address: 2FDD9H, 12250</td>
<td>MSR_UNC_ARB_1_PERFEVTSEL1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 1, Performance Counter 0</td>
<td>Register Address: 2FDBH, 12251</td>
<td>MSR_UNC_ARB_1_PERFCTR0</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 1, Performance Counter 1</td>
<td>Register Address: 2FDCH, 12252</td>
<td>MSR_UNC_ARB_1_PERFCTR1</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 1, Performance Status</td>
<td>Register Address: 2FEFH, 12253</td>
<td>MSR_UNC_ARB_1_PERF_STATUS</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Arb Unit 1, Performance Control</td>
<td>Register Address: 2FDEH, 12254</td>
<td>MSR_UNC_ARB_1_PERF_CTRL</td>
<td>Package</td>
</tr>
<tr>
<td>Uncore Fixed Counter Control (R/W)</td>
<td>Register Address: 2FDH, 12255</td>
<td>MSR_UNC_PERF_FIXED_CTRL</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-49. Uncore PMU MSRs Supported by 12th and 13th Generation Intel® Core™ Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>2FDFH, 12255</td>
<td>MSR_UNC_PERF_FIXED_CTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2FF0H, 12272</td>
<td>MSR_UNC_PERF_GLOBAL_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2FF2H, 12274</td>
<td>MSR_UNC_PERF_GLOBAL_STATUS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.17.6 MSRs Introduced in the Intel® Xeon® Scalable Processor Family

The Intel® Xeon® Scalable Processor Family (CPUID Signature DisplayFamily_DisplayModel value of 06_55H) supports the MSRs listed in Table 2-50.

#### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control Features in Intel 64 Processor (R/W)</strong>&lt;br&gt;See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Lock (R/WL)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>1</td>
<td>Enable VMX Inside SMX Operation (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Enable VMX Outside SMX Operation (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>SENTER Local Functions Enables (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SENTER Global Functions Enable (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>SGX Global Functions Enable (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>LMCE_ENABLED (R/WL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:21</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 4EH, 78</strong></td>
<td>IA32_PPIN_CTL (MSR_PPIN_CTL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Protected Processor Inventory Number Enable Control (R/W)</strong>&lt;br&gt;Package</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LockOut (R/WO)&lt;br&gt;See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Enable_PPIN (R/W)&lt;br&gt;See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 4FH, 79</strong></td>
<td>IA32_PPIN (MSR_PPIN)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Protected Processor Inventory Number (R/O)</strong>&lt;br&gt;Package</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Protected Processor Inventory Number (R/O)&lt;br&gt;See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: CEH, 206</strong></td>
<td>MSR_PLATFORM_INFO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Platform Information</strong>&lt;br&gt;Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td></td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)&lt;br&gt;See Table 2-26.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>22:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>PPIN_CAP (R/O)&lt;br&gt;See Table 2-26.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>27:24</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)&lt;br&gt;See Table 2-26.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)&lt;br&gt;See Table 2-26.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>30</td>
<td>Programmable TJ OFFSET (R/O)&lt;br&gt;See Table 2-26.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>39:31</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------</td>
<td>----------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
<td>See Table 2-26.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: E2H, 226

MSR_PKG_CST_CONFIG_CONTROL

C-State Configuration Control (R/W)
Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

<table>
<thead>
<tr>
<th>2:0</th>
<th>Package C-State Limit (R/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit.</td>
</tr>
<tr>
<td></td>
<td>The following C-state code name encodings are supported:</td>
</tr>
<tr>
<td></td>
<td>000b: C0/C1 (no package C-state support)</td>
</tr>
<tr>
<td></td>
<td>001b: C2</td>
</tr>
<tr>
<td></td>
<td>010b: C6 (non-retention)</td>
</tr>
<tr>
<td></td>
<td>011b: C6 (retention)</td>
</tr>
<tr>
<td></td>
<td>111b: No Package C state limits. All C states supported by the processor are available.</td>
</tr>
</tbody>
</table>

| 9:3  | Reserved.                                                                                     |
| 10   | I/O MWAIT Redirection Enable (R/W)                                                            |
| 14:11| Reserved.                                                                                     |
| 15   | CFG Lock (R/WO)                                                                               |
| 16   | Automatic C-State Conversion Enable (R/W)                                                     |
|     | If 1, the processor will convert HALT or MWAIT(C1) to MWAIT(C6).                             |

| 24:17| Reserved.                                                                                     |
| 25   | C3 State Auto Demotion Enable (R/W)                                                          |
| 26   | C1 State Auto Demotion Enable (R/W)                                                          |
| 27   | Enable C3 Undemotion (R/W)                                                                   |
| 28   | Enable C1 Undemotion (R/W)                                                                   |
| 29   | Package C State Demotion Enable (R/W)                                                         |
| 30   | Package C State Undemotion Enable (R/W)                                                       |
| 63:31| Reserved.                                                                                     |

Register Address: 179H, 377

IA32_MCG_CAP

Global Machine Check Capability (R/O)

| 7:0  | Count.                                                                                       |
| 8    | MCG_CTL_P                                                                                   |
| 9    | MCG_EXT_P                                                                                    |
| 10   | MCP_CMCI_P                                                                                   |
| 11   | MCG_TES_P                                                                                    |

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature
DisplayFamily_DisplayModel Value of 06_55H (Contd.)
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature
TableFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>MCG_EXT_CNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MCG_SER_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>MCG_EM_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>MCG_ELOG_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:27</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 17DH, 381  
**MSR_SMM_MCA_CAP**

<table>
<thead>
<tr>
<th>Enhanced SMM Capabilities (SMM-RO)</th>
<th>Reports SMM capability Enhancement. Accessible only while in SMM.</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>57:0</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
| 58                                 | SMM_Code_Access_Chk (SMM-RO)  
If set to 1 indicates that the SMM code access restriction is supported and a host-space interface is available to SMM handler. |        |
| 59                                 | Long_Flow_Indication (SMM-RO)  
If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface is available to SMM handler. |        |
| 63:60                             | Reserved.                                                             |        |

Register Address: 19CH, 412  
**IA32_THERM_STATUS**

<table>
<thead>
<tr>
<th>Thermal Monitor Status (R/W)</th>
<th>See Table 2-2.</th>
<th>Core</th>
</tr>
</thead>
</table>
| 0                           | Thermal Status (R/O)  
See Table 2-2. |      |
| 1                           | Thermal Status Log (R/WC0)  
See Table 2-2. |      |
| 2                           | PROTCHOT # or FORCEPR# Status (R/O)  
See Table 2-2. |      |
| 3                           | PROTCHOT # or FORCEPR# Log (R/WC0)  
See Table 2-2. |      |
| 4                           | Critical Temperature Status (R/O)  
See Table 2-2. |      |
| 5                           | Critical Temperature Status Log (R/WC0)  
See Table 2-2. |      |
| 6                           | Thermal Threshold #1 Status (R/O)  
See Table 2-2. |      |
| 7                           | Thermal Threshold #1 Log (R/WC0)  
See Table 2-2. |      |
| 8                           | Thermal Threshold #2 Status (R/O)  
See Table 2-2. |      |
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>Thermal Threshold #2 Log (R/WCO)</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td>Power Limitation Status (R/O)</td>
</tr>
<tr>
<td><strong>11</strong></td>
<td>Power Limitation Log (R/WCO)</td>
</tr>
<tr>
<td><strong>12</strong></td>
<td>Current Limit Status (R/O)</td>
</tr>
<tr>
<td><strong>13</strong></td>
<td>Current Limit Log (R/WCO)</td>
</tr>
<tr>
<td><strong>14</strong></td>
<td>Cross Domain Limit Status (R/O)</td>
</tr>
<tr>
<td><strong>15</strong></td>
<td>Cross Domain Limit Log (R/WCO)</td>
</tr>
<tr>
<td><strong>22:16</strong></td>
<td>Digital Readout (R/O)</td>
</tr>
<tr>
<td><strong>26:23</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>30:27</strong></td>
<td>Resolution in Degrees Celsius (R/O)</td>
</tr>
<tr>
<td><strong>31</strong></td>
<td>Reading Valid (R/O)</td>
</tr>
<tr>
<td><strong>63:32</strong></td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 1A2H, 418 MSR_TEMPERATURE_TARGET

**Temperature Target** Package

| 15:0 | Reserved. | |
| 23:16 | Temperature Target (R/O) | See Table 2-26. |
| 27:24 | TCC Activation Offset (R/W) | See Table 2-26. |
| 63:28 | Reserved. | |

Register Address: 1ADH, 429 MSR_TURBO_RATIO_LIMIT

This register defines the ratio limits. RATIO[0:7] must be populated in ascending order. RATIO[i+1] must be less than or equal to RATIO[i]. Entries with RATIO[i] will be ignored. If any of the rules above are broken, the configuration is silently rejected. If the programmed ratio is:
- Above the fused ratio for that core count, it will be clipped to the fuse limits (assuming !OC).
- Below the min supported ratio, it will be clipped.

| 7:0 | RATIO_0 | Defines ratio limits. |
| 15:8 | RATIO_1 | Defines ratio limits. |
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature

#### DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>23:16 RATIO_2</td>
<td>Defines ratio limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24 RATIO_3</td>
<td>Defines ratio limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:32 RATIO_4</td>
<td>Defines ratio limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:40 RATIO_5</td>
<td>Defines ratio limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55:48 RATIO_6</td>
<td>Defines ratio limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:56 RATIO_7</td>
<td>Defines ratio limits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1AEH, 430</td>
<td>MSR_TURBO_RATIO_LIMIT_CORES</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>This register defines the active core ranges for each frequency point. NUMCORE[0:7] must be populated in ascending order. NUMCORE[i+1] must be greater than NUMCORE[i]. Entries with NUMCORE[i] == 0 will be ignored. The last valid entry must have NUMCORE &gt;= the number of cores in the SKU. If any of the rules above are broken, the configuration is silently rejected.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 7:0 NUMCORE_0                 | Defines the active core ranges for each frequency point. | Package |
| 15:8 NUMCORE_1                | Defines the active core ranges for each frequency point. |       |
| 23:16 NUMCORE_2               | Defines the active core ranges for each frequency point. |       |
| 31:24 NUMCORE_3               | Defines the active core ranges for each frequency point. |       |
| 39:32 NUMCORE_4               | Defines the active core ranges for each frequency point. |       |
| 47:40 NUMCORE_5               | Defines the active core ranges for each frequency point. |       |
| 55:48 NUMCORE_6               | Defines the active core ranges for each frequency point. |       |
| 63:56 NUMCORE_7               | Defines the active core ranges for each frequency point. |       |

<p>| Register Address: 280H, 640    | IA32_MCO_CTL2 | Core |
| Register Address: 281H, 641    | IA32_MC1_CTL2 | Core |
| Register Address: 282H, 642    | IA32_MC2_CTL2 | Core |</p>
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>283H, 643</td>
<td>IA32_MC3_CTL2</td>
<td>See Table 2-2.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>284H, 644</td>
<td>IA32_MC4_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>285H, 645</td>
<td>IA32_MC5_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>286H, 646</td>
<td>IA32_MC6_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>287H, 647</td>
<td>IA32_MC7_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>288H, 648</td>
<td>IA32_MC8_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>289H, 649</td>
<td>IA32_MC9_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>28AH, 650</td>
<td>IA32_MC10_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>28BH, 651</td>
<td>IA32_MC11_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>28CH, 652</td>
<td>IA32_MC12_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>28DH, 653</td>
<td>IA32_MC13_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>28EH, 654</td>
<td>IA32_MC14_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>28FH, 655</td>
<td>IA32_MC15_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>290H, 656</td>
<td>IA32_MC16_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>291H, 657</td>
<td>IA32_MC17_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>292H, 658</td>
<td>IA32_MC18_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>293H, 659</td>
<td>IA32_MC19_CTL2</td>
<td>See Table 2-2.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td>See Section 16.3.2.1, “IA32_MC1_CTL MSRs,” through Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>Core</td>
<td></td>
</tr>
</tbody>
</table>

Bank MC0 reports MC errors from the IFU module.
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IA32_MC0_STATUS</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.</td>
</tr>
<tr>
<td><strong>IA32_MC0_ADDR</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.</td>
</tr>
<tr>
<td><strong>IA32_MC0_MISC</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC0 reports MC errors from the IFU module.</td>
</tr>
<tr>
<td><strong>IA32_MC1_CTL</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.</td>
</tr>
<tr>
<td><strong>IA32_MC1_STATUS</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.</td>
</tr>
<tr>
<td><strong>IA32_MC1_ADDR</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.</td>
</tr>
<tr>
<td><strong>IA32_MC1_MISC</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC1 reports MC errors from the DCU module.</td>
</tr>
<tr>
<td><strong>IA32_MC2_CTL</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.</td>
</tr>
<tr>
<td><strong>IA32_MC2_STATUS</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.</td>
</tr>
<tr>
<td><strong>IA32_MC2_ADDR</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.</td>
</tr>
<tr>
<td><strong>IA32_MC2_MISC</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC2 reports MC errors from the DTLB module.</td>
</tr>
<tr>
<td><strong>IA32_MC3_CTL</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC3 reports MC errors from the MLC module.</td>
</tr>
<tr>
<td><strong>IA32_MC3_STATUS</strong></td>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.” Bank MC3 reports MC errors from the MLC module.</td>
</tr>
</tbody>
</table>
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature

#### DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC3 reports MC errors from the MLC module.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MCI_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC3 reports MC errors from the MLC module.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 40FH, 1039</td>
<td>IA32_MCI_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC3 reports MC errors from the MLC module.</td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
<td>IA32_MCI_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC4 reports MC errors from the PCU module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 411H, 1041</td>
<td>IA32_MCI_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC4 reports MC errors from the PCU module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 412H, 1042</td>
<td>IA32_MCI_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC4 reports MC errors from the PCU module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 413H, 1043</td>
<td>IA32_MCI_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC4 reports MC errors from the PCU module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
<td>IA32_MCI_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC5 reports MC errors from a link interconnect module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
<td>IA32_MCI_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC5 reports MC errors from a link interconnect module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 416H, 1046</td>
<td>IA32_MCI_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC5 reports MC errors from a link interconnect module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 417H, 1047</td>
<td>IA32_MCI_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC5 reports MC errors from a link interconnect module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 418H, 1048</td>
<td>IA32_MCI_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 419H, 1049</td>
<td>IA32_MCI_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td></td>
<td></td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 41AH, 1050</td>
<td>IA32_MC6_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41BH, 1051</td>
<td>IA32_MC6_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC6 reports MC errors from the integrated I/O module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41CH, 1052</td>
<td>IA32_MC7_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the M2M 0.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 41DH, 1053</td>
<td>IA32_MC7_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC7 reports MC errors from the M2M 0.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 420H, 1056</td>
<td>IA32_MC8_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the M2M 1.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 421H, 1057</td>
<td>IA32_MC8_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the M2M 1.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 422H, 1058</td>
<td>IA32_MC8_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the M2M 1.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 423H, 1059</td>
<td>IA32_MC8_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Bank MC8 reports MC errors from the M2M 1.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 424H, 1060</td>
<td>IA32_MC9_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 425H, 1061</td>
<td>IA32_MC9_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCi_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MCi_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 426H, 1062</td>
<td>IA32_MC9_ADDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC9_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 427H, 1063</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC10_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 428H, 1064</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC10_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 429H, 1065</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC10_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42AH, 1066</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC11_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42CH, 1068</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC11_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42DH, 1069</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC11_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42EH, 1070</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC11_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 42FH, 1071</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC12_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 430H, 1072</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC12_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 431H, 1073</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC_MISC MSRs.&quot;</td>
<td>Banks MC9 - MC11 report MC errors from the CHA.</td>
<td>IA32_MC12_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 432H, 1074</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Register Name (Former Register Name)</td>
<td>Bit Description</td>
<td>Scope</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------</td>
<td>----------------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Register Address: 433H, 1075</td>
<td>IA32_MC12_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC12_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC12_MISC MSRs.&quot;</td>
<td>Banks MC12 report MC errors from each channel of a link interconnect module.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 434H, 1076</td>
<td>IA32_MC13_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC13_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC13_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 435H, 1077</td>
<td>IA32_MC13_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC13_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC13_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 436H, 1078</td>
<td>IA32_MC13_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC13_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC13_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 437H, 1080</td>
<td>IA32_MC13_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC13_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC13_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 438H, 1081</td>
<td>IA32_MC14_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC14_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC14_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 439H, 1082</td>
<td>IA32_MC14_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC14_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC14_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43AH, 1083</td>
<td>IA32_MC14_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC14_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC14_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43BH, 1084</td>
<td>IA32_MC15_CTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC15_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC15_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43CH, 1085</td>
<td>IA32_MC15_ADDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC15_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC15_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43DH, 1086</td>
<td>IA32_MC15_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MC15_CTL MSRs,&quot; through Section 16.3.2.4, &quot;IA32_MC15_MISC MSRs.&quot;</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 43FH, 1087</td>
<td>IA32_MC15_MISC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>Banks MC13 through MC 18 report MC errors from the integrated memory controllers.</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
</tbody>
</table>
## Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 44CH, 1100</td>
<td>IA32_MC19_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from a link interconnect module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44DH, 1101</td>
<td>IA32_MC19_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from a link interconnect module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44EH, 1102</td>
<td>IA32_MC19_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from a link interconnect module.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 44FH, 1103</td>
<td>IA32_MC19_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs,” through Section 16.3.2.4, “IA32_MCi_MISC MSRs.”</td>
<td>Bank MC19 reports MC errors from a link interconnect module.</td>
<td>Package</td>
<td></td>
</tr>
</tbody>
</table>

### Unit Multipliers Used in RAPL Interfaces (R/O)

#### 3:0
- Power Units
  - See Section 15.10.1, “RAPL Interfaces.”
  - Package

#### 7:4
- Reserved.
  - Package

#### 12:8
- Energy Status Units
  - Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).
  - Package

#### 15:13
- Reserved.
  - Package

#### 19:16
- Time Units
  - See Section 15.10.1, “RAPL Interfaces.”
  - Package

#### 63:20
- Reserved.
  - Package

| Register Address: 618H, 1560 | MSR_DRAM_POWER_LIMIT                  |                 |       |
| DRAM RAPL Power Limit Control (R/W) | See Section 15.10.5, “DRAM RAPL Domain.” | Package |
| Register Address: 619H, 1561 | MSR_DRAM_ENERGY_STATUS                |                 |       |
| DRAM Energy Status (R/O) | Energy consumed by DRAM devices. | Package |
| 31:0 | Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR). | Package |
| 63:32 | Reserved. | Package |

| Register Address: 61BH, 1563 | MSR_DRAM_PERF_STATUS                  |                 |       |
| DRAM Performance Throttling Status (R/O) | See Section 15.10.5, “DRAM RAPL Domain.” | Package |
| Register Address: 61CH, 1564 | MSR_DRAM_POWER_INFO                   |                 |       |
### MODEL-SPECIFIC REGISTERS (MSRS)

#### DRAM RAPL Parameters (R/W)
See Section 15.10.5, "DRAM RAPL Domain."

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Register Name (Former Register Name)</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>620H, 1568</td>
<td>MSR_UNCORE_RATIO_LIMIT</td>
<td>Package</td>
</tr>
</tbody>
</table>

Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:15</td>
<td>Reserved.</td>
</tr>
<tr>
<td>14:8</td>
<td>MIN_RATIO</td>
</tr>
<tr>
<td></td>
<td>Writing to this field controls the minimum possible ratio of the LLC/Ring.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
</tr>
<tr>
<td>6:0</td>
<td>MAX_RATIO</td>
</tr>
<tr>
<td></td>
<td>This field is used to limit the max ratio of the LLC/Ring.</td>
</tr>
</tbody>
</table>

#### Package

<table>
<thead>
<tr>
<th>Register Address</th>
<th>MSR_PP0_ENERGY_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C8DH, 3213</td>
<td>Reserved (R/O)</td>
</tr>
</tbody>
</table>

Reads return 0.

<table>
<thead>
<tr>
<th>Register Address</th>
<th>IA32_QM_EVTSEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>C8FH, 3215</td>
<td>Monitoring Event Select Register (R/w)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>EventID (R/w)</td>
</tr>
<tr>
<td></td>
<td>Event encoding:</td>
</tr>
<tr>
<td></td>
<td>0x00: No monitoring.</td>
</tr>
<tr>
<td></td>
<td>0x01: L3 occupancy monitoring.</td>
</tr>
<tr>
<td></td>
<td>0x02: Total memory bandwidth monitoring.</td>
</tr>
<tr>
<td></td>
<td>0x03: Local memory bandwidth monitoring.</td>
</tr>
<tr>
<td></td>
<td>All other encoding reserved.</td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
</tr>
<tr>
<td>41:32</td>
<td>RMID (R/w)</td>
</tr>
<tr>
<td>63:42</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

#### Thread

<table>
<thead>
<tr>
<th>Register Address</th>
<th>IA32_PQR_ASSOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>C90H, 3216</td>
<td>Resource Association Register (R/w)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:0</td>
<td>RMID</td>
</tr>
<tr>
<td>31:10</td>
<td>Reserved.</td>
</tr>
<tr>
<td>51:32</td>
<td>COS (R/w)</td>
</tr>
<tr>
<td>63:52</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

#### Package

<table>
<thead>
<tr>
<th>Register Address</th>
<th>IA32_L3_QOS_MASK_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C90H, 3216</td>
<td>L3 Class Of Service Mask - COS 0 (R/w)</td>
</tr>
<tr>
<td></td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 0 enforcement.</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

---

**Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_55H (Contd.)**

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>620H, 1568</td>
<td>MSR_UNCORE_RATIO_LIMIT</td>
<td>Package</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:15</td>
<td>Reserved.</td>
</tr>
<tr>
<td>14:8</td>
<td>MIN_RATIO</td>
</tr>
<tr>
<td></td>
<td>Writing to this field controls the minimum possible ratio of the LLC/Ring.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
</tr>
<tr>
<td>6:0</td>
<td>MAX_RATIO</td>
</tr>
<tr>
<td></td>
<td>This field is used to limit the max ratio of the LLC/Ring.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address</th>
<th>MSR_PP0_ENERGY_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C8DH, 3213</td>
<td>Reserved (R/O)</td>
</tr>
</tbody>
</table>

Reads return 0.

<table>
<thead>
<tr>
<th>Register Address</th>
<th>IA32_QM_EVTSEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>C8FH, 3215</td>
<td>Monitoring Event Select Register (R/w)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>EventID (R/w)</td>
</tr>
<tr>
<td></td>
<td>Event encoding:</td>
</tr>
<tr>
<td></td>
<td>0x00: No monitoring.</td>
</tr>
<tr>
<td></td>
<td>0x01: L3 occupancy monitoring.</td>
</tr>
<tr>
<td></td>
<td>0x02: Total memory bandwidth monitoring.</td>
</tr>
<tr>
<td></td>
<td>0x03: Local memory bandwidth monitoring.</td>
</tr>
<tr>
<td></td>
<td>All other encoding reserved.</td>
</tr>
<tr>
<td>31:8</td>
<td>Reserved.</td>
</tr>
<tr>
<td>41:32</td>
<td>RMID (R/w)</td>
</tr>
<tr>
<td>63:42</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address</th>
<th>IA32_PQR_ASSOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>C90H, 3216</td>
<td>Resource Association Register (R/w)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:0</td>
<td>RMID</td>
</tr>
<tr>
<td>31:10</td>
<td>Reserved.</td>
</tr>
<tr>
<td>51:32</td>
<td>COS (R/w)</td>
</tr>
<tr>
<td>63:52</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address</th>
<th>IA32_L3_QOS_MASK_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C90H, 3216</td>
<td>L3 Class Of Service Mask - COS 0 (R/w)</td>
</tr>
<tr>
<td></td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=0.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 0 enforcement.</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
## Model-Specific Registers (MSRs)

Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: C91H, 3217</td>
<td>IA32_L3_QOS_MASK_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 1 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=1.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 1 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C92H, 3218</td>
<td>IA32_L3_QOS_MASK_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 2 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=2.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 2 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C93H, 3219</td>
<td>IA32_L3_QOS_MASK_3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 3 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C94H, 3220</td>
<td>IA32_L3_QOS_MASK_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 4 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C95H, 3221</td>
<td>IA32_L3_QOS_MASK_5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 5 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=5.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 5 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C96H, 3222</td>
<td>IA32_L3_QOS_MASK_6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 6 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C97H, 3223</td>
<td>IA32_L3_QOS_MASK_7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Class Of Service Mask - COS 7 (R/W)</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] &gt;=7.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 7 enforcement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C98H, 3224</td>
<td>IA32_L3_QOS_MASK_8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-50. MSRs Supported by the Intel® Xeon® Scalable Processor Family with a CPUID Signature
*DisplayFamily_DisplayModel Value of 06_55H (Contd.)*

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name (Former Register Name)</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 8 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 8 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C99H, 3225</td>
<td>IA32_L3_QOS_MASK_9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 9 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 9 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C9AH, 3226</td>
<td>IA32_L3_QOS_MASK_10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 10 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 10 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C9BH, 3227</td>
<td>IA32_L3_QOS_MASK_11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 11 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 11 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C9CH, 3228</td>
<td>IA32_L3_QOS_MASK_12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 12 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 12 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C9DH, 3229</td>
<td>IA32_L3_QOS_MASK_13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 13 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 13 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C9EH, 3230</td>
<td>IA32_L3_QOS_MASK_14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 14 enforcement.</td>
<td>0:19</td>
<td>CBM: Bit vector of available L3 ways for COS 14 enforcement.</td>
<td>Package</td>
</tr>
<tr>
<td>63:20</td>
<td>Reserved.</td>
<td>63:20</td>
<td>Reserved.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C9FH, 3231</td>
<td>IA32_L3_QOS_MASK_15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### MSRs Specific to the 3rd Generation Intel® Xeon® Scalable Processor Family Based on Ice Lake Microarchitecture

The 3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_6AH or 06_6CH) support the MSRs listed in Table 2-51.

#### Table 2-51. MSRs Supported by the 3rd Generation Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_6AH or 06_6CH

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 612H, 1554</td>
<td>MSR_PACKAGE_ENERGY_TIME_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package energy consumed by the entire CPU (R/W)</td>
<td>Package energy consumed by the entire CPU (R/W)</td>
<td>Package energy consumed by the entire CPU (R/W)</td>
<td>Package energy consumed by the entire CPU (R/W)</td>
</tr>
<tr>
<td>31:0</td>
<td>Total amount of energy consumed since last reset.</td>
<td>Total amount of energy consumed since last reset.</td>
<td>Total amount of energy consumed since last reset.</td>
</tr>
<tr>
<td>63:32</td>
<td>Total time elapsed when the energy was last updated. This is a monotonic increment counter with auto wrap back to zero after overflow. Unit is 10ns.</td>
<td>Total time elapsed when the energy was last updated. This is a monotonic increment counter with auto wrap back to zero after overflow. Unit is 10ns.</td>
<td>Total time elapsed when the energy was last updated. This is a monotonic increment counter with auto wrap back to zero after overflow. Unit is 10ns.</td>
</tr>
<tr>
<td>Register Address: 618H, 1560</td>
<td>MSR_DRAM_POWER_LIMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.</td>
<td>Allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.</td>
<td>Allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.</td>
<td>Allows software to set power limits for the DRAM domain and measurement attributes associated with each limit.</td>
</tr>
<tr>
<td>16</td>
<td>Reserved.</td>
<td>Reserved.</td>
<td>Reserved.</td>
</tr>
<tr>
<td>23:17</td>
<td>CTRL_TIME_WIN: Power Limit[0] time window Y value, for DDR domain. Actual time_window for RAPL is: (1/1024 seconds) * (1+(x/4)) * (2^y)</td>
<td>CTRL_TIME_WIN: Power Limit[0] time window Y value, for DDR domain. Actual time_window for RAPL is: (1/1024 seconds) * (1+(x/4)) * (2^y)</td>
<td>CTRL_TIME_WIN: Power Limit[0] time window Y value, for DDR domain. Actual time_window for RAPL is: (1/1024 seconds) * (1+(x/4)) * (2^y)</td>
</tr>
<tr>
<td>62:24</td>
<td>Reserved.</td>
<td>Reserved.</td>
<td>Reserved.</td>
</tr>
<tr>
<td>63</td>
<td>PP_PWR_LIM_LOCK: When set, this entire register becomes read-only. This bit will typically be set by BIOS during boot.</td>
<td>PP_PWR_LIM_LOCK: When set, this entire register becomes read-only. This bit will typically be set by BIOS during boot.</td>
<td>PP_PWR_LIM_LOCK: When set, this entire register becomes read-only. This bit will typically be set by BIOS during boot.</td>
</tr>
<tr>
<td>Register Address: 619H, 1561</td>
<td>MSR_DRAM_ENERGY_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).</td>
<td>Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).</td>
<td>Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td>Reserved.</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 61BH, 1563</td>
<td>MSR_DRAM_PERF_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM Performance Throttling Status (R/O)</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>DRAM Performance Throttling Status (R/O)</td>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
</tr>
<tr>
<td>Register Address: 61CH, 1564</td>
<td>MSR_DRAM_POWER_INFO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM Power Parameters (R/W)</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-51. MSRs Supported by the 3rd Generation Intel® Xeon® Scalable Processor Family with a CPUID Signature DisplayFamily_DisplayModel Value of 06_6AH or 06_6CH (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:0</td>
<td>Spec DRAM Power (DRAM_TDP):</td>
<td>The Spec power allowed for DRAM. The TDP setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:16</td>
<td>Minimal DRAM Power (DRAM_MIN_PWR):</td>
<td>The minimal power setting allowed for DRAM. Lower values will be clamped to this value. The minimum setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46:32</td>
<td>Maximal Package Power (DRAM_MAX_PWR):</td>
<td>The maximal power setting allowed for DRAM. Higher values will be clamped to this value. The maximum setting is typical (not guaranteed). The units for this value are defined in MSR_DRAM_POWER_INFO_UNIT[PWR_UNIT].</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54:48</td>
<td>Maximal Time Window (DRAM_MAX_WIN):</td>
<td>The maximal time window allowed for the DRAM. Higher values will be clamped to this value. (x = \text{PKG_MAX_WIN}[54:53]) (y = \text{PKG_MAX_WIN}[52:48]) The timing interval window is a floating-point number given by (1.x \times \text{power}(2,y)). The unit of measurement is defined in MSR_DRAM_POWER_INFO_UNIT[TIME_UNIT].</td>
<td></td>
</tr>
<tr>
<td>62:55</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>LOCK:</td>
<td>Lock bit to lock the register.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 981H, 2433</td>
<td>IA32_TME_CAPABILITY</td>
<td></td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 982H, 2434</td>
<td>IA32_TME_ACTIVATE</td>
<td></td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 983H, 2435</td>
<td>IA32_TME_EXCLUDE_MASK</td>
<td></td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 984H, 2436</td>
<td>IA32_TME_EXCLUDE_BASE</td>
<td></td>
<td>See Table 2-2.</td>
</tr>
</tbody>
</table>
2.17.8 MSRs Specific to the 4th and 5th Generation Intel® Xeon® Scalable Processor Families

The 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_8FH) and the 5th generation Intel® Xeon® Scalable Processor Family based on Emerald Rapids microarchitecture (CPUID Signature DisplayFamily_DisplayModel value of 06_CFH) both support the MSRs listed in Section 2.17, "MSRs In the 6th—13th Generation Intel® Core™ Processors, 1st—5th Generation Intel® Xeon® Scalable Processor Families, Intel® Core™ Ultra 7 Processors, 8th Generation Intel® Core™ i3 Processors, and Intel® Xeon® E Processors," including Table 2-52. For an MSR listed in Table 2-52 that also appears in the model-specific tables of prior generations, Table 2-52 supersedes prior generation tables.

Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 33H, 51</td>
<td>MSR_MEMORY_CTRL</td>
<td>Memory Control Register (R/W)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>27:0 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28 UC_LOCK_DISABLE</td>
<td></td>
<td>If set to 1, a UC lock will cause a #GP(0) exception.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>29 SPLIT_LOCK_DISABLE</td>
<td></td>
<td>If set to 1, a split lock will cause an #AC(0) exception.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>31:30 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: A7H, 167</td>
<td>MSR_BIOS_DEBUG</td>
<td>BIOS DEBUG (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>BIOS DEBUG (R/O)</td>
<td></td>
<td>See Table 2-45.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: BCH, 188</td>
<td>IA32_MISC_PACKAGE_CTLS</td>
<td>Power Filtering Control (R/W)</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>IA32_ARCH_CAPABILITIES[bit 10] enumerates support for this MSR.</td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: CFH, 207</td>
<td>IA32_CORE_CAPABILITIES</td>
<td>IA32 Core Capabilities Register (R/W)</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>IA32 Core Capabilities Register (R/W)</td>
<td></td>
<td>If CPUID.(EAX=07H, ECX=0):EDX[30] = 1.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>This MSR provides an architectural enumeration function for model-specific behavior.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 Reserved: returns zero.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Reserved: returns zero.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 INTEGRITY_CAPABILITIES</td>
<td></td>
<td>When set to 1, the processor supports MSR_INTEGRITY_CAPABILITIES.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>3 RSM_IN_CPL0_ONLY</td>
<td></td>
<td>Indicates that RSM will only be allowed in CPL0 and will #GP for all non-CPL0 privilege levels.</td>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>4 UC_LOCK_DISABLE_SUPPORTED</td>
<td></td>
<td>When read as 1, software can set bit 28 of MSR_MEMORY_CTRL (MSR address 33H).</td>
<td>Core</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>SPLIT_LOCK_DISABLE_SUPPORTED</td>
</tr>
<tr>
<td>When read as 1, software can set bit 29 of MSR_MEMORY_CTRL.</td>
<td></td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>Reserved: returns zero.</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>UC_STORE_THROTTLING_SUPPORTED</td>
</tr>
<tr>
<td>Indicates that the snoop filter quality of service MSRs are supported on this core. This is based on the existence of a non-inclusive cache and the L2/MLC QoS feature supported.</td>
<td></td>
</tr>
<tr>
<td><strong>63:8</strong></td>
<td>Reserved: returns zero.</td>
</tr>
<tr>
<td><strong>Register Address: E1H, 225</strong></td>
<td>IA32_UMWAIT_CONTROL</td>
</tr>
<tr>
<td><strong>UMWAIT Control (R/w)</strong></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: EDH, 237</strong></td>
<td>MSR_RAR_CONTROL</td>
</tr>
<tr>
<td><strong>RAR Control (R/w)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>63:32</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>31</strong></td>
<td>ENABLE</td>
</tr>
<tr>
<td>RAR events are recognized. When RAR is not enabled, RARs are dropped.</td>
<td></td>
</tr>
<tr>
<td><strong>30</strong></td>
<td>IGNORE_IF</td>
</tr>
<tr>
<td>Allow RAR servicing at the RLP regardless of the value of RFLAGS.IF.</td>
<td></td>
</tr>
<tr>
<td><strong>29:0</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: EEH, 238</strong></td>
<td>MSR_RAR_ACTION_VECTOR_BASE</td>
</tr>
<tr>
<td>Pointer to RAR Action Vector (R/w)</td>
<td></td>
</tr>
<tr>
<td><strong>63:MAXPHYADDR</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>MAXPHYADDR-1:6</strong></td>
<td>VECTOR_PHYSICAL_ADDRESS</td>
</tr>
<tr>
<td>Pointer to the physical address of the 64B aligned RAR action vector.</td>
<td></td>
</tr>
<tr>
<td><strong>5:0</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: EFH, 239</strong></td>
<td>MSR_RAR_PAYLOAD_TABLE_BASE</td>
</tr>
<tr>
<td>Pointer to Base of RAR Payload Table (R/w)</td>
<td></td>
</tr>
<tr>
<td><strong>63:MAXPHYADDR</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>MAXPHYADDR-1:12</strong></td>
<td>TABLE_PHYSICAL_ADDRESS</td>
</tr>
<tr>
<td>Pointer to the base physical address of the 4K aligned RAR payload table.</td>
<td></td>
</tr>
<tr>
<td><strong>11:0</strong></td>
<td>Reserved.</td>
</tr>
<tr>
<td><strong>Register Address: F0H, 240</strong></td>
<td>MSR_RAR_INFO</td>
</tr>
<tr>
<td>Read Only RAR Information (RO)</td>
<td></td>
</tr>
<tr>
<td><strong>63:38</strong></td>
<td>Always zero.</td>
</tr>
<tr>
<td><strong>37:32</strong></td>
<td>Table Max Index</td>
</tr>
<tr>
<td>Maximum supported payload table index.</td>
<td></td>
</tr>
<tr>
<td><strong>31:0</strong></td>
<td>Supported payload type bitmap. A value of 1 in bit position [i] indicates that payload type [i] is supported.</td>
</tr>
<tr>
<td><strong>Register Address: 105H, 261</strong></td>
<td>MSR_CORE_BIST</td>
</tr>
</tbody>
</table>
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core BIST (R/W)</td>
<td>Controls Array BIST activation and status checking as part of FUSA.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>31:0</td>
<td>BIST_ARRAY</td>
<td>Bitmap indicating which arrays to run BIST on (WRITE). Bitmap indicating which arrays were not processed, i.e., completion mask (READ).</td>
<td></td>
</tr>
<tr>
<td>39:32</td>
<td>BANK</td>
<td>Array bank of the [least significant set bit] array indicated in EAX to start BIST (WRITE). Array bank interrupted or failed (READ).</td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>DWORD</td>
<td>Array dword of the [least significant set bit] array indicated in EAX to start BIST (WRITE). Array dword interrupted or failed (READ).</td>
<td></td>
</tr>
<tr>
<td>62:48</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>CTRL_RESULT</td>
<td>Indicates whether WRMSR should signal Machine-Check upon BIST-error (WRITE). BIST result PASS(0)/FAIL(1) of the (least significant set bit) array indicated in EAX (READ).</td>
<td></td>
</tr>
<tr>
<td>Register Address: 10AH, 266</td>
<td>IA32_ARCH_CAPABILITIES</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enumeration of Architectural Features (R/O)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A4H, 420</td>
<td>MSR_PREFETCH_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prefetch Disable Bits (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>L2_HARDWARE_PREFETCHER_DISABLE</td>
<td>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L2_ADJACENT_CACHE_LINE_PREFETCHER_DISABLE</td>
<td>If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DCU_HARDWARE_PREFETCHER_DISABLE</td>
<td>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DCU_IP_PREFETCHER_DISABLE</td>
<td>If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>AMP_PREFETCH_DISABLE</td>
<td>If 1, disables the L2 Adaptive Multipath Probability (AMP) prefetcher.</td>
<td></td>
</tr>
<tr>
<td>63:6</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

#### Register Address: 1ADH, 429
- **Register Name:** MSR_PRIMARY_TURBO_RATIO_LIMIT
- **Bit Description:** Primary Maximum Turbo Ratio Limit (R/W)
- **Scope:** Package
- **See Table 2-46.**

#### Register Address: 1AEH, 430
- **Register Name:** MSR_TURBO_RATIO_LIMIT_CORES
- **See Table 2-50.**
- **Scope:** Package

#### Register Address: 1C4H, 452
- **Register Name:** IA32_XFD
- **Bit Description:** Extended Feature Detect (R/W)
- **Scope:** Die
- **See Table 2-2.**

#### Register Address: 1C5H, 453
- **Register Name:** IA32_XFD_ERR
- **Bit Description:** XFD Error Code (R/W)
- **Scope:** Die
- **See Table 2-2.**

#### Register Address: 2C2H, 706
- **Register Name:** MSR_COPY_SCAN_HASHES
- **Bit Description:** COPY_SCAN_HASHES (w)
- **Scope:** Die
- **63:0** SCAN_HASH_ADDR
  - Contains the linear address of the SCAN Test HASH Binary loaded into memory.

#### Register Address: 2C3H, 707
- **Register Name:** MSR_SCAN_HASHES_STATUS
- **Bit Description:** SCAN_HASHES_STATUS (R/O)
- **Scope:** Die
- **15:0** CHUNK_SIZE
  - Chunk size of the test in KB.
- **23:16** NUM_CHUNKS
  - Total number of chunks.
- **31:24** Reserved: all zeros.
- **39:32** ERROR_CODE
  - The error-code refers to the LP that runs WRMSR(2C2H).
  - 0x0: No error reported.
  - 0x1: Attempt to copy scan-hashes when copy already in progress.
  - 0x2: Secure Memory not set up correctly.
  - 0x3: Scan-image header Image_info.ProgramID doesn't match RDMSR(2D9H)[31:24], or scan-image header Processor-Signature doesn't match F/M/S, or scan-image header Processor-Flags doesn't match PlatformID.
  - 0x4: Reserved
  - 0x5: Integrity check failed.
  - 0x6: Re-install of scan test image attempted when current scan test image is in use by other LPs.
- **50:40** Reserved: set to all zeros.
- **62:51** MAX_CORE_LIMIT
  - Maximum Number of cores that can run Intel® In-field Scan simultaneously minus 1.
  - 0 means 1 core at a time.
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td><strong>MSR_AUTHENTICATE_AND_COPY CHUNK</strong></td>
<td>Valid</td>
<td>Valid bit is set when COPY_SCAN_HASHES has completed successfully.</td>
<td>Die</td>
</tr>
<tr>
<td><strong>MSR_AUTHENTICATE_AND_COPY CHUNK (W)</strong></td>
<td></td>
<td>7:0</td>
<td><strong>CHUNK_INDEX</strong>&lt;br&gt;Chunk Index, should be less than the total number of chunks defined by NUM CHUNKS (MSR_SCAN_HASHES_STATUS[23:16]).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:8</td>
<td><strong>CHUNK_ADDR</strong>&lt;br&gt;Bits 63:8 of 256B aligned Linear address of scan chunk in memory.</td>
<td></td>
</tr>
<tr>
<td><strong>MSR_CHUNKS_AUTHENTICATION_STATUS (R/O)</strong></td>
<td></td>
<td>7:0</td>
<td><strong>VALID_CHUNKS</strong>&lt;br&gt;Total number of Valid (authenticated) chunks.</td>
<td>Die</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15:8</td>
<td><strong>TOTAL CHUNKS</strong>&lt;br&gt;Total number of chunks.</td>
<td>Die</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31:16</td>
<td>Reserved: all zeros.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>39:32</td>
<td><strong>ERROR_CODE</strong>&lt;br&gt;The error code refers to the LP that runs WRMSR(2C4H).&lt;br&gt;0x0: No error reported.&lt;br&gt;0x1: Attempt to authenticate a CHUNK which is already marked as authentic or is currently being installed by another core.&lt;br&gt;0x2: CHUNK authentication error. HASH of chunk did not match expected value.</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>63:40</td>
<td>Reserved: set to all zeros.</td>
<td></td>
</tr>
<tr>
<td><strong>MSR_ACTIVATE_SCAN (W)</strong></td>
<td></td>
<td>7:0</td>
<td><strong>CHUNK_START_INDEX</strong>&lt;br&gt;Indicates chunk index to start from.</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15:8</td>
<td><strong>CHUNK_STOP_INDEX</strong>&lt;br&gt;Indicates what chunk index to stop at (inclusive).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>31:16</td>
<td>Reserved: all zeros.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>62:32</td>
<td><strong>THREAD_WAIT_DELAY</strong>&lt;br&gt;TSC based delay to allow threads to rendezvous.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>63</td>
<td><strong>SIGNAL_MCE</strong>&lt;br&gt;If 1, then on scan-error log MC in MC4_STATUS and signal MCE if machine check signaling enabled in MC4_CTL[0].&lt;br&gt;If 0, then no logging/no signaling.</td>
<td></td>
</tr>
<tr>
<td><strong>MSR_SCAN_STATUS (R/O)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>CHUNK_NUM</td>
<td>SCAN Chunk that was reached.</td>
<td>Core</td>
</tr>
<tr>
<td>15:8</td>
<td>CHUNK_STOP_INDEX</td>
<td>Indicates what chunk index to stop at (inclusive). Maps to same field in WRMSR(ACTIVATE_SCAN).</td>
<td>Core</td>
</tr>
<tr>
<td>31:16</td>
<td>Reserved: return all zeros.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:32</td>
<td>ERROR_CODE</td>
<td>0x0: No error. 0x1: SCAN operation did not start. Other thread did not join in time. 0x2: SCAN operation did not start. Interrupt occurred prior to threads rendezvous. 0x3: SCAN operation did not start. Power Management conditions are inadequate to run Intel In-field Scan. 0x4: SCAN operation did not start. Non-valid chunks in the range CHUNK_STOP_INDEX : CHUNK_START_INDEX. 0x5: SCAN operation did not start. Mismatch in arguments between threads T0/T1. 0x6: SCAN operation did not start. Core not capable of performing SCAN currently. 0x8: SCAN operation did not start. Exceeded number of Logical Processors (LP) allowed to run Intel In-field Scan concurrently. MAX_CORE_LIMIT exceeded. 0x9: Interrupt occurred. Scan operation aborted prematurely, not all chunks requested have been executed.</td>
<td>Thread</td>
</tr>
<tr>
<td>61:40</td>
<td>Reserved: return all zeros.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>SCAN_CONTROL_ERROR</td>
<td>Scan-System-Controller malfunction.</td>
<td>Core</td>
</tr>
<tr>
<td>63</td>
<td>SCAN_SIGNATURE_ERROR</td>
<td>Core failed SCAN-SIGNATURE checking for this chunk.</td>
<td>Core</td>
</tr>
</tbody>
</table>

Register Address: 2C8H, 712

### MSR_SCAN_MODULE_ID

<table>
<thead>
<tr>
<th>SCAN_MODULE_ID (R/O)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RevID of the currently installed scan test image. Maps to Revision field in external header (offset 4).</td>
<td>Module</td>
</tr>
</tbody>
</table>

Register Address: 2C9H, 713

### MSR_LAST_SAF_WP

<table>
<thead>
<tr>
<th>LAST_SAF_WP (R/O)</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>Provides information about the core when the last WRMSR(ACTIVATE_SCAN) was executed. Available only if enumerated in MSR_INTEGRITY_CAPABILITIES[10:9].</td>
<td>Core</td>
</tr>
</tbody>
</table>

Register Address: 2D9H, 729

### MSR_INTEGRITY_CAPABILITIES
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INTEGRITY_CAPABILITIES (R/O)</td>
<td></td>
<td>Module</td>
</tr>
<tr>
<td>0</td>
<td>STARTUP_SCAN_BIST</td>
<td>When set, supports Intel In-field Scan.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:1</td>
<td>Reserved: return all zeros.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>PERIODIC_SCAN_BIST</td>
<td>When set, supports Intel In-field Scan.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:5</td>
<td>Reserved: return all zeros.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>ID of the scan programs supported for this part. WRMISR(2C2H) verifies this value against the corresponding value in the scan-image header, i.e., Image_info.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 410H, 1040 IA32_MC4_CTL
See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”
Bank MC4 reports MC errors from the PCU module.
If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.

Register Address: 411H, 1041 IA32_MC4_STATUS
See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”
Bank MC4 reports MC errors from the PCU module.
If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.

Register Address: 412H, 1042 IA32_MC4_ADDR
See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”
Bank MC4 reports MC errors from the PCU module.
If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.

Register Address: 413H, 1043 IA32_MC4_MISC
See Section 16.3.2.1, “IA32_MCI_CTL MSRs,” through Section 16.3.2.4, “IA32_MCI_MISC MSRs.”
Bank MC4 reports MC errors from the PCU module.
If SIGNAL_MCE is set, a Scan Status is logged in MC4_STATUS and MC4_MISC.

Register Address: 492H, 1170 IA32_VMX_PROCBASED_CTLS3
Capability Reporting Register of Tertiary Processor-Based VM-Execution Controls (R/O)
See Table 2-2.

Register Address: 493H, 1171 IA32_VMX_EXIT_CTLS2
Capability Reporting Register of Secondary VM-Exit Controls (R/O)
See Table 2-2.

Register Address: 540H, 1344 MSR_THREAD_UARCH_CTL
Thread Microarchitectural Control (R/W)
See Table 2-47.

Register Address: 64DH, 1613 MSR_PLATFORM_ENERGY_STATUS
Platform Energy Status (R/O)

31:0 TOTAL_ENERGY_CONSUMED
Total energy consumption in J (32.0), in 10nsec units.
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>TIME_STAMP</td>
<td>Time stamp (U32.0).</td>
<td></td>
</tr>
<tr>
<td>65CH, 1628</td>
<td>MSR_PLATFORM_POWER_LIMIT</td>
<td>Platform Power Limit Control (R/W-L)</td>
<td>Package</td>
</tr>
<tr>
<td>16:0</td>
<td>POWER_LIMIT_1</td>
<td>The average power limit value that the platform must not exceed over a time window as specified by the Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>POWER_LIMIT_1_EN</td>
<td>When set, the processor can apply control policies such that the platform average power does not exceed the Power_Limit_1 value over an exponential weighted moving average of the time window.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>CRITICAL_POWER_CLAMP_1</td>
<td>When set, the processor can go below the OS-requested P States to maintain the power below the specified Power_Limit_1 value.</td>
<td></td>
</tr>
<tr>
<td>25:19</td>
<td>POWER_LIMIT_1_TIME</td>
<td>This indicates the time window over which the Power_Limit_1 value should be maintained. This field is made up of two numbers from the following equation: [ \text{Time Window} = (\text{float}) \left( (1+(X/4)) \right) \text{\times} 2^Y ], where: ( X = \text{POWER_LIMIT_1_TIME}[23:22] ) ( Y = \text{POWER_LIMIT_1_TIME}[21:17] ) The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is ODH, and the unit is specified in MSR_RAPL_POWER_UNIT[Time Unit].</td>
<td></td>
</tr>
<tr>
<td>31:26</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48:32</td>
<td>POWER_LIMIT_2</td>
<td>This is the Duration Power limit value that the platform must not exceed. The unit is specified in MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>Enable Platform Power Limit #2</td>
<td>When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Platform Clamping Limitation #2</td>
<td>When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.</td>
<td></td>
</tr>
<tr>
<td>57:51</td>
<td>POWER_LIMIT_2_TIME</td>
<td>This indicates the time window over which the Power_Limit_2 value should be maintained. This field has the same format as the POWER_LIMIT_1_TIME field.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>62:58</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>LOCK</td>
<td>Setting this bit will lock all other bits of this MSR until system RESET.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 665H, 1637</td>
<td>MSR_PLATFORM_POWER_INFO</td>
<td>Platform Power Information (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16:0</td>
<td>MAX_PPL1</td>
<td>Maximum PP L1 value.</td>
<td>The unit is specified in MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>31:17</td>
<td>MIN_PPL1</td>
<td>Minimum PP L1 value.</td>
<td>The unit is specified in MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>48:32</td>
<td>MAX_PPL2</td>
<td>Maximum PP L2 value.</td>
<td>The unit is specified in MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>55:49</td>
<td>MAX_TW</td>
<td>Maximum time window.</td>
<td>The unit is specified in MSR_RAPL_POWER_UNIT.</td>
<td></td>
</tr>
<tr>
<td>62:56</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td></td>
<td>LOCK</td>
<td>Setting this bit will lock all other bits of this MSR until system RESET.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 666H, 1638</td>
<td>MSR_PLATFORM_RAPL_SOCKET_PERF_STATUS</td>
<td>Platform RAPL Socket Performance Status (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td></td>
<td>Count of limited performance due to platform RAPL limit.</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A0H, 1696</td>
<td>IA32_U_CET</td>
<td>Configure User Mode CET (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A2H, 1698</td>
<td>IA32_S_CET</td>
<td>Configure Supervisor Mode CET (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A4H, 1700</td>
<td>IA32_PL0_SSP</td>
<td>Linear address to be loaded into SSP on transition to privilege level 0. (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A5H, 1701</td>
<td>IA32_PL1_SSP</td>
<td>Linear address to be loaded into SSP on transition to privilege level 1. (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A6H, 1702</td>
<td>IA32_PL2_SSP</td>
<td>Linear address to be loaded into SSP on transition to privilege level 2. (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A7H, 1703</td>
<td>IA32_PL3_SSP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families (CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear address to be loaded into SSP on transition to privilege level 3. (R/W)</td>
<td>See Table 2-2.</td>
<td>IA32_INTERRUPT_SSP_TABLE_ADDR</td>
<td>Linear address of a table of seven shadow stack pointers that are selected in IA-32e mode using the IST index (when not 0) from the interrupt gate descriptor. (R/W)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6A8H, 1704</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_PKRS</td>
<td>Specifies the PK permissions associated with each protection domain for supervisor pages (R/w)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6E1H, 1761</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Memory Encryption Capability MSR</td>
<td>See Table 2-2.</td>
<td>IA32_TME_CAPABILITY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 776H, 1910</td>
<td></td>
<td>IA32_HWP_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Interrupt Request Register (R/w)</td>
<td>See Table 2-2.</td>
<td>IA32_UINTR_RR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 985H, 2437</td>
<td></td>
<td></td>
<td>User Interrupt Request Register (R/w)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_UINTR_HANDLER</td>
<td>User Interrupt Handler Address (R/W)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 986H, 2438</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_UINTR_STACKADJUST</td>
<td>User Interrupt Stack Adjustment (R/W)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 987H, 2439</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_UINTR_MISC</td>
<td>User-Interrupt Target-Table Size and Notification Vector (R/w)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 988H, 2440</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_UINTR_PD</td>
<td>User Interrupt PID Address (R/w)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 989H, 2441</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA32_UINTR_TT</td>
<td>User-Interrupt Target Table (R/w)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 98AH, 2442</td>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR_B1_PMON_EVNT_SEL0</td>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C70H, 3184</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR_B1_PMON_CTR0</td>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C71H, 3185</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSR_B1_PMON_EVNT_SEL1</td>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C72H, 3186</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-52. Additional MSRs Supported by the 4th and 5th Generation Intel® Xeon® Scalable Processor Families
(CPUID Signature DisplayFamily_DisplayModel Values of 06_8FH and 06_CFH) (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>Register Address: C73H, 3187</td>
<td>MSR_B1_PMON_CTR1</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C74H, 3188</td>
<td>MSR_B1_PMON_EVNT_SEL2</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C75H, 3189</td>
<td>MSR_B1_PMON_CTR2</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C76H, 3190</td>
<td>MSR_B1_PMON_EVNT_SEL3</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon event select MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C77H, 3191</td>
<td>MSR_B1_PMON_CTR3</td>
</tr>
<tr>
<td>Uncore B-box 1 perfmon counter MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C82H, 3192</td>
<td>MSR_W_PMON_BOX_OVF_CTRL</td>
</tr>
<tr>
<td>Uncore W-box perfmon local box overflow control MSR.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: C8FH, 3215</td>
<td>IA32_PQR_ASSOC</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C90H−C9EH, 3216−3230</td>
<td>IA32_L3_QOS_MASK_0 through IA32_L3_QOS_MASK_14</td>
</tr>
<tr>
<td>See Table 2-50.</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: D10H−D17H, 3344−3351</td>
<td>IA32_L2_QOS_MASK_[0-7]</td>
</tr>
<tr>
<td>IA32_CR_L2_QOS_MASK_[0-7]</td>
<td>If CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] ≥ 0. See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: D93H, 3475</td>
<td>IA32_PASID</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1200H−121FH, 4608−4639</td>
<td>IA32_LBR_x_INFO</td>
</tr>
<tr>
<td>Last Branch Record Entry X Info Register (R/w)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 1406H, 5126</td>
<td>IA32_MCU_CONTROL</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 14CEH, 5326</td>
<td>IA32_LBR_CTL</td>
</tr>
<tr>
<td>Last Branch Record Enabling and Configuration Register (R/w)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 14CFH, 5327</td>
<td>IA32_LBR_DEPTH</td>
</tr>
<tr>
<td>Last Branch Record Maximum Stack Depth Register (R/w)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 1500H−151FH, 5376−5407</td>
<td>IA32_LBR_x_FROM_IP</td>
</tr>
<tr>
<td>Last Branch Record Entry X Source IP Register (R/w)</td>
<td>See Table 2-2.</td>
</tr>
</tbody>
</table>
2.17.9 MSRs Introduced in the Intel® Core™ Ultra 7 Processor Supporting Performance Hybrid Architecture

Table 2-53 lists additional MSRs for the Intel Core Ultra 7 processor with a CPUID Signature DisplayFamily_Display-Model value of 06_AAH. Table 2-54 lists the MSRs unique to the processor P-core. Table 2-55 lists the MSRs unique to the processor E-core.

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 1600H-161FH, 5632-5663</td>
<td>IA32_LBR_x_TO_IP</td>
<td>Last Branch Record Entry X Destination IP Register (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>45:18</td>
<td></td>
<td>ADDRESS</td>
<td>AET target address in Trace Hub MMIO space.</td>
<td></td>
</tr>
<tr>
<td>63:46</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
<td>C-State Configuration (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:0</td>
<td>PKG_C_STATE_LIMIT</td>
<td>Specifies the lowest processor-specific C-state code name (consuming the least power) for the package.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is set as factory-configured package C-state limit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The following C-state code name encodings may be supported:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000b: C0/C1 (no package C-state support)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001b: C2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010b: C3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011b: C6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100b: C7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101b: C7s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110b: C8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111b: C9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000b: C10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td>MAX_CORE_C_STATE</td>
<td>Possible values are: 0000—reserved; 0001—C1; 0010—C3, 0011—C6.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:8</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>IO_MWAIT_REDIRECT_ENABLE</td>
<td>When set, will map IO_read instructions sent to IO registers PMG_IO_BASE_ADDR.PMB0+0/1/2 to MWAIT(C2,3,4) instructions; applies to deepc4 too.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:11</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CFG_LOCK</td>
<td>When set, locks bits 15:0 of this register for further writes, until the next reset occurs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24:16</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>C3_STATE_AUTO_DEMOTION_ENABLE</td>
<td>When set, processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>C1_STATE_AUTO_DEMOTION_ENABLE</td>
<td>When set, processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>ENABLE_C3_UNDEMOTION</td>
<td>Enable Un-Demotion from Demoted C3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
<td><strong>Scope</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 28 | ENABLE_C1_UNDEMOTION  
Enable Un-Demotion from Demoted C1. |  
 | 29 | ENABLE_PKGC_AUTODEMOTION  
Enable Package C-State Auto-Demotion. It enables use of the history of past package C-state depth and residence, as a factor in determining C-State depth. |  
 | 30 | ENABLE_PKGC_UNDEMOTION  
Enable Package C-State Un-Demotion. It enables considering cases where demotion was the incorrect decision in determining C-State depth. |  
 | 31 | TIMED_MWAIT_ENABLE  
When set, enables Timed MWait feature. MWait would #GP on attempts to do setup MWait timer if this bit is not set. |  
 | 63:32 | Reserved. |  
| **Register Address: E4H, 228** | MSR_IO_CAPTURE_BASE  
Power Management IO Redirection in C-state. See http://biosbits.org. | Core  
| 15:0 | LVL_2_BASE_ADDRESS  
Specifies the base address visible to software for IO redirection. If MSR_PKG_CST_CONFIG_CONTROL.IO_MWAIT_REDIRECTION_ENABLE, reads to this address will be consumed by the power management logic and decoded to MWait instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software. |  
| 18:16 | CST_RANGE  
Specifies the encoding value of the maximum C-State code name to be included when IO read to MWait redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL.IO_MWAIT_REDIRECTION_ENABLE:  
000b—C3 is the max C-State to include.  
001b—C6 is the max C-State to include.  
010b—C7 is the max C-State to include. |  
| 63:19 | Reserved. |  
| **Register Address: 13CH, 316** | MSR_FEATURE_CONFIG  
AES Feature Configuration (R/w) | Core  
| 0 | AESNI_LOCK  
Once this bit is set, writes to this register will not be allowed. |  
| 1 | AESNI_DISABLE  
This bit disables Advanced Encryption Standard feature on this processor core. To disable AES, BIOS will write '11 to this MSR on every core. |  
| 63:2 | Reserved. |  
| **Register Address: 140H, 320** | MSR_FEATURE_ENABLES |  

Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Enable (R/w)</td>
<td></td>
<td>Miscellaneous enables for thread specific features.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>CPUID_GP_ON_CPL_GT_0</td>
<td>Causes CPUID to #GP if CPL greater than 0 and not in SMM.</td>
<td></td>
</tr>
<tr>
<td>63:1</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A2H, 418</td>
<td></td>
<td>MSR_TEMPERATURE_TARGET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temperature Target (R/w)</td>
<td></td>
<td>Legacy register holding temperature related constants for Platform use.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>6:0</td>
<td></td>
<td>TCC Offset Time Window</td>
<td>Describes the RATL averaging time window.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>TCC Offset Clamping Bit</td>
<td>When enabled will allow RATL throttling below P1.</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>Temperature Control Offset</td>
<td>Fan Temperature Target Offset (a.k.a. T-Control) indicates the relative offset from the Thermal Monitor Trip Temperature at which fans should be engaged.</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td></td>
<td>TCC Activation Temperature</td>
<td>The minimum temperature at which PROCHOT# will be asserted. The value is degrees C.</td>
<td></td>
</tr>
<tr>
<td>30:24</td>
<td></td>
<td>TCC Activation Offset</td>
<td>Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only if MSR_PLATFORM_INFO[30] is set.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>LOCKED</td>
<td>When set, this entire register becomes read-only.</td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A4H, 420</td>
<td></td>
<td>MSR_PREFETCH_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREFETCH Control (R/w)</td>
<td></td>
<td>Prefetch disable bits.</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>L2_HARDWARE_PREFETCHER_DISABLE</td>
<td>If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>L2_ADJACENT_CACHE_LINE_PREFETCHER_DISABLE</td>
<td>If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>DCU_HARDWARE_PREFETCHER_DISABLE</td>
<td>If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>DCU_IP_PREFETCHER_DISABLE</td>
<td>If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction pointer of previous loads) to determine whether to prefetch additional lines.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DCU_NEXT_PAGE_PREFETCH_DISABLE</td>
<td>If 1, disables Next Page prefetcher.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>AMP_PREFETCH_DISABLE</td>
<td>If 1, disables L2 Adaptive Multipath Probability (AMP) prefetcher.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LLC_PAGE_PREFETCH_DISABLE</td>
<td>If 1, disables the LLC Page prefetcher.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>AOP_PREFETCH_DISABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STREAM_PREFETCH_CODE_FETCH_DISABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A6H, 422</td>
<td>MSR_OFFCORE_RSP_0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### OFFCORE_RSP_0 (R/W)

**Offcore Response Event Select Register**

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TRUE_DEMAND_CACHE_LOAD</td>
<td>Demand Data Rd = DCU reads (includes partials) that is not tagged homeless.</td>
</tr>
<tr>
<td>1</td>
<td>DEMAND_RFO</td>
<td>Demand Instruction fetch = IFU Fetches. ItoM or RFO that is not tagged homeless.</td>
</tr>
<tr>
<td>2</td>
<td>DEMAND_CODE_READ</td>
<td>Demand Instruction fetch = IFU Fetches. CRd or CRd_UC.</td>
</tr>
<tr>
<td>3</td>
<td>CORE_MODIFIED_WRITEBACK</td>
<td>WBMtol or WBMtoE.</td>
</tr>
<tr>
<td>4</td>
<td>HW_PREFETCH_MLC_LOAD</td>
<td>L2 prefetcher requests triggered by reads from MEC (except those triggered by l-side).</td>
</tr>
<tr>
<td>5</td>
<td>HW_PREFETCH_MLC_RFO</td>
<td>L2 prefetcher requests triggered by RFOs.</td>
</tr>
<tr>
<td>6</td>
<td>HW_PREFETCH_MLC_CODE</td>
<td>L2 prefetcher requests triggered by l-side requests.</td>
</tr>
<tr>
<td>7</td>
<td>HW_PREFETCH_LLC_LOAD</td>
<td>LLC prefetch requests triggered by DRd.</td>
</tr>
<tr>
<td>8</td>
<td>HW_PREFETCH_LLC_RFO</td>
<td>LLC prefetch requests triggered by RFO.</td>
</tr>
<tr>
<td>9</td>
<td>HW_PREFETCH_LLC_CODE</td>
<td>LLC prefetch requests triggered by CRd.</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>10</td>
<td>L1_HwPREFETCH</td>
<td>Covers Hardware PFRFO, PFNEAR, PFMED, PFFAR, PFHW, PFNTA, PFNPP, PFIPP including the homeless versions.</td>
</tr>
<tr>
<td>11</td>
<td>ALL_STREAMING_STORE</td>
<td>Write Combining, WCIL or WCILF.</td>
</tr>
<tr>
<td>12</td>
<td>CORE_NON_MODIFIED_WB</td>
<td>WBEFtol or WBEFtoE.</td>
</tr>
<tr>
<td>13</td>
<td>LLC_PREFETCH</td>
<td>LLC prefetch of load/code/RFO.</td>
</tr>
<tr>
<td>14</td>
<td>L1_SwPREFETCH</td>
<td>Covers Software PFRFO, PFNEAR, PFMED, PFFAR, PFHW, PFNTA, PFNPP, PFIPP including the homeless versions.</td>
</tr>
<tr>
<td>15</td>
<td>OTHER</td>
<td>Includes CLFlush, CLFlushOPT, CLEmote, CLWB, Enqueue, SetMonitor, PortIn, IntA, Lock, SplitLock, Unlock, SpCyc, ClrMonitor, PortOut, IntPriUp, IntLog, IntPhy, EOI, RdCurr, WbStol, LLCWBlInv, LLCInv, NOP, PCCOMMIT.</td>
</tr>
<tr>
<td>16</td>
<td>ANY_RESP</td>
<td>Match on any response.</td>
</tr>
<tr>
<td>17</td>
<td>SUPPLIER_NONE</td>
<td>No Supplier Details, DATA_PRE [6:3] = 0.</td>
</tr>
<tr>
<td>18</td>
<td>LLC_HIT_M_STATE</td>
<td>LLC/L3, M-state, DATA_PRE [6:3] = 2.</td>
</tr>
<tr>
<td>23</td>
<td>FAR_MEM_REMOTE_0_HOP</td>
<td>Far Memory, Remote 0-hop, DATA_PRE [6:3] = 3.</td>
</tr>
<tr>
<td>24</td>
<td>FAR_MEM_REMOTE_1_HOP</td>
<td>Far Memory, Remote 1-hop, DATA_PRE [6:3] = 5.</td>
</tr>
<tr>
<td>27</td>
<td>NEAR_MEM_REMOTE_0_HOP</td>
<td>Near Memory, Remote 0-hop, DATA_PRE [6:3] = B</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Register Information / Bit Fields</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>28</td>
<td>NEAR_MEM_REMOTE_1_HOP</td>
<td>Near Memory, Remote 1-hop, DATA_PRE [6:3] = D.</td>
</tr>
<tr>
<td>29</td>
<td>NEAR_MEM_REMOTE_2_PLUS_HOP</td>
<td>Near Memory, Remote 2+ hop, DATA_PRE [6:3] = F.</td>
</tr>
<tr>
<td>31</td>
<td>SNOOP_NONE</td>
<td>No details as to Snoop-related info. Snoop Info: None, DATA_PRE [2:0] = 0.</td>
</tr>
<tr>
<td>32</td>
<td>NOT_NEEDED</td>
<td>No snoop was needed to satisfy the request. Snoop Info: Not needed, DATA_PRE [2:0] = 1.</td>
</tr>
<tr>
<td>33</td>
<td>MISS</td>
<td>No snoop was needed to satisfy the request. Snoop Info: Miss, DATA_PRE [2:0] = 2.</td>
</tr>
<tr>
<td>34</td>
<td>HIT_NO_FWD</td>
<td>A snoop was needed and it Hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. Snoop Info: Hit No Fwd, DATA_PRE [2:0] = 3.</td>
</tr>
<tr>
<td>35</td>
<td>HIT_EF_WITH_FWD</td>
<td>A snoop was needed and data was Forwarded from a remote socket. Snoop Info: Hit EF w/Fwd, DATA_PRE [2:0] = 4.</td>
</tr>
<tr>
<td>36</td>
<td>HITM</td>
<td>A snoop was needed and it HitMed in local or remote cache. HitM denotes a cache-line was modified before snoop effect. Snoop Info: HitM, DATA_PRE [2:0] = 5.</td>
</tr>
<tr>
<td>37</td>
<td>NON_DRAM</td>
<td>Target was non-DRAM system address. Snoop Info: HitM, DATA_PRE [2:0] = 5.</td>
</tr>
<tr>
<td>38</td>
<td>GO_ERR</td>
<td>GO-ERR, RspData[3:0] = 0100.</td>
</tr>
<tr>
<td>39</td>
<td>GO_NO_GO</td>
<td>GO-NoGO, RspData[3:0] = 0111.</td>
</tr>
<tr>
<td>41</td>
<td>INPKG_MEM_NONLOCAL</td>
<td>In-package Memory, Non-Local, DATA_PRE [6:3] = C.</td>
</tr>
<tr>
<td>43:42</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>44</td>
<td>UC_LOAD</td>
<td>PRd or UCRdF.</td>
</tr>
<tr>
<td>45</td>
<td>UC_STORE</td>
<td>WiL</td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>46</td>
<td>PARTIAL_STREAMING_STORES WCiL.</td>
</tr>
<tr>
<td>47</td>
<td>FULL_STREAMING_STORES WCiLF.</td>
</tr>
<tr>
<td>48</td>
<td>L1_MODIFIED_WB EVICTION EXTTYPE from MEC.</td>
</tr>
<tr>
<td>49</td>
<td>L2_MODIFIED_WB WBMTol or WBMTolE.</td>
</tr>
<tr>
<td>50</td>
<td>PSMI MemPushWr_NS (PSMI only).</td>
</tr>
<tr>
<td>51</td>
<td>ITOM ItoM.</td>
</tr>
<tr>
<td>63:52</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1A7H, 423</td>
<td>MSR_OFFCORE_RSP_1</td>
</tr>
<tr>
<td>OFFCORE_RSP_1 (R/W)</td>
<td>Offcore Response Event Select Register. See MSR_OFFCORE_RSP_0 (at 1A6H).</td>
</tr>
<tr>
<td>Register Address: 1AAH, 426</td>
<td>MSR_MISC_PWR_MGMT</td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>1</td>
<td>ENABLE_HWP_VOTING_RIGHT</td>
</tr>
<tr>
<td>When set (1), The CPU will take into account thread HWP requests for threads that have voting rights only (ignores thread requests if they do not have voting rights). When reset(0), The CPU will take into account all thread HWP requests, even for threads that don't have voting rights. Setting this bit will cause the HWP Base feature bit to be reported in CPUID as present; clearing will cause it to be reported as non-present.</td>
<td></td>
</tr>
<tr>
<td>5:2</td>
<td>Reserved.</td>
</tr>
<tr>
<td>6</td>
<td>ENABLE_HWP</td>
</tr>
<tr>
<td>Setting this bit will cause the HWP Base feature bit to report as present in CPUID; clearing this bit will cause CPUID to report the feature as non-present.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ENABLE_HWP_INTERRUPT</td>
</tr>
<tr>
<td>Setting this bit will cause the HWP Interrupt feature CPUID[6].EAX[8] bit to report as present; clearing will report as non-present.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ENABLE_OUT_OF_BAND_AUTONOMOUS</td>
</tr>
<tr>
<td>Setting this bit will cause the HWP Autonomous feature bit to report as present; clearing will report as non-present.</td>
<td></td>
</tr>
<tr>
<td>11:9</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>** ENABLE_HWP_EPP**</td>
<td>12</td>
<td>Enable HWP EPP. Setting this bit (1) will cause the HWP CPUID[6].EAX[10] Energy Performance Preference bit to report as present (1); clearing will report as non-present (0).</td>
</tr>
<tr>
<td>** LOCK**</td>
<td>13</td>
<td>Setting this bit will prevent the BIOS specific bits from changing until the next reset, i.e., only Bits [0,22] which are meant for OS use can be changed once the LOCK bit is set.</td>
</tr>
<tr>
<td>** Reserved.**</td>
<td>63:14</td>
<td>Register Address: 1ADH, 429</td>
</tr>
</tbody>
</table>

**MSR_PRIMARY_TURBO_RATIO_LIMIT**

Primary Maximum Turbo Ratio Limit (R/W)
Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.

<table>
<thead>
<tr>
<th>Register Address: 1F1H, 497</th>
<th>MSR_CRASHLOG_CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>** Crash Log Control (R/W)**</td>
<td>Write data to a Crash Log configuration.</td>
</tr>
<tr>
<td>0</td>
<td>CDDIS</td>
</tr>
<tr>
<td></td>
<td>CrashDump_Disable: If set, indicates that Crash Dump is disabled.</td>
</tr>
<tr>
<td>1</td>
<td>EN_GPRS</td>
</tr>
<tr>
<td></td>
<td>Collect GPRs on a crash dump. Only meaningful when CDDIS is zero.</td>
</tr>
<tr>
<td>2</td>
<td>EN_GPRS_IN_SMM</td>
</tr>
<tr>
<td></td>
<td>Collect GPRs in SMM on a crash dump. Only meaningful when CDDIS is zero. EN_GPRS will override this control.</td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>3</td>
<td>TRIPLE_FAULT_SHUTDOWN</td>
</tr>
<tr>
<td></td>
<td>Collect a crash log on a triple fault shutdown. Only meaningful when CDDIS is zero.</td>
</tr>
<tr>
<td>63:4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1F5H, 501</td>
<td>MSR_PRMRR_PHYS_MASK</td>
</tr>
<tr>
<td>Processor Reserved Memory Range Register - Physical Mask (R/W)</td>
<td></td>
</tr>
<tr>
<td>9:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>10</td>
<td>LOCK</td>
</tr>
<tr>
<td></td>
<td>Once set, this bit prevents software from modifying the PRMRR.</td>
</tr>
<tr>
<td>11</td>
<td>VALID</td>
</tr>
<tr>
<td></td>
<td>This bit serves as the enable for the PRMRR; the PRMRR must be LOCKed before it can be enabled.</td>
</tr>
<tr>
<td>19:12</td>
<td>Reserved.</td>
</tr>
<tr>
<td>45:20</td>
<td>MASK</td>
</tr>
<tr>
<td></td>
<td>PRMRR Address Mask.</td>
</tr>
<tr>
<td>63:46</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1FCH, 508</td>
<td>MSR_POWER_CTL</td>
</tr>
<tr>
<td>Power Control Register (R/W)</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>17:4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>18</td>
<td>PWR_PERF_PLTFRM_OVR</td>
</tr>
<tr>
<td>19</td>
<td>EE_TURBO_DISABLE</td>
</tr>
<tr>
<td>20</td>
<td>RTH_DISABLE</td>
</tr>
<tr>
<td>21</td>
<td>DIS_PROCHOT_OUT</td>
</tr>
<tr>
<td>22</td>
<td>PROCHOT_RESPONSE</td>
</tr>
<tr>
<td>23</td>
<td>VR_THERM_ALERT_DISABLE_LOCK</td>
</tr>
<tr>
<td>24</td>
<td>VR_THERM_ALERT_DISABLE</td>
</tr>
<tr>
<td>25</td>
<td>DISABLE_RING_EE</td>
</tr>
<tr>
<td>26</td>
<td>DISABLE_SA_OPTIMIZATION</td>
</tr>
<tr>
<td>27</td>
<td>DISABLE_OOK</td>
</tr>
<tr>
<td>28</td>
<td>DISABLE_AUTONOMOUS</td>
</tr>
<tr>
<td>29</td>
<td>Reserved.</td>
</tr>
<tr>
<td>30</td>
<td>CSTATE_PREWAKE_DISABLE</td>
</tr>
<tr>
<td>63:31</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Register Address: 2A0H, 672

MSR_PRMRR_BASE_0

Processor Reserved Memory Range Register - Physical Base Control Register (R/W) | Core
---|---|
2:0 | MEMTYPE | Memory type for PRMRR accesses.
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>CONFIGURED</td>
<td>PRMRR base configured.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45:20</td>
<td>BASE</td>
<td>PRMRR base address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:46</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 474H, 1140</td>
<td>IA32_MC29_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC29_CTL. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 475H, 1141</td>
<td>IA32_MC29_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC29_STATUS. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 476H, 1142</td>
<td>IA32_MC29_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC29_ADDR. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 477H, 1143</td>
<td>IA32_MC29_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC29_MISC. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 478H, 1144</td>
<td>IA32_MC30_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC30_CTL. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 479H, 1145</td>
<td>IA32_MC30_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC30_STATUS. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 47AH, 1146</td>
<td>IA32_MC30_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC30_ADDR. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 47BH, 1147</td>
<td>IA32_MC30_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC30_MISC. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 47CH, 1148</td>
<td>IA32_MC31_CTL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC31_CTL. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 47DH, 1149</td>
<td>IA32_MC31_STATUS</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC31_STATUS. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 47EH, 1150</td>
<td>IA32_MC31_ADDR</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC31_ADDR. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 47FH, 1151</td>
<td>IA32_MC31_MISC</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>MC31_MISC. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 4E0H, 1248</td>
<td>MSR_SMM_FEATURE_CONTROL</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Enhanced SMM Feature Control (R/W)</td>
<td>Reports SMM capability enhancement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LOCK</td>
<td>When set, locks this register from further changes.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SMM_CPU_SAVE_EN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If 0, SMI/RSM will save/restore state in SMRAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If 1, SMI/RSM will save/restore state from SRAM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SMM_CODE_CHK_EN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When clear (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set, any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 601H, 1537</td>
<td>MSR_VR_CURRENT_CONFIG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Limit 4 (PL4) (R/W)</td>
<td>Package-level maximum power limit (in Watts). It is a proactive, instantaneous limit.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:0</td>
<td>CURRENT_LIMIT</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>PL4 Value in 0.125 A increments. This field is locked by MSR_VR_CURRENT_CONFIG.LOCK. When the LOCK bit is set to 1, this field becomes Read Only.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>LOCK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This bit will lock the CURRENT_LIMIT settings in this register and will also lock this setting. This means that once set to 1, the CURRENT_LIMIT setting and this bit become Read Only until the next Warm Reset.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 620H, 1568</td>
<td>MSR_UNCORE_RATIO_LIMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncore Ratio Limit (R/w)</td>
<td>Min/Max Ratio Limits for Uncore LLC and Ring.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:0</td>
<td>MAX_CLR_RATIO</td>
<td></td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>Maximum allowed ratio for the Ring and Last Level Cache (LLC).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:8</td>
<td>MIN_CLR_RATIO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minimum allowed ratio for the Ring and Last Level Cache (LLC).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:15</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 638H, 1592</td>
<td>MSR_PP0_POWER_LIMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR_PP0_POWER_LIMIT (R/w)</td>
<td>PPO RAPL power unit control.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:0</td>
<td>IA_PP_PWR_LIM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This is the power limitation on the IA cores power plane. The unit of measurement is defined in PACKAGE_POWERSKU_UNIT_MSR[PWR_UNIT].</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### MODEL-SPECIFIC REGISTERS (MSRS)

#### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
</table>
| 15                            | PWR_LIM_CTRL_EN | This bit must be set in order to limit the power of the IA cores power plane.  
0b: IA cores power plane power limitation is disabled.  
1b: IA cores power plane power limitation is enabled. |          |
| 16                            | PP_CLAMP_LIM  | Power Plane Clamping limitation; allow going below P1.  
0b: PBM is limited between P1 and P0.  
1b: PBM can go below P1. |          |
| 23:17                         | CTRL_TIME_WIN | x = CTRL_TIME_WIN[23:22]  
y = CTRL_TIME_WIN[21:17]  
The timing interval window is Floating Point number given by 1.x * power(2,y).  
The unit of measurement is defined in PACKAGE_POWER_SKU_UNIT_MSR[TIME_UNIT].  
The maximal time window is bounded by PACKAGE_POWER_SKU_MSR[PKG_MAX_WIN]. The minimum time window is 1 unit of measurement (as defined above). |          |
| 30:24                         | Reserved.     |                 |       |
| 31                            | PP_PWR_LIM_LOCK | When set, all settings in this register are locked and are treated as Read Only. |          |
| 63:32                         | Reserved.     |                 |       |

Register Address: 64FH, 1615

**MSR_CORE_PERF_LIMIT_REASONS**

Core Performance Limit Reasons
Indicator of Frequency Clipping in Processor Cores. (Frequency refers to processor core frequency.)

<table>
<thead>
<tr>
<th></th>
<th>Package</th>
</tr>
</thead>
</table>
| 0 | PROCHOT (R/O)  
PROCHOT Status. When set, frequency is reduced below the operating system request due to assertion of external PROCHOT. |
| 1 | THERMAL (R/O)  
Thermal Status. When set, frequency is reduced below the operating system request due to a thermal event. |
| 3:2 | Reserved. |
| 4 | RSR_LIMIT (R/O)  
Residency State Regulation Status. When set, frequency is reduced below the operating system request due to residency state regulation limit. |
| 5 | RATL (R/O)  
Running Average Thermal Limit Status. When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL). |
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| **6** | VR_THERMALERT (R/O)  
When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR). |
| **7** | VR_TDC (R/O)  
When set, frequency is reduced below the operating system request due to VR thermal design current limit. |
| **8** | OTHER (R/O)  
Other Status. When set, frequency is reduced below the operating system request due to electrical or other constraints. |
| **9** | Reserved. |
| **10** | PBM_PL1 (R/O)  
Package/Platform-Level Power Limiting PL1 Status. When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1. |
| **11** | PBM_PL2 (R/O)  
Package/Platform-Level PL2 Power Limiting Status. When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3. |
| **12** | MAX_TURBO_LIMIT (R/O)  
Max Turbo Limit Status. When set, frequency is reduced below the operating system request due to multi-core turbo limits. |
| **13** | TURBO_ATTEN (R/O)  
Turbo Transition Attenuation Status. When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes. |
| **15:14** | Reserved. |
| **16** | PROCHOT_LOG (R/w)  
PROCHOT Log. When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0. |
| **17** | THERMAL_LOG (R/w)  
Thermal Log. When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0. |
| **19:18** | Reserved. |
| **20** | RSR_LIMIT_LOG (R/w)  
Residency State Regulation Log. When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0. |
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>RATL_LOG (R/W)</td>
<td>Running average thermal limit Log, RW, When set by PCODE indicates that Running average thermal limit has cause IA frequency clipping. Software should write to this bit to clear the status in this bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>VR_THERMALERT_LOG (R/W)</td>
<td>VR Therm Alert Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>VR_TDC_LOG (R/W)</td>
<td>VR Thermal Design Current Log. When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>OTHER_LOG (R/W)</td>
<td>Other Log. When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>PBM_PL1_LOG (R/W)</td>
<td>Package/Platform-Level PL1 Power Limiting Log. When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>PBM_PL2_LOG (R/W)</td>
<td>Package/Platform-Level PL2 Power Limiting Log. When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>MAX_TURBO_LIMIT_LOG (R/W)</td>
<td>Max Turbo Limit Log. When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>TURBO_ATTEN_LOG (R/W)</td>
<td>Turbo Transition Attenuation Log. When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 650H, 1616</td>
<td>MSR_SECONDARY_TURBO_RATIO_LIMIT</td>
<td>Secondary Maximum Turbo Ratio Limit (R/W)</td>
<td>Software can configure these limits when MSR_PLATFORM_INFO[28] = 1. Specifies Maximum Ratio Limit for each group. Maximum ratio for groups with more cores must decrease monotonically.</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>MAX_TURBO_GROUP_0:</td>
<td>Maximum turbo ratio limit with 1 core active.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:8</td>
<td>MAX_TURBO_GROUP_1:</td>
<td>Maximum turbo ratio limit with 2 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>MAX_TURBO_GROUP_2:</td>
<td>Maximum turbo ratio limit with 3 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>MAX_TURBO_GROUP_3:</td>
<td>Maximum turbo ratio limit with 4 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:32</td>
<td>MAX_TURBO_GROUP_4:</td>
<td>Maximum turbo ratio limit with 5 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td>MAX_TURBO_GROUP_5:</td>
<td>Maximum turbo ratio limit with 6 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55:48</td>
<td>MAX_TURBO_GROUP_6:</td>
<td>Maximum turbo ratio limit with 7 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:56</td>
<td>MAX_TURBO_GROUP_7:</td>
<td>Maximum turbo ratio limit with 8 cores active.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 65CH, 1628</td>
<td>MSR_PLATFORM_POWER_LIMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Platform Power Limit Control (R/W)

Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.

| 14:0 | POWER_LIMIT_1 | Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (a.k.a TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT. | Package |
| 15   | POWER_LIMIT_1_EN | When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit 1 over the time window specified by Power Limit 1 Time Window. | |
| 16   | CRITICAL_POWER_CLAMP_1 | When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit 1 value. | |
Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
</tbody>
</table>
| **23:17** | POWER_LIMIT_1_TIME | Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation:  
Time Window = (float) ((1+(X/4))*(2^Y)), where:  
X = POWER_LIMIT_1_TIME[23:22]  
Y = POWER_LIMIT_1_TIME[21:17]  
The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN].  
The default value is 0DH, The unit is specified in MSR_RAPLPOWER_UNIT[Time Unit] |
<p>| <strong>31:24</strong> | Reserved. | |
| <strong>46:32</strong> | POWER_LIMIT_2 | Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e., Platform Power Limit 1). |
| <strong>47</strong> | POWER_LIMIT_2_EN | When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit 2 over the Short Duration time window. |
| <strong>48</strong> | CRITICAL_POWER_CLAMP_2 | When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit 2 value. |
| <strong>62:49</strong> | Reserved. | |
| <strong>63</strong> | LOCK | Setting this bit will lock all other bits of this MSR until system RESET. |
| <strong>Register Address: 6BOH, 1712</strong> | MSR_GRAPHICS_PERF_LIMIT_REASONS | |
| <strong>MSR_GRAPHICS_PERF_LIMIT_REASONS</strong> | Indicator of Frequency Clipping in the Processor Graphics. (Frequency refers to processor graphics frequency.) | Package |
| <strong>0</strong> | PROCHOT (R/O) | PROCHOT Status. When set, frequency is reduced due to assertion of external PROCHOT. |
| <strong>1</strong> | THERMAL (R/O) | Thermal Status. When set, frequency is reduced due to a thermal event. |
| <strong>4:2</strong> | Reserved. | |
| <strong>5</strong> | RATL (R/O) | Running Average Thermal Limit Status. When set, frequency is reduced due to running average thermal limit. |</p>
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6 VR_THERMALERT (R/O)</td>
<td>VR Therm Alert Status. When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7 VR_TDC (R/O)</td>
<td>VR Thermal Design Current Status. When set, frequency is reduced due to VR TDC limit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8 OTHER (R/O)</td>
<td>Other Status. When set, frequency is reduced due to electrical or other constraints.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10 PBM_PL1 (R/O)</td>
<td>Package/Platform-Level Power Limiting PL1 Status. When set, frequency is reduced due to package/platform-level power limiting PL1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11 PBM_PL2 (R/O)</td>
<td>Package/Platform-Level PL2 Power Limiting Status. When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12 INEFFICIENT_OPERATION (R/O)</td>
<td>Inefficient Operation Status. When set, processor graphics frequency is operating below target frequency.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:13</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16 PROCHOT_LOG (R/W)</td>
<td>PROCHOT Log. When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>17 THERMAL_LOG (R/W)</td>
<td>Thermal Log. When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20:18</td>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>21 RATL_LOG (R/W)</td>
<td>Running Average Thermal Limit Log. When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>22 VR_THERMALERT_LOG (R/W)</td>
<td>VR Therm Alert Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>23 VR_TDC_LOG (R/W)</td>
<td>VR Thermal Design Current Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>OTHER_LOG (R/W)</td>
<td>Other Log. When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>PBM_PL1_LOG (R/W)</td>
<td>Package/Platform-Level PL1 Power Limiting Log. When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>PBM_PL2_LOG (R/W)</td>
<td>Package/Platform-Level PL2 Power Limiting Log. When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>INEFFICIENT_OPERATION_LOG (R/W)</td>
<td>Inefficient Operation Log. When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>63:29</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6B1H, 1713</td>
<td>MSR_RING_PERF_LIMIT_REASONS</td>
<td>MSR_RING_PERF_LIMIT_REASONS</td>
<td></td>
</tr>
</tbody>
</table>

MSR_RING_PERF_LIMIT_REASONS
Indicator of Frequency Clipping in the Ring Interconnect. (Frequency refers to ring interconnect in the uncore.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PROCHOT (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>PROCHOT Status. When set, frequency is reduced due to assertion of external PROCHOT.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>THERMAL (R/O)</td>
<td></td>
</tr>
<tr>
<td>Thermal Status. When set, frequency is reduced due to a thermal event.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:2</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RATL (R/O)</td>
<td></td>
</tr>
<tr>
<td>Running Average Thermal Limit Status. When set, frequency is reduced due to running average thermal limit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VR_THERMALERT (R/O)</td>
<td>VR Therm Alert Status. When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.</td>
</tr>
<tr>
<td>7</td>
<td>VR_TDC (R/O)</td>
<td></td>
</tr>
<tr>
<td>VR Thermal Design Current Status. When set, frequency is reduced due to VR TDC limit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>OTHER (R/O)</td>
<td></td>
</tr>
<tr>
<td>Other Status. When set, frequency is reduced due to electrical or other constraints.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>PBM_PL1 (R/O)</td>
<td>Package/Platform-Level Power Limiting PL1 Status. When set, frequency is reduced due to package/platform-level power limiting PL1.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>PBM_PL2 (R/O)</td>
<td>Package/Platform-Level PL2 Power Limiting Status. When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.</td>
<td></td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>PROCHOT_LOG (R/W)</td>
<td>PROCHOT Log. When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>THERMAL_LOG (R/W)</td>
<td>Thermal Log. When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>20:18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>RATL_LOG (R/W)</td>
<td>Running Average Thermal Limit Log. When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>VR_THERMALERT_LOG (R/W)</td>
<td>VR Therm Alert Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>VR_TDC_LOG (R/W)</td>
<td>VR Thermal Design Current Log. When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>OTHER_LOG (R/W)</td>
<td>Other Log. When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>PBM_PL1_LOG (R/W)</td>
<td>Package/Platform-Level PL1 Power Limiting Log. When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.</td>
<td></td>
</tr>
</tbody>
</table>
MODEL-SPECIFIC REGISTERS (MSRS)

The MSRs listed in Table 2-54 are unique to the Intel Core Ultra 7 processor P-core. These MSRs are not supported on the processor E-core.

### Table 2-53. Additional MSRs Supported by the Intel® Core™ Ultra 7 Processors Supporting Performance Hybrid Architecture (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>27</td>
<td>PBM_PL2_LOG (R/W)</td>
</tr>
<tr>
<td>63:28</td>
<td>Reserved</td>
</tr>
<tr>
<td>Register Address: 9FBH, 2555</td>
<td>IA32_TME_CLEAR_SAVED_KEY</td>
</tr>
<tr>
<td>IA32_TME_CLEAR_SAVED_KEY (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 9FFH, 2559</td>
<td>MSR_CORE_MKTME_ACTIVATE</td>
</tr>
<tr>
<td>MSR_CORE_MKTME_ACTIVATE (R/O)</td>
<td>Core</td>
</tr>
<tr>
<td>MSR to read TME_ACTIVATE[MK_TMEKEYID_BITS].</td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>35:32</td>
<td>READ_MK_TMEKEYID_BITS</td>
</tr>
<tr>
<td>63:36</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### Table 2-54. MSRs Supported by the Intel® Core™ Ultra 7 Processor P-core

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>Register Address: 30CH, 780</td>
<td>IA32_FIXED_CTR3</td>
</tr>
<tr>
<td>Fixed-Function Performance Counter 3 (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>47:0</td>
<td>FIXED_COUNTER</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved</td>
</tr>
<tr>
<td>Register Address: 329H, 809</td>
<td>MSR_PERF_METRICS</td>
</tr>
<tr>
<td>Performance Metrics (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>This register provides built-in support for Top-down Micro-architecture Analysis (TMA) metrics. It exposes the four TMA Level 1 metrics where the lower 32 bits are divided into four 8 bit fields, each of which is an integer percentage of the total TOPDOWN.SLOTS (as reported by fixed counter 3).</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>RETIRING</td>
</tr>
<tr>
<td>15:8</td>
<td>BAD_SPECULATION</td>
</tr>
</tbody>
</table>
The MSRs listed in Table 2-48 are unique to the Intel Core Ultra 7 processor E-core. These MSRs are not supported on the processor P-core.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRONTEND_BOUND</td>
<td>Percent of Unutilized slots where Front-end did not deliver a uop while Back-end is ready.</td>
<td></td>
</tr>
<tr>
<td>BACKEND_BOUND</td>
<td>Percent of Unutilized slots where a uop was not delivered to Back-end due to lack of Back-end resources.</td>
<td></td>
</tr>
<tr>
<td>MULTI_UOPS</td>
<td>Frontend bound.</td>
<td></td>
</tr>
<tr>
<td>BRANCH_MISPREDICTS</td>
<td>Frontend bound.</td>
<td></td>
</tr>
<tr>
<td>FRONTEND_LATENCY</td>
<td>Frontend bound.</td>
<td></td>
</tr>
<tr>
<td>MEMORY_BOUND</td>
<td>Frontend bound.</td>
<td></td>
</tr>
<tr>
<td>MSR_THREAD_UARCH_CTL</td>
<td>Thread Microarchitectural Control (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td>MSR_CORE_UARCH_CTL</td>
<td>Core Microarchitecture Control MSR (R/W)</td>
<td>Core</td>
</tr>
</tbody>
</table>

The MSRs listed in Table 2-48 are unique to the Intel Core Ultra 7 processor E-core. These MSRs are not supported on the processor P-core.

Table 2-55. MSRs Supported by the Intel® Core™ Ultra 7 Processor E-core

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_SAF_CTRL</td>
<td>SAF Control (W/O) Extension to SAF.</td>
<td>Package</td>
</tr>
<tr>
<td>INVALIDATE_CURRENT_STRIDE</td>
<td>Invalidate all chunks in current stride.</td>
<td></td>
</tr>
<tr>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_L2_MASK_[8-15]</td>
<td>Controls MLC (L2) Intel RDT allocation. For more details on CAT/RDT, see Chapter 18, &quot;Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features.&quot;</td>
<td>Module</td>
</tr>
<tr>
<td>WAY_MASK</td>
<td>Capacity Bit Mask. Available ways vectors for class of service of IA core. '1' in bit indicates allocation to the way is allowed. '0' indicates allocation to the way is not allowed.</td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 4F0H, 1264
See Table 2-47.

Register Address: D18H−D1FH, 3352−3359
See Table 2-44.
### 2.18 MSRS IN THE INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES AND THE INTEL® XEON PHI™ PROCESSOR 7215/7285/7295 SERIES

The Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with a CPUID Signature DisplayFamily_DisplayModel value of 06_57H, supports the MSR interfaces listed in Table 2-56. These processors are based on the Knights Landing microarchitecture. The Intel® Xeon Phi™ processor 7215, 7285, 7295 series, with a CPUID Signature DisplayFamily_DisplayModel value of 06_85H, supports the MSR interfaces listed in Table 2-56 and Table 2-57. These processors are based on the Knights Mill microarchitecture. Some MSRs are shared between a pair of processor cores, and the scope is marked as module.

#### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_P5_MC_ADDR</td>
<td></td>
</tr>
</tbody>
</table>

---

### Table 2-55. MSRs Supported by the Intel® Core™ Ultra 7 Processor E-core (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_PMCx</td>
<td>MSR_RELOAD_PMCx</td>
</tr>
<tr>
<td>47:0</td>
<td>Value loaded into IA32_PMCx when a PEBS record is generated while PEBS_EN_PMCx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and PMCx is overflowed.</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
</tr>
<tr>
<td>IA32_FIXED_CTRx</td>
<td>MSR_RELOAD_FIXED_CTRx</td>
</tr>
<tr>
<td>47:0</td>
<td>Value loaded into IA32_FIXED_CTRx when a PEBS record is generated while PEBS_EN_FIXEDx = 1 and PEBS_OUTPUT = 01B in IA32_PEBS_ENABLE, and FIXED_CTRx is overflowed.</td>
</tr>
<tr>
<td>31:16</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

---

Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_P5_MC_ADDR</td>
<td></td>
</tr>
</tbody>
</table>

---
Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature
DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1H, 1</td>
<td>IA32_P5_MC_TYPE</td>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>6H, 6</td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td>See Section 9.10.5, “Monitor/Mwait Address Range Determination.” See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
<td>See Section 18.17, “Time-Stamp Counter,” and Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>1BH, 27</td>
<td>IA32_APIC_BASE</td>
<td>See Section 11.4.4, “Local APIC Status and Location,” and Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>34H, 52</td>
<td>MSR_SMI_COUNT</td>
<td>See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td>See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>3BH, 59</td>
<td>IA32_TSC_ADJUST</td>
<td>See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>4EH, 78</td>
<td>IA32_PPIN_CTL (MSR_PPIN_CTL)</td>
<td>See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>4FH, 79</td>
<td>IA32_PPIN (MSR_PPIN)</td>
<td>See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>79H, 121</td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td>See Table 2-2.</td>
<td>Module</td>
<td></td>
</tr>
</tbody>
</table>


## Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOS Update Trigger Register (W)</td>
<td>IA32_BIOS_SIGN_ID</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>BIOS Update Signature ID (R/W)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>IA32_PMC0</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>IA32_PMC1</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Platform Information</td>
<td>MSR_PLATFORM_INFO</td>
</tr>
<tr>
<td>Contains power management and other model specific features enumeration. See <a href="http://biosbits.org">http://biosbits.org</a>.</td>
<td>Package</td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Non-Turbo Ratio (R/O)</td>
</tr>
<tr>
<td>This is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.</td>
<td>Package</td>
</tr>
<tr>
<td>27:16</td>
<td>Reserved.</td>
</tr>
<tr>
<td>28</td>
<td>Programmable Ratio Limit for Turbo Mode (R/O)</td>
</tr>
<tr>
<td>When set to 1, indicates that Programmable Ratio Limit for Turbo mode is enabled. When set to 0, indicates Programmable Ratio Limit for Turbo mode is disabled.</td>
<td>Package</td>
</tr>
<tr>
<td>29</td>
<td>Programmable TDP Limit for Turbo Mode (R/O)</td>
</tr>
<tr>
<td>When set to 1, indicates that TDP Limit for Turbo mode is programmable. When set to 0, indicates TDP Limit for Turbo mode is not programmable.</td>
<td>Package</td>
</tr>
<tr>
<td>39:30</td>
<td>Reserved.</td>
</tr>
<tr>
<td>47:40</td>
<td>Maximum Efficiency Ratio (R/O)</td>
</tr>
<tr>
<td>This is the minimum ratio (maximum efficiency) that the processor can operate, in units of 100MHz.</td>
<td>Package</td>
</tr>
<tr>
<td>63:48</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: E2H, 226</td>
<td>MSR_PKG_CST_CONFIG_CONTROL</td>
</tr>
<tr>
<td>C-State Configuration Control (R/W)</td>
<td>Package</td>
</tr>
</tbody>
</table>
Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>2:0</td>
<td>Package C-State Limit (R/W)</td>
</tr>
<tr>
<td></td>
<td>000b - C0/C1 (No package C-state support)</td>
</tr>
<tr>
<td></td>
<td>001b - C2</td>
</tr>
<tr>
<td></td>
<td>010b - C6 (non retention)*</td>
</tr>
<tr>
<td></td>
<td>011b - C6 (Retention)*</td>
</tr>
<tr>
<td></td>
<td>100b - Reserved</td>
</tr>
<tr>
<td></td>
<td>101b - Reserved</td>
</tr>
<tr>
<td></td>
<td>110b - Reserved</td>
</tr>
<tr>
<td></td>
<td>111b - No package C-state limit. All C-States supported by the processor are available.</td>
</tr>
<tr>
<td></td>
<td>Note: C6 retention mode provides more power saving than C6 non-retention mode. Limiting the package to C6 non retention mode does prevent the MSR_PKG_C6_RESIDENCY counter (MSR 3F9h) from being incremented.</td>
</tr>
<tr>
<td>9:3</td>
<td>Reserved.</td>
</tr>
<tr>
<td>10</td>
<td>I/O MWAIT Redirection Enable (R/W)</td>
</tr>
<tr>
<td>14:11</td>
<td>Reserved.</td>
</tr>
<tr>
<td>15</td>
<td>CFG Lock (R/O)</td>
</tr>
<tr>
<td>25</td>
<td>Reserved.</td>
</tr>
<tr>
<td>26</td>
<td>C1 State Auto Demotion Enable (R/w)</td>
</tr>
<tr>
<td>27</td>
<td>Reserved.</td>
</tr>
<tr>
<td>28</td>
<td>C1 State Auto Undemotion Enable (R/w)</td>
</tr>
<tr>
<td>29</td>
<td>PKG C-State Auto Demotion Enable (R/w)</td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: E4H, 22B</td>
<td>MSR_PKG_IO_CAPTURE_BASE</td>
</tr>
<tr>
<td>Power Management IO Capture Base (R/W)</td>
<td>Tile</td>
</tr>
</tbody>
</table>
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature

**DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:0</td>
<td>LVL_2 Base Address (R/W)</td>
<td>Microcode will compare IO-read zone to this base address to determine if an MWAIT(C2/3/4) needs to be issued instead of the IO-read. Should be programmed to the chipset Plevel_2 IO address.</td>
<td></td>
</tr>
<tr>
<td>22:16</td>
<td>C-State Range (R/W)</td>
<td>The IO-port block size in which IO-redirection will be executed (0-127). Should be programmed based on the number of LVLx registers existing in the chipset.</td>
<td></td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: E7H, 231</td>
<td>IA32_MPERF</td>
<td>Maximum Performance Frequency Clock Count (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: E8H, 232</td>
<td>IA32_APERF</td>
<td>Actual Performance Frequency Clock Count (R/W)</td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td>IA32_MTRRCAP</td>
<td>Memory Type Range Register (R)</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 13CH, 316</td>
<td>MSR_FEATURE_CONFIG</td>
<td>AES Configuration (Rw-L)</td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.</td>
<td></td>
</tr>
<tr>
<td>1:0</td>
<td>AES Configuration (RW-L)</td>
<td>Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. Otherwise, AES instructions are available. Note, the AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instructions can be mis-configured if a privileged agent unintentionally writes 11b.</td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 140H, 320</td>
<td>MISC_FEATURE_ENABLES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MISC_FEATURE_ENABLES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>1</td>
<td>User Mode MONITOR and MWAIT (R/w)</td>
<td>If set to 1, the MONITOR and MWAIT instructions do not cause invalid-opcode exceptions when executed with CPL &gt; 0 or in virtual-8086 mode. If MWAIT is executed when CPL &gt; 0 or in virtual-8086 mode, and if EAX indicates a C-state other than C0 or C1, the instruction operates as if EAX indicated the C-state C1.</td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>
### Model-Specific Registers (MSRs)

#### Register Information / Bit Fields

**Register Address:** 175H, 373  
**Register Name:** IA32_SYSENTER_ESP

See Table 2-2.

**Register Address:** 176H, 374  
**Register Name:** IA32_SYSENTER_EIP

See Table 2-2.

**Register Address:** 179H, 377  
**Register Name:** IA32_MCG_CAP

See Table 2-2.

**Register Address:** 17AH, 378  
**Register Name:** IA32_MCG_STATUS

See Table 2-2.

**Register Address:** 17DH, 381  
**Register Name:** MSR_SMM_MCA_CAP

Enhanced SMM Capabilities (SMM-RO)  
Reports SMM capability Enhancement. Accessible only while in SMM.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| 31:0  | Bank Support (SMM-RO)  
One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA). |
| 55:32 | Reserved. |
| 56    | Targeted SMI (SMM-RO)  
Set if targeted SMI is supported. |
| 57    | SMM_CPU_SVRSTR (SMM-RO)  
Set if SMM SRAM save/restore feature is supported. |
| 58    | SMM_CODE_ACCESS_CHK (SMM-RO)  
Set if SMM code access check feature is supported. |
| 59    | Long_Flow_Indication (SMM-RO)  
If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler. |
| 63:60 | Reserved. |

**Register Address:** 186H, 390  
**Register Name:** IA32_PERFEVTSEL0

Performance Monitoring Event Select Register (R/W)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Event Select.</td>
</tr>
<tr>
<td>15:8</td>
<td>UMask.</td>
</tr>
<tr>
<td>16</td>
<td>USR.</td>
</tr>
<tr>
<td>17</td>
<td>OS.</td>
</tr>
<tr>
<td>18</td>
<td>Edge.</td>
</tr>
<tr>
<td>19</td>
<td>PC.</td>
</tr>
<tr>
<td>20</td>
<td>INT.</td>
</tr>
<tr>
<td>21</td>
<td>AnyThread.</td>
</tr>
<tr>
<td>22</td>
<td>EN.</td>
</tr>
<tr>
<td>23</td>
<td>INV.</td>
</tr>
</tbody>
</table>

---

Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17DH, 381</td>
<td>MSR_SMM_MCA_CAP</td>
</tr>
</tbody>
</table>
| Enhanced SMM Capabilities (SMM-RO)  
Reports SMM capability Enhancement. Accessible only while in SMM. | | |
| | | |
| 31:0 | Bank Support (SMM-RO)  
One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA). | |
| 55:32 | Reserved. | |
| 56 | Targeted SMI (SMM-RO)  
Set if targeted SMI is supported. | |
| 57 | SMM_CPU_SVRSTR (SMM-RO)  
Set if SMM SRAM save/restore feature is supported. | |
| 58 | SMM_CODE_ACCESS_CHK (SMM-RO)  
Set if SMM code access check feature is supported. | |
| 59 | Long_Flow_Indication (SMM-RO)  
If set to 1, indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler. | |
| 63:60 | Reserved. | |
| Register Address: 186H, 390 | IA32_PERFEVTSEL0 | Thread |
| Performance Monitoring Event Select Register (R/W)  
See Table 2-2. | | |
| 7:0 | Event Select. | |
| 15:8 | UMask. | |
| 16 | USR. | |
| 17 | OS. | |
| 18 | Edge. | |
| 19 | PC. | |
| 20 | INT. | |
| 21 | AnyThread. | |
| 22 | EN. | |
| 23 | INV. | |
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>31:24</td>
<td>CMASK.</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 187H, 391</td>
<td>IA32_PERFEVTSEL1</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 19AH, 410</td>
<td>IA32_CLOCK_MODULATION</td>
</tr>
<tr>
<td>Clock Modulation (R/W)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 19BH, 411</td>
<td>IA32_THERM_INTERRUPT</td>
</tr>
<tr>
<td>Thermal Interrupt Control (R/W)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Thermal Status (R/O)</td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status Log (R/wC0)</td>
</tr>
<tr>
<td>2</td>
<td>PROTCOT # or FORCEPR# Status (R/O)</td>
</tr>
<tr>
<td>3</td>
<td>PROTCOT # or FORCEPR# Log (R/wC0)</td>
</tr>
<tr>
<td>4</td>
<td>Critical Temperature Status (R/O)</td>
</tr>
<tr>
<td>5</td>
<td>Critical Temperature Status Log (R/wC0)</td>
</tr>
<tr>
<td>6</td>
<td>Thermal Threshold #1 Status (R/O)</td>
</tr>
<tr>
<td>7</td>
<td>Thermal Threshold #1 Log (R/wC0)</td>
</tr>
<tr>
<td>8</td>
<td>Thermal Threshold #2 Status (R/O)</td>
</tr>
<tr>
<td>9</td>
<td>Thermal Threshold #2 Log (R/wC0)</td>
</tr>
<tr>
<td>10</td>
<td>Power Limitation Status (R/O)</td>
</tr>
<tr>
<td>11</td>
<td>Power Limitation Log (R/wC0)</td>
</tr>
<tr>
<td>15:12</td>
<td>Reserved.</td>
</tr>
<tr>
<td>22:16</td>
<td>Digital Readout (R/O)</td>
</tr>
<tr>
<td>26:23</td>
<td>Reserved.</td>
</tr>
<tr>
<td>30:27</td>
<td>Resolution in Degrees Celsius (R/O)</td>
</tr>
<tr>
<td>31</td>
<td>Reading Valid (R/O)</td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
</tr>
</tbody>
</table>

2-438 Vol. 4
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature

**DisplayFamily_Dispayload Model Value of 06_57H or 06_85H (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A2H, 418</td>
<td>MSR_TEMPERATURE_TARGET</td>
<td>Temperature Target</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>15:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>Temperature Target (R)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29:24</td>
<td>Target Offset (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:30</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1A4H, 420</td>
<td>MSR_MISC_FEATURE_CONTROL</td>
<td>Miscellaneous Feature Control (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>DCU Hardware Prefetcher Disable (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>If 1, disables the L1 data cache prefetcher.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L2 Hardware Prefetcher Disable (R/W)</td>
<td></td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td></td>
<td>If 1, disables the L2 hardware prefetcher.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1A6H, 422</td>
<td>MSR_OFFCORE_RSP_0</td>
<td>Offcore Response Event Select Register (R/W)</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A7H, 423</td>
<td>MSR_OFFCORE_RSP_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offcore Response Event Select Register (R/W)</td>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1ADH, 429</td>
<td>MSR_TURBO_RATIO_LIMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Ratio Limit of Turbo Mode for Groups of Cores (R/W)</td>
<td>Package</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:1</td>
<td>Maximum Number of Cores in Group 0 Number active processor cores which operates under the maximum ratio limit for group 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td>Maximum Ratio Limit for Group 0 Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20:16</td>
<td>Number of Incremental Cores Added to Group 1 Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = “group 0 Max ratio limit” - “group ratio delta for group 1”.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:21</td>
<td>Group Ratio Delta for Group 1 An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28:24</td>
<td>Number of Incremental Cores Added to Group 2 Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = “group 1 Max ratio limit” - “group ratio delta for group 2”.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:29</td>
<td>Group Ratio Delta for Group 2 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36:32</td>
<td>Number of Incremental Cores Added to Group 3 Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = “group 2 Max ratio limit” - “group ratio delta for group 3”.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:37</td>
<td>Group Ratio Delta for Group 3 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44:40</td>
<td>Number of Incremental Cores Added to Group 4 Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = “group 3 Max ratio limit” - “group ratio delta for group 4”.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47:45</td>
<td>Group Ratio Delta for Group 4 An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52:48</td>
<td>Number of Incremental Cores Added to Group 5 Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = “group 4 Max ratio limit” - “group ratio delta for group 5”.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>55:53</td>
<td>Group Ratio Delta for Group 5</td>
<td>An unsigned integer specifying the ratio decrement relative to the Max ratio</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>limit for Group 4.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60:56</td>
<td>Number of Incremental Cores Added</td>
<td>Group 6, which includes the specified number of additional cores plus all</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>to Group 6</td>
<td>the cores in group 5, operates under the group 6 turbo max ratio limit =</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>“group 5 Max ratio limit” - “group ratio delta for group 6”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:61</td>
<td>Group Ratio Delta for Group 6</td>
<td>An unsigned integer specifying the ratio decrement relative to the Max ratio</td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>limit for Group 5.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1B0H, 432</td>
<td>IA32_ENERGY_PERF_BIAS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1B1H, 433</td>
<td>IA32_PACKAGE_THERM_STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1B2H, 434</td>
<td>IA32_PACKAGE_THERM_INTERRUPT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1C8H, 456</td>
<td>MSR_LBR_SELECT</td>
<td>Last Branch Record Filtering Select Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 18.9.2, “Filtering of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Last Branch Records.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CPL_EQ_0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CPL_NEQ_0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JCC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NEAR_REL_CALL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NEAR_IND_CALL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>NEAR_RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>NEAR_IND_IMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>NEAR_REL_IMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>FAR_BRANCH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1C9H, 457</td>
<td>MSR_LASTBRANCH_TOS</td>
<td>Last Branch Record Stack TOS (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contains an index (bits 0-2) that</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>points to the MSR containing the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>most recent branch record.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See MSR_LASTBRANCH_0_FROM_IP.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D9H, 473</td>
<td>IA32_DEBUGCTL</td>
<td>Debug Control (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LBR</td>
<td>Setting this bit to 1 enables the processor to record a running trace of the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>most recent branches taken by the processor in the LBR stack.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 BTF</td>
<td>Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 TR</td>
<td>Setting this bit to 1 enables branch trace messages to be sent.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 BTS</td>
<td>Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 BTINT</td>
<td>When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 BTS_OFF_OS</td>
<td>When set, BTS or BTM is skipped if CPL = 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 BTS_OFF_USR</td>
<td>When set, BTS or BTM is skipped if CPL &gt; 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 FREEZE_LBR_ON_PMI</td>
<td>When set, the LBR stack is frozen on a PMI request.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 FREEZE_PERFMON_ON_PMI</td>
<td>When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 FREEZE_WHILE_SMM</td>
<td>When set, freezes perfmon and trace messages while in SMM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:15 Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Address: 1DDH, 477 MSR_LER_FROM_LIP
Last Exception Record from Linear IP (R) Thread

Register Address: 1DEH, 478 MSR_LER_TO_LIP
Last Exception Record to Linear IP (R) Thread

Register Address: 1F2H, 498 IA32_SMRR_PHYSBASE
See Table 2-2. Core

Register Address: 1F3H, 499 IA32_SMRR_PHYSMASK
See Table 2-2. Core

Register Address: 200H, 512 IA32_MTRR_PHYSBASE0
See Table 2-2. Core

Register Address: 201H, 513 IA32_MTRR_PHYSMASK0
See Table 2-2. Core

Register Address: 202H, 514 IA32_MTRR_PHYSBASE1

Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)
Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 203H, 515</td>
<td>IA32_MTRR_PHYSMASK1</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 204H, 516</td>
<td>IA32_MTRR_PHYSBASE2</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 205H, 517</td>
<td>IA32_MTRR_PHYSMASK2</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 206H, 518</td>
<td>IA32_MTRR_PHYSBASE3</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 207H, 519</td>
<td>IA32_MTRR_PHYSMASK3</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 208H, 520</td>
<td>IA32_MTRR_PHYSBASE4</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 209H, 521</td>
<td>IA32_MTRR_PHYSMASK4</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20AH, 522</td>
<td>IA32_MTRR_PHYSBASE5</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20BH, 523</td>
<td>IA32_MTRR_PHYSMASK5</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20CH, 524</td>
<td>IA32_MTRR_PHYSBASE6</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20DH, 525</td>
<td>IA32_MTRR_PHYSMASK6</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20EH, 526</td>
<td>IA32_MTRR_PHYSBASE7</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20FH, 527</td>
<td>IA32_MTRR_PHYSMASK7</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 250H, 592</td>
<td>IA32_MTRR_FIX64K_00000</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 258H, 600</td>
<td>IA32_MTRR_FIX16K_80000</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 259H, 601</td>
<td>IA32_MTRR_FIX16K_A0000</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>IA32_MTRR_FIX4K_C0000</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>IA32_MTRR_FIX4K_C8000</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 26AH, 618</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Default Memory Types (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 309H, 777</td>
<td>IA32_FIXED_CTR0</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 0 (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30AH, 778</td>
<td>IA32_FIXED_CTR1</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 1 (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 30BH, 779</td>
<td>IA32_FIXED_CTR2</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Fixed-Function Performance Counter Register 2 (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 345H, 837</td>
<td>IA32_PERF_CAPABILITIES</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-2. See Section 18.4.1, &quot;IA32_DEBUGCTL MSR.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38DH, 909</td>
<td>IA32_FIXED_CTR_CTRL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Fixed-Function-Counter Control Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38EH, 910</td>
<td>IA32_PERF_GLOBAL_STATUS</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 38FH, 911</td>
<td>IA32_PERF_GLOBAL_CTRL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 390H, 912</td>
<td>IA32_PERF_GLOBAL_OVF_CTRL</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Scope</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBS_ENABLE (MSR_PEBS_ENABLE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F8H, 1016</td>
<td>MSR_PKG_C3_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C3 Residency Counter (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F9H, 1017</td>
<td>MSR_PKG_C6_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C6 Residency Counter (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FAH, 1018</td>
<td>MSR_PKG_C7_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Package C7 Residency Counter (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FCH, 1020</td>
<td>MSR_MC0_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Module C0 Residency Counter (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FDH, 1021</td>
<td>MSR_MC6_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Module C6 Residency Counter (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3FFH, 1023</td>
<td>MSR_CORE_C6_RESIDENCY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>CORE C6 Residency Counter (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MC1_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MC1_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, &quot;IA32_MCI_STATUS MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, &quot;IA32_MCI_ADDR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, &quot;IA32_MCI_CTL MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature

<table>
<thead>
<tr>
<th>DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: Hex, Decimal</td>
</tr>
<tr>
<td>Register Information / Bit Fields</td>
</tr>
<tr>
<td>Register Address: 40DH, 1037</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
</tr>
<tr>
<td>Register Address: 411H, 1041</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
</tr>
<tr>
<td>Register Address: 412H, 1042</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
</tr>
<tr>
<td>The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
</tr>
<tr>
<td>Register Address: 414H, 1044</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
</tr>
<tr>
<td>Register Address: 415H, 1045</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRs.”</td>
</tr>
<tr>
<td>Register Address: 416H, 1046</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
</tr>
<tr>
<td>Register Address: 4C1H, 1217</td>
</tr>
<tr>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 4C2H, 1218</td>
</tr>
<tr>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>Register Address: 600H, 1536</td>
</tr>
<tr>
<td>DS Save Area (R/W)</td>
</tr>
<tr>
<td>Register Address: 606H, 1542</td>
</tr>
<tr>
<td>Unit Multipliers Used in RAPL Interfaces (R/O)</td>
</tr>
<tr>
<td>3:0</td>
</tr>
<tr>
<td>See Section 15.10.1, “RAPL Interfaces.”</td>
</tr>
<tr>
<td>7:4</td>
</tr>
<tr>
<td>12:8</td>
</tr>
<tr>
<td>Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules).</td>
</tr>
<tr>
<td>15:13</td>
</tr>
<tr>
<td>19:16</td>
</tr>
<tr>
<td>See Section 15.10.1, “RAPL Interfaces.”</td>
</tr>
</tbody>
</table>
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature
DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:20 Reserved.</td>
<td>POOL_A1_RESIDENCY</td>
</tr>
<tr>
<td>Register Address: 600H, 1549</td>
<td>MSR_PKG_C2_RESIDENCY</td>
</tr>
<tr>
<td>Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states.</td>
<td>Package</td>
</tr>
<tr>
<td>63:0 Package C2 Residency Counter (R/O)</td>
<td>MSR_PKG_POWER_LIMIT</td>
</tr>
<tr>
<td>Register Address: 610H, 1552</td>
<td>PKG RAPL Power Limit Control (R/W)</td>
</tr>
<tr>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 611H, 1553</td>
<td>MSR_PKG_ENERGY_STATUS</td>
</tr>
<tr>
<td>PKG Energy Status (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 613H, 1555</td>
<td>MSR_PKG_PERF_STATUS</td>
</tr>
<tr>
<td>PKG Perf Status (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 614H, 1556</td>
<td>MSR_PKG_POWER_INFO</td>
</tr>
<tr>
<td>PKG RAPL Parameters (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.3, “Package RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 618H, 1560</td>
<td>MSR_DRAM_POWER_LIMIT</td>
</tr>
<tr>
<td>DRAM RAPL Power Limit Control (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 619H, 1561</td>
<td>MSR_DRAM_ENERGY_STATUS</td>
</tr>
<tr>
<td>DRAM Energy Status (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 61BH, 1563</td>
<td>MSR_DRAM_PERF_STATUS</td>
</tr>
<tr>
<td>DRAM Performance Throttling Status (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 61CH, 1564</td>
<td>MSR_DRAM_POWER_INFO</td>
</tr>
<tr>
<td>DRAM RAPL Parameters (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.5, “DRAM RAPL Domain.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 638H, 1592</td>
<td>MSR_PP0_POWER_LIMIT</td>
</tr>
<tr>
<td>PPO RAPL Power Limit Control (R/W)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 639H, 1593</td>
<td>MSR_PP0_ENERGY_STATUS</td>
</tr>
<tr>
<td>PPO Energy Status (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>See Section 15.10.4, “PP0/PP1 RAPL Domains.”</td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 648H, 1608</td>
<td>MSR_CONFIG_TDP_NOMINAL</td>
</tr>
<tr>
<td>Base TDP Ratio (R/O)</td>
<td>Package</td>
</tr>
<tr>
<td>See Table 2-25.</td>
<td>Package</td>
</tr>
</tbody>
</table>
### Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature

DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 649H, 1609</td>
<td>MSR_CONFIG_TDP_LEVEL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Level 1 ratio and power level (R/O)</td>
<td>See Table 2-25.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 64AH, 1610</td>
<td>MSR_CONFIG_TDP_LEVEL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Level 2 ratio and power level (R/O)</td>
<td>See Table 2-25.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 64BH, 1611</td>
<td>MSR_CONFIG_TDP_CONTROL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Control (R/w)</td>
<td>See Table 2-25.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 64CH, 1612</td>
<td>MSR_TURBO_ACTIVATION_RATIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConfigTDP Control (R/w)</td>
<td>See Table 2-25.</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>Register Address: 690H, 1680</td>
<td>MSR_CORE_PERF_LIMIT_REASONS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicator of Frequency Clipping in Processor Cores (R/w)</td>
<td>(Frequency refers to processor core frequency,)</td>
<td></td>
<td>Package</td>
</tr>
<tr>
<td>0</td>
<td>PROCHOT Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Thermal Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:2</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VR Therm Alert Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Electrical Design Point Status (R0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6E0H, 1760</td>
<td>IA32_TSC_DEADLINE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSC Target of Local APIC's TSC Deadline Mode (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>Register Address: 802H, 2050</td>
<td>IA32_X2APIC_APICID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC ID Register (R/O)</td>
<td></td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>Register Address: 803H, 2051</td>
<td>IA32_X2APIC_VERSION</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>x2APIC Version Register (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 808H, 2056</td>
<td>IA32_X2APIC_TPR</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>x2APIC Task Priority Register (R/w)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80AH, 2058</td>
<td>IA32_X2APIC_PPR</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>x2APIC Processor Priority Register (R/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80BH, 2059</td>
<td>IA32_X2APIC_EOI</td>
<td></td>
<td>Thread</td>
</tr>
<tr>
<td>x2APIC EOI Register (w/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 80DH, 2061</td>
<td>IA32_X2APIC_LDR</td>
<td></td>
<td>Thread</td>
</tr>
</tbody>
</table>
Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_X2APIC_SIVR</td>
<td>x2APIC Spurious Interrupt Vector Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>80FH, 2063</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR0</td>
<td>x2APIC In-Service Register Bits [31:0] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>810H, 2064</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR1</td>
<td>x2APIC In-Service Register Bits [63:32] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>811H, 2065</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR2</td>
<td>x2APIC In-Service Register Bits [95:64] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>812H, 2066</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR3</td>
<td>x2APIC In-Service Register Bits [127:96] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>813H, 2067</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR4</td>
<td>x2APIC In-Service Register Bits [159:128] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>814H, 2068</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR5</td>
<td>x2APIC In-Service Register Bits [191:160] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>815H, 2069</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR6</td>
<td>x2APIC In-Service Register Bits [223:192] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>816H, 2070</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_ISR7</td>
<td>x2APIC In-Service Register Bits [255:224] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>817H, 2071</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR0</td>
<td>x2APIC Trigger Mode Register Bits [31:0] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>818H, 2072</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR1</td>
<td>x2APIC Trigger Mode Register Bits [63:32] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>819H, 2073</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR2</td>
<td>x2APIC Trigger Mode Register Bits [95:64] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>81AH, 2074</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR3</td>
<td>x2APIC Trigger Mode Register Bits [127:96] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>81BH, 2075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR4</td>
<td>x2APIC Trigger Mode Register Bits [159:128] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>81CH, 2076</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR5</td>
<td>x2APIC Trigger Mode Register Bits [191:160] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>81DH, 2077</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR6</td>
<td>x2APIC Trigger Mode Register Bits [223:192] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>81EH, 2078</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_TMR7</td>
<td>x2APIC Trigger Mode Register Bits [255:224] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>81FH, 2079</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_IRR0</td>
<td>x2APIC Interrupt Request Register Bits [31:0] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>820H, 2080</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA32_X2APIC_IRR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>821H, 2081</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-56. Selected MSRs Supported by Intel® Xeon Phi™ Processors with a CPUID Signature DisplayFamily_DisplayModel Value of 06_57H or 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32.2APIC.IRR2</td>
<td>x2APIC Interrupt Request Register Bits [63:32] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.IRR3</td>
<td>x2APIC Interrupt Request Register Bits [95:64] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.IRR4</td>
<td>x2APIC Interrupt Request Register Bits [127:96] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.IRR5</td>
<td>x2APIC Interrupt Request Register Bits [159:128] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.IRR6</td>
<td>x2APIC Interrupt Request Register Bits [191:160] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.IRR7</td>
<td>x2APIC Interrupt Request Register Bits [223:192] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.IRR8</td>
<td>x2APIC Interrupt Request Register Bits [255:224] (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.ESR</td>
<td>x2APIC Error Status Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.CMCI</td>
<td>x2APIC LVT Corrected Machine Check Interrupt Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.ICR</td>
<td>x2APIC Interrupt Command Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.TIMER</td>
<td>x2APIC LVT Timer Interrupt Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.THERMAL</td>
<td>x2APIC LVT Thermal Sensor Interrupt Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.PMI</td>
<td>x2APIC LVT Performance Monitor Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.LINT0</td>
<td>x2APIC LVT LINT0 Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.LINT1</td>
<td>x2APIC LVT LINT1 Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.LVT.ERROR</td>
<td>x2APIC LVT Error Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.INIT.COUNT</td>
<td>x2APIC Initial Count Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.CUR.COUNT</td>
<td>x2APIC Current Count Register (R/O)</td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>IA32.2APIC.DIV_CONF</td>
<td>x2APIC Divide Configuration Register (R/W)</td>
<td>Thread</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-57 lists model-specific registers that are supported by the Intel® Xeon Phi™ processor 7215, 7285, 7295 series based on the Knights Mill microarchitecture.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: B3FH, 2111</td>
<td>IA32_X2APIC_SELF_IPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2APIC Self IPI Register (W/O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_EFER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Feature Enables</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_STAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Call Target Address (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_LSTAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA-32e Mode System Call Target Address (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FMASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Call Flag Mask (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_FS_BASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map of BASE Address of FS (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0101H</td>
<td>IA32_GS_BASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map of BASE Address of GS (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0102H</td>
<td>IA32_KERNEL_GS_BASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap Target of BASE Address of GS (R/w)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0103H</td>
<td>IA32_TSC_AUX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUXILIARY TSC Signature (R/W)</td>
<td>See Table 2-2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-57 lists model-specific registers that are supported by the Intel® Xeon Phi™ processor 7215, 7285, 7295 series based on the Knights Mill microarchitecture.

Table 2-57. Additional MSRs Supported by the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_85H

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 9BH, 155</td>
<td>IA32_SMM_MONITOR_CTL</td>
<td></td>
<td>Core</td>
</tr>
<tr>
<td>SMM Monitor Configuration (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This MSR is readable only if VMX is enabled, and writeable only if VMX is enabled and in SMM mode, and is used to configure the VMX MSEG base address. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 480H, 1152</td>
<td>IA32_VMX_BASIC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-57. Additional MSRs Supported by the Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series with a CPUID Signature DisplayFamily_DisplayModel Value of 06_85H (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporting Register of Basic VMX Capabilities (R/O)</td>
<td>See Table 2-2.</td>
</tr>
<tr>
<td>IA32_VMX.PinBased_CTLS</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.ProcBased_CTLS</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.Exit_CTLS</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of VM-exit Controls (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.Entry_CTLS</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of VM-entry Controls (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.Misc</td>
<td></td>
</tr>
<tr>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.CRO.Fixed.0</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.CRO.Fixed.1</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.CR4.Fixed.0</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.CR4.Fixed.1</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.VMCS.Enumerable</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.ProcBased_CTLS2</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.EPT.VPID.Enumerable</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of EPT and VPID (R/O) See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>IA32_VMX.True.PinBased_CTLS</td>
<td></td>
</tr>
<tr>
<td>Capability Reporting Register of True Pin-based VM-execution Controls (R/O) See Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
</tr>
<tr>
<td>Register Address: 0H, 0</td>
<td>IA32_P5_MC_ADDR</td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>IA32_P5_MC_TYPE</td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 6H, 6</td>
<td>IA32_MONITOR_FILTER_LINE_SIZE</td>
</tr>
<tr>
<td>See Section 9.10.5, “Monitor/Mwait Address Range Determination.”</td>
<td></td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
</tr>
<tr>
<td>Time Stamp Counter</td>
<td>See Table 2-2.</td>
</tr>
</tbody>
</table>

2.19 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-58 lists MSRs (architectural and model-specific) that are defined across processor generations based on Intel NetBurst microarchitecture. The processor can be identified by its CPUID signatures of DisplayFamily encoding of 0FH, see Table 2-1.

- MSRs with an "IA32_" prefix are designated as "architectural." This means that the functions of these MSRs and their addresses remain the same for succeeding families of IA-32 processors.
- MSRs with an "MSR_" prefix are model specific with respect to address functionalities. The column "Model Availability" lists the model encoding value(s) within the Pentium 4 and Intel Xeon processor family at the specified register address. The model encoding value of a processor can be queried using CPUID. See "CPUID—CPU Identification" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.
**Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique&lt;sup&gt;1&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>On earlier processors, only the lower 32 bits are writable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>IA32_PLATFORM_ID</td>
<td>Platform ID (R)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2. The operating system can use this MSR to determine &quot;slot&quot; information for the processor and the proper microcode update to load.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1BH, 27</td>
<td>IA32_APIC_BASE</td>
<td>APIC Location and Status (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table 2-2. See Section 11.4.4, &quot;Local APIC Status and Location.&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2AH, 42</td>
<td>MSR_EBC_HARD_POWERON</td>
<td>Processor Hard Power-On Configuration (R/W) Enables and disables processor features. (R) Indicates current processor configuration.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>0</td>
<td>Output Tri-state Enabled (R)</td>
<td>Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Execute BIST (R)</td>
<td>Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>In Order Queue Depth (R)</td>
<td>Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MCERR# Observation Disabled (R)</td>
<td>Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BINIT# Observation Enabled (R)</td>
<td>Indicates whether BINIT# observation is enabled (0) or disabled (1) as determined by the strapping of A10#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:5</td>
<td>APIC Cluster ID (R)</td>
<td>Contains the logical APIC cluster ID value as set by the strapping of A12# and A11#. The logical cluster ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Bus Park Disable (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indicates whether bus park is enabled (0) or disabled (1) as set by the strapping of A15#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:12</td>
<td>Agent ID (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:14</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register Address: 2BH, 43**  
**MSR_EBC_SOFT_POWERON**

**Processor Soft Power-On Configuration (R/W)**
Enables and disables processor features.

<table>
<thead>
<tr>
<th>0</th>
<th>RCNT/SCNT On Request Encoding Enable (R/W)</th>
<th>0, 1, 2, 3, 4, 6</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data Error Checking Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set to disable system data bus parity checking; clear to enable parity checking.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Response Error Checking Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set to disable (default); clear to enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Address/Request Error Checking Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set to disable (default); clear to enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Initiator MCERR# Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set to disable MCERR# driving for initiator bus requests (default); clear to enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Internal MCERR# Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set to disable MCERR# driving for initiator internal errors (default); clear to enable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BINIT# Driver Disable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set to disable BINIT# driver (default); clear to enable driver.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:7</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register Address: 2CH, 44**  
**MSR_EBC_FREQUENCY_ID**

**Processor Frequency Configuration**
The bit field layout of this MSR varies according to the MODEL value in the CPUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2.

(R) The field Indicates the current processor frequency configuration.

<table>
<thead>
<tr>
<th>15:0</th>
<th>Reserved.</th>
<th>2, 3, 4, 6</th>
<th>Shared</th>
</tr>
</thead>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>18:16</td>
<td>MSR_EBC_FREQUENCY_ID</td>
<td>Scalable Bus Speed (R/w)</td>
<td>Indicates the intended scalable bus speed: Encoding Scalable Bus Speed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000B 100 MHz (Model 2)</td>
<td></td>
<td>0, 1</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000B 266 MHz (Model 3 or 4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>001B 133 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>010B 200 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>011B 166 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100B 333 MHz (Model 6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:19</td>
<td>Reserved.</td>
<td>Core Clock Frequency to System Bus Frequency Ratio (R)</td>
<td>The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>Reserved.</td>
<td>MSR_EBC_FREQUENCY_ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:25</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2CH, 44</td>
<td></td>
<td>Processor Frequency Configuration (R)</td>
<td>The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.</td>
<td>0, 1</td>
<td>Shared</td>
</tr>
<tr>
<td>20:0</td>
<td>Reserved.</td>
<td>Scalable Bus Speed (R/w)</td>
<td>Indicates the intended scalable bus speed: Encoding Scalable Bus Speed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23:21</td>
<td>Reserved.</td>
<td>000B 100 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reserved.</td>
<td>All others values reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AH, 58</td>
<td></td>
<td>Control Features in IA-32 Processor (R/w)</td>
<td>See Table 2-2. (If CPUID.01H:ECX.[bit 5])</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>79H, 121</td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOS Update Trigger Register (W)</td>
<td>See Table 2-2.</td>
<td>IA32_BIOS_SIGN_ID</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 8BH, 139</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIOS Update Signature ID (R/W)</td>
<td>See Table 2-2.</td>
<td>IA32_SMM_MONITOR_CTL</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 9BH, 155</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMM Monitor Configuration (R/w)</td>
<td>See Table 2-2.</td>
<td>IA32_MTRRCAP</td>
<td>3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTRR Information</td>
<td>See Section 12.11.1, “MTRR Feature Identification.”</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS Register Target for CPL 0 Code (R/W)</td>
<td>See Table 2-2 and Section 5.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”</td>
<td>IA32_SYSENTER_ESP</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack Pointer for CPL 0 Stack (R/W)</td>
<td>See Table 2-2 and Section 5.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”</td>
<td>IA32_SYSENTER_EIP</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPL 0 Code Entry Point (R/W)</td>
<td>See Table 2-2 and Section 5.8.7, “Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions.”</td>
<td>IA32_MCG_CAP</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Capabilities (R)</td>
<td>See Table 2-2 and Section 16.3.1.1, “IA32_MCG_CAP MSR.”</td>
<td>IA32_MCG_STATUS</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Status (R)</td>
<td>See Table 2-2 and Section 16.3.1.2, “IA32_MCG_STATUS MSR.”</td>
<td>IA32_MCG_CTL</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 17BH, 379</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Feature Enable (R/W)</td>
<td>See Table 2-2 and Section 16.3.1.3, “IA32_MCG_CTL MSR.”</td>
<td>MSR_MCG_RAX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 180H, 384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check EAX/RAX Save State</td>
<td>See Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”</td>
<td>MSR_MCG_RBX</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 181H, 385</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

63:0 Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
### Model-Specific Registers (MSRs)

**Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Check EBX/RBX Save State</td>
<td>MSR_MCG_RCX</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 182H, 386</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check ECX/RCX Save State</td>
<td>MSR_MCG_RDX</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 183H, 387</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check EDX/RDX Save State</td>
<td>MSR_MCG_RSI</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 184H, 388</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check ESI/RSI Save State</td>
<td>MSR_MCG_RDI</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 185H, 389</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check EDI/RDI Save State</td>
<td>MSR_MCG_RBP</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 186H, 390</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check EBP/RBP Save State</td>
<td>MSR_MCG_RSP</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 187H, 391</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check ESP/RSP Save State</td>
<td>MSR_MCG_RFLAGS</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 188H, 392</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check EFLAGS/RFLAG Save State</td>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Machine Check EIP/RIP Save State</td>
<td>Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Machine Check Miscellaneous</td>
<td>When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Machine Check R8</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Machine Check R9D/R9</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Machine Check R10</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
</tbody>
</table>
Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 193H, 403</td>
<td>MSR_MCG_R11</td>
<td></td>
</tr>
<tr>
<td>Machine Check R11</td>
<td>See Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 194H, 404</td>
<td>MSR_MCG_R12</td>
<td></td>
</tr>
<tr>
<td>Machine Check R12</td>
<td>See Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 195H, 405</td>
<td>MSR_MCG_R13</td>
<td></td>
</tr>
<tr>
<td>Machine Check R13</td>
<td>See Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 196H, 406</td>
<td>MSR_MCG_R14</td>
<td></td>
</tr>
<tr>
<td>Machine Check R14</td>
<td>See Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 197H, 407</td>
<td>MSR_MCG_R15</td>
<td></td>
</tr>
<tr>
<td>Machine Check R15</td>
<td>See Section 16.3.2.6, “IA32_MCG Extended Machine Check State MSRs.”</td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 15.1, “Enhanced Intel Speedstep® Technology.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2. See Section 15.1, “Enhanced Intel Speedstep® Technology.”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_CLOCK_MODULATION</td>
<td>19AH, 410</td>
<td>Thermal Monitor Control (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_THERM_INTERRUPT</td>
<td>19BH, 411</td>
<td>Thermal Interrupt Control (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>IA32_THERM_STATUS</td>
<td>19CH, 412</td>
<td>Thermal Monitor Status (R/w)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>MSR_THERM2_CTL</td>
<td>19DH, 413</td>
<td>Thermal Monitor 2 Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.</td>
<td>3</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>4, 6</td>
<td>For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.</td>
<td>4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>IA32_MISC_ENABLE</td>
<td>1A0H, 416</td>
<td>Enable Miscellaneous Processor Features (R/w)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>0</td>
<td>Fast-Strings Enable. See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x87 FPU Fopcode Compatibility Mode Enable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Thermal Monitor 1 Enable</td>
<td>See Section 15.8.2, “Thermal Monitor,” and Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Split-Lock Disable</td>
<td>When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the bit is clear (default), normal split-locks are issued to the bus.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This debug feature is specific to the Pentium 4 processor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Third-Level Cache Disable (R/W)</td>
<td>When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CR0, the page-level cache controls, and/or the MTRRs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Section 12.5.4, “Disabling and Enabling the L3 Cache.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Suppress Lock Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Prefetch Queue Disable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, disables the prefetch queue. When clear (default), enables the prefetch queue.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>FERR# Interrupt Reporting Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SM#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (BTS_UNAVILABLE) (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>PEBS_UNAVILABLE: Processor Event Based Sampling Unavailable (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When set, the processor does not support processor event-based sampling (PEBS); when clear, PEBS is supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>TM2 Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:14</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ENABLE MONITOR FSM (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ Shared/Unique: 1: Shared, 2: Unique.
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Adjacent Cache Line Prefetch Disable (R/W)</td>
<td>When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector. Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Limit CPUID MAXVAL (R/W)</td>
<td>See Table 2-2. Setting this can cause unexpected behavior to software that depends on the availability of CPUID leaves greater than 3.</td>
<td>3, 4, 6</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>xTPR Message Disable (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>24</td>
<td>L1 Data Cache Context Mode (R/W)</td>
<td>When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 12.5.6, “L1 Data Cache Context Mode.” When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33:25</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>XD Bit Disable (R/W)</td>
<td>See Table 2-3.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>63:35</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 1A1H, 417</td>
<td><strong>MSR_PLATFORM_BRV</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Platform Feature Requirements (R)</strong></td>
<td><strong>3, 4, 6</strong></td>
<td><strong>Shared</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>PLATFORM Requirements</td>
<td>When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:19</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 1D7H, 471</td>
<td><strong>MSR_LER_FROM_LIP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address</td>
<td>Bit Description</td>
<td>Model Availability</td>
<td>Shared/Unique</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------</td>
<td>--------------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td><strong>Last Exception Record From Linear IP (R)</strong></td>
<td>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 18.13.3, &quot;Last Exception Records.&quot;</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>From Linear IP Linear address of the last branch instruction.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D7H, 471</td>
<td>MSR_LER_FROM_LIP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>From Linear IP Linear address of the last branch instruction (If IA-32e mode is active).</td>
<td></td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D8H, 472</td>
<td>MSR_LER_TO_LIP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Exception Record To Linear IP (R)</strong></td>
<td>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 18.13.3, &quot;Last Exception Records.&quot;</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>31:0</td>
<td>From Linear IP Linear address of the target of the last branch instruction.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D8H, 472</td>
<td>MSR_LER_TO_LIP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:0</td>
<td>From Linear IP Linear address of the target of the last branch instruction (If IA-32e mode is active).</td>
<td></td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1D9H, 473</td>
<td>MSR_DEBUGCTLA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Debug Control (R/W)</strong></td>
<td>Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 18.13.1, &quot;MSR_DEBUGCTLA MSR.&quot;</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1DAH, 474</td>
<td>MSR_LASTBRANCH_TOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record Stack TOS (R/O)</strong></td>
<td>Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record). See Section 18.13.2, &quot;LBR Stack for Processors Based on Intel NetBurst® Microarchitecture,&quot; and addresses 1DBH-1DEH and 680H-68FH.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 1DBH, 475</td>
<td>MSR_LASTBRANCH_0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 0 (R/O)</strong></td>
<td>One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6COH-6CFH. See Section 18.12, &quot;Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture.&quot;</td>
<td>0, 1, 2</td>
<td>Unique</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Register Name</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>1DCH, 476</td>
<td>MSR_LASTBRANCH_1</td>
<td>0, 1, 2</td>
<td>Unique</td>
</tr>
<tr>
<td>1DCH, 476</td>
<td>Last Branch Record 1</td>
<td>0, 1, 2</td>
<td>Unique</td>
</tr>
<tr>
<td>1DCH, 477</td>
<td>MSR_LASTBRANCH_2</td>
<td>0, 1, 2</td>
<td>Unique</td>
</tr>
<tr>
<td>1DCH, 478</td>
<td>MSR_LASTBRANCH_3</td>
<td>0, 1, 2</td>
<td>Unique</td>
</tr>
<tr>
<td>1DCH, 479</td>
<td>IA32_MTRR_PHYSBASE0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 480</td>
<td>IA32_MTRR_PHYSMASK0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 481</td>
<td>IA32_MTRR_PHYSBASE1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 482</td>
<td>IA32_MTRR_PHYSMASK1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 483</td>
<td>IA32_MTRR_PHYSBASE2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 484</td>
<td>IA32_MTRR_PHYSMASK2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 485</td>
<td>IA32_MTRR_PHYSBASE3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 486</td>
<td>IA32_MTRR_PHYSMASK3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 487</td>
<td>IA32_MTRR_PHYSBASE4</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>1DCH, 488</td>
<td>IA32_MTRR_PHYSMASK4</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Variable Range Mask MTRR</td>
<td>IA32_MTRR_PHYSBASE5</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.3, “Variable Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variable Range Mask MTRR</td>
<td>IA32_MTRR_PHYSMASK5</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.3, “Variable Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variable Range Mask MTRR</td>
<td>IA32_MTRR_PHYSBASE6</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.3, “Variable Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variable Range Mask MTRR</td>
<td>IA32_MTRR_PHYSMASK6</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.3, “Variable Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variable Range Mask MTRR</td>
<td>IA32_MTRR_PHYSBASE7</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.3, “Variable Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variable Range Mask MTRR</td>
<td>IA32_MTRR_PHYSMASK7</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.3, “Variable Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Range MTRR</td>
<td>IA32_MTRR_FIX64K_00000</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Range MTRR</td>
<td>IA32_MTRR_FIX16K_80000</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Range MTRR</td>
<td>IA32_MTRR_FIX16K_A0000</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Range MTRR</td>
<td>IA32_MTRR_FIX4K_C0000</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Range MTRR</td>
<td>IA32_MTRR_FIX4K_C8000</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Range MTRR</td>
<td>IA32_MTRR_FIX4K_D0000</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td>Model Availability</td>
<td>Shared/Unique</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------</td>
<td>----------------</td>
<td>--------------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>IA32_MTRR_FIX4K_D8000</td>
<td>Fixed Range MTRR</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>IA32_MTRR_FIX4K_E0000</td>
<td>Fixed Range MTRR</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>IA32_MTRR_FIX4K_E8000</td>
<td>Fixed Range MTRR</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>IA32_MTRR_FIX4K_F0000</td>
<td>Fixed Range MTRR</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>IA32_MTRR_FIX4K_F8000</td>
<td>Fixed Range MTRR</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 277H, 631</td>
<td>IA32_PAT</td>
<td>Page Attribute Table</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 12.11.2.2, “Fixed Range MTRRs.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td>Default Memory Types (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Table 2-2 and Section 12.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 300H, 768</td>
<td>MSR_BPU_COUNTER0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 301H, 769</td>
<td>MSR_BPU_COUNTER1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 302H, 770</td>
<td>MSR_BPU_COUNTER2</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 303H, 771</td>
<td>MSR_BPU_COUNTER3</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 304H, 772</td>
<td>MSR_MS_COUNTER0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 305H, 773</td>
<td>MSR_MS_COUNTER1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 306H, 774</td>
<td>MSR_MS_COUNTER2</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 307H, 775</td>
<td>MSR_MS_COUNTER3</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 308H, 776</td>
<td>MSR_FLAME_COUNTER0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.3.2, “Performance Counters.”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address</td>
<td>Register Name</td>
<td>Model Availability</td>
<td>Shared/Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>---------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>309H, 777</td>
<td>MSR_FLAME_COUNTER1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30AH, 778</td>
<td>MSR_FLAME_COUNTER2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30BH, 779</td>
<td>MSR_FLAME_COUNTER3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30CH, 780</td>
<td>MSR_IQ_COUNTER0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30DH, 781</td>
<td>MSR_IQ_COUNTER1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30EH, 782</td>
<td>MSR_IQ_COUNTER2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30FH, 783</td>
<td>MSR_IQ_COUNTER3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>310H, 784</td>
<td>MSR_IQ_COUNTER4</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>311H, 785</td>
<td>MSR_IQ_COUNTER5</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>360H, 864</td>
<td>MSR_BPU_CCCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>361H, 865</td>
<td>MSR_BPU_CCCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>362H, 866</td>
<td>MSR_BPU_CCCR2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>363H, 867</td>
<td>MSR_BPU_CCCR3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>364H, 868</td>
<td>MSR_MS_CCCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>365H, 869</td>
<td>MSR_MS_CCCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>366H, 870</td>
<td>MSR_MS_CCCR2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>367H, 871</td>
<td>MSR_MS_CCCR3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>368H, 872</td>
<td>MSR_FLAME_CCCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Section 20.6.3.2, "Performance Counters."
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 369H, 873</td>
<td>MSR_FLAME_CCCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 36AH, 874</td>
<td>MSR_FLAME_CCCR2</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 36BH, 875</td>
<td>MSR_FLAME_CCCR3</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 36CH, 876</td>
<td>MSR_IQ_CCCR0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 36DH, 877</td>
<td>MSR_IQ_CCCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 36EH, 878</td>
<td>MSR_IQ_CCCR2</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 36FH, 879</td>
<td>MSR_IQ_CCCR3</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 370H, 880</td>
<td>MSR_IQ_CCCR4</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 371H, 881</td>
<td>MSR_IQ_CCCR5</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.3, &quot;CCCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A0H, 928</td>
<td>MSR_BSU_ESCR0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A1H, 929</td>
<td>MSR_BSU_ESCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A2H, 930</td>
<td>MSR_FSB_ESCR0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A3H, 931</td>
<td>MSR_FSB_ESCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A4H, 932</td>
<td>MSR_FIRM_ESCR0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A5H, 933</td>
<td>MSR_FIRM_ESCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A6H, 934</td>
<td>MSR_FLAME_ESCR0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A7H, 935</td>
<td>MSR_FLAME_ESCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A8H, 936</td>
<td>MSR_DAC_ESCR0</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, &quot;ESCR MSRs.&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3A9H, 937</td>
<td>MSR_DAC_ESCR1</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
</tbody>
</table>
### MODEL-SPECIFIC REGISTERS (MSRS)

**Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_MOB_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AAH, 938</td>
<td>MSR_MOB_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_PMH_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3ABH, 939</td>
<td>MSR_PMH_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_SAAT_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3ACH, 940</td>
<td>MSR_SAAT_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_U2L_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3ADH, 941</td>
<td>MSR_U2L_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_BPU_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AEH, 942</td>
<td>MSR_BPU_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IS_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AFH, 943</td>
<td>MSR_IS_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_ITLB_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B0H, 944</td>
<td>MSR_ITLB_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_CRU_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B1H, 945</td>
<td>MSR_CRU_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR0</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B2H, 946</td>
<td>MSR_IQ_ESCR1</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR2</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B3H, 947</td>
<td>MSR_IQ_ESCR3</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR4</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B4H, 948</td>
<td>MSR_IQ_ESCR5</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR6</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B5H, 949</td>
<td>MSR_IQ_ESCR7</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR8</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B6H, 950</td>
<td>MSR_IQ_ESCR9</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR10</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B7H, 951</td>
<td>MSR_IQ_ESCR11</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR12</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B8H, 952</td>
<td>MSR_IQ_ESCR13</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR14</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3B9H, 953</td>
<td>MSR_IQ_ESCR15</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_IQ_ESCR16</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3BAH, 954</td>
<td>MSR_IQ_ESCR17</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
</tr>
</tbody>
</table>

*This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.*

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>MSR_MOB_ESCR0</td>
<td>0, 1, 2</td>
<td>Shared</td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AAH, 938</td>
<td>MSR_MOB_ESCR1</td>
<td>0, 1, 2</td>
<td>Shared</td>
<td></td>
</tr>
</tbody>
</table>

*This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.*
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>3BBH, 955</td>
<td>MSR_IQ_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.” This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.</td>
<td>0, 1, 2</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3BCH, 956</td>
<td>MSR_RAT_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3BDH, 957</td>
<td>MSR_RAT_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3BEH, 958</td>
<td>MSR_SSU_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3COH, 960</td>
<td>MSR_MS_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C1H, 961</td>
<td>MSR_MS_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C2H, 962</td>
<td>MSR_TBPU_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C3H, 963</td>
<td>MSR_TBPU_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C4H, 964</td>
<td>MSR_TC_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C5H, 965</td>
<td>MSR_TC_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C8H, 968</td>
<td>MSR_IX_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C9H, 969</td>
<td>MSR_IX_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3CAH, 970</td>
<td>MSR_ALF_ESCR0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3CBH, 971</td>
<td>MSR_ALF_ESCR1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3CCH, 972</td>
<td>MSR_CRU_ESCR2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3CDH, 973</td>
<td>MSR_CRU_ESCR3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3E0H, 992</td>
<td>MSR_CRU_ESCR4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3E1H, 993</td>
<td>MSR_CRU_ESCR5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ Model availability indicates whether the register is shared (Shared) or unique (Unique).
Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors  (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 3F0H, 1008</td>
<td>MSR_TC_PRECISE_EVENT</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.3.1, “ESCR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3F1H, 1009</td>
<td>IA32_PEBs_ENABLE (MSR_PEBs_ENABLE)</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Processor Event Based Sampling (PEBS) (R/W)</td>
<td>12:0</td>
<td>See <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a>.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td></td>
</tr>
<tr>
<td>Controls the enabling of processor event sampling and replay tagging.</td>
<td>23:13</td>
<td>Reserved.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 20.6.4.3, “IA32_PEBs_ENABLE MSR,” for an explanation of the target logical processor.</td>
<td>24</td>
<td>UOP Tag</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Enables replay tagging when set.</td>
<td>25</td>
<td>ENABLE_PEBs_MY_THR (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Enables PEBS for the target logical processor when set; disables PEBS when clear (default).</td>
<td>26</td>
<td>ENABLE_PEBs_OTH_THR (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Enables PEBS for the target logical processor when set; disables PEBS when clear (default).</td>
<td>63:27</td>
<td>Reserved.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See <a href="https://perfmon-events.intel.com/">https://perfmon-events.intel.com/</a>.</td>
<td>3F2H, 1010</td>
<td>MSR_PEBs_MATRIX_VERT</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCI_STATUS MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCI_ADDR MSRs.”</td>
<td>0, 1, 2, 3, 4, 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear.</td>
<td></td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 403H, 1027</td>
<td>IA32_MC0_MISC</td>
<td></td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
</tbody>
</table>

¹ This column indicates whether the MSR is available in all models or if it is unique to certain models.
Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Model Availability</th>
<th>Shared/Unique 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>The IA32_MC0_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MC1_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td>The IA32_MC1_CTL register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MC1_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs.”</td>
<td>The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 406H, 1030</td>
<td>IA32_MC1_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 407H, 1031</td>
<td>IA32_MC1_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>The IA32_MC2_MISC register is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td>The IA32_MC3_CTL register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs.”</td>
<td>The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MC_ADDR MSRs.”</td>
<td>The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 40BH, 1035</td>
<td>IA32_MC2_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MC_MISC MSRs.”</td>
<td>The IA32_MC3_MISC register is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>IA32_MC3_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MC_CTL MSRs.”</td>
<td>The IA32_MC3_STATUS register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 40DH, 1037</td>
<td>IA32_MC3_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MC_STATUS MSRs.”</td>
<td>The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>IA32_MC3_ADDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)**

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register Information / Bit Fields</strong></td>
<td><strong>Bit Description</strong></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCI_ADDR MSRs.”</td>
<td>The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 40FH, 1039</td>
<td><strong>IA32_MC3_MISC</strong></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 410H, 1040</td>
<td><strong>IA32_MC4_CTL</strong></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCI_CTL MSRs.”</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 411H, 1041</td>
<td><strong>IA32_MC4_STATUS</strong></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCI_STATUS MSRs.”</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address:</strong> 412H, 1042</td>
<td><strong>IA32_MC4_ADDR</strong></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCI_ADDR MSRs.”</td>
<td>The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 413H, 1043</td>
<td><strong>IA32_MC4_MISC</strong></td>
</tr>
<tr>
<td>See Section 16.3.2.4, “IA32_MCI_MISC MSRs.”</td>
<td>The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 480H, 1152</td>
<td><strong>IA32_VMX_BASIC</strong></td>
</tr>
<tr>
<td>Reporting Register of Basic VMX Capabilities (R/O)</td>
<td>See Table 2-2 and Appendix A.1, “Basic VMX Information.”</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 481H, 1153</td>
<td><strong>IA32_VMX_PINBASED_CTLS</strong></td>
</tr>
<tr>
<td>Capability Reporting Register of Pin-Based VM-Execution Controls (R/O)</td>
<td>See Table 2-2 and Appendix A.3, “VM-Execution Controls.”</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 482H, 1154</td>
<td><strong>IA32_VMX_PROCBASED_CTLS</strong></td>
</tr>
<tr>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)</td>
<td>See Appendix A.3, “VM-Execution Controls,” and Table 2-2.</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 483H, 1155</td>
<td><strong>IA32_VMX_EXIT_CTLS</strong></td>
</tr>
<tr>
<td>Capability Reporting Register of VM-Exit Controls (R/O)</td>
<td>See Appendix A.4, “VM-Exit Controls,” and Table 2-2.</td>
</tr>
<tr>
<td><strong>Register Address:</strong> 484H, 1156</td>
<td><strong>IA32_VMX_ENTRY_CTLS</strong></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Information / Bit Fields</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>485H, 1157</td>
<td>Capability Reporting Register of VM-Entry Controls (R/O)</td>
</tr>
<tr>
<td>486H, 1158</td>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
</tr>
<tr>
<td>487H, 1159</td>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
</tr>
<tr>
<td>488H, 1160</td>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
</tr>
<tr>
<td>489H, 1161</td>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
</tr>
<tr>
<td>48AH, 1162</td>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O)</td>
</tr>
<tr>
<td>680H-68FH</td>
<td>DS Save Area (R/W)</td>
</tr>
<tr>
<td>681H, 1664</td>
<td>MSR_LASTBRANCH_0_FROM_IP</td>
</tr>
<tr>
<td>682H, 1666</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
</tr>
</tbody>
</table>

### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>485H, 1157</td>
<td>Capability Reporting Register of VM-Entry Controls (R/O)</td>
<td>See Appendix A.5, &quot;VM-Entry Controls,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>486H, 1158</td>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
<td>See Appendix A.6, &quot;Miscellaneous Data,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>487H, 1159</td>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
<td>See Appendix A.7, &quot;VMX-Fixed Bits in CR0,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>488H, 1160</td>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
<td>See Appendix A.7, &quot;VMX-Fixed Bits in CR0,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>489H, 1161</td>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
<td>See Appendix A.8, &quot;VMX-Fixed Bits in CR4,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>48AH, 1162</td>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O)</td>
<td>See Appendix A.9, &quot;VMCS Enumeration,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>48BH, 1163</td>
<td>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)</td>
<td>See Appendix A.3, &quot;VM-Execution Controls,&quot; and Table 2-2.</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>600H, 1536</td>
<td>IA32_DS_AREA</td>
<td>DS Save Area (R/W)</td>
<td>0, 1, 2, 3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>681H, 1665</td>
<td>MSR_LASTBRANCH_1_FROM_IP</td>
<td>Last Branch Record 1</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>682H, 1666</td>
<td>MSR_LASTBRANCH_2_FROM_IP</td>
<td>Last Branch Record 1</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 2</td>
<td>MSR_LASTBRANCH_3_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 683H, 1667</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 3</td>
<td>MSR_LASTBRANCH_4_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 684H, 1668</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 4</td>
<td>MSR_LASTBRANCH_5_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 685H, 1669</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 5</td>
<td>MSR_LASTBRANCH_6_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 686H, 1670</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 6</td>
<td>MSR_LASTBRANCH_7_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 687H, 1671</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 7</td>
<td>MSR_LASTBRANCH_8_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 688H, 1672</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 8</td>
<td>MSR_LASTBRANCH_9_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 689H, 1673</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 9</td>
<td>MSR_LASTBRANCH_10_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68AH, 1674</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 10</td>
<td>MSR_LASTBRANCH_11_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68BH, 1675</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11</td>
<td>MSR_LASTBRANCH_12_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68CH, 1676</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12</td>
<td>MSR_LASTBRANCH_13_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68DH, 1677</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13</td>
<td>MSR_LASTBRANCH_14_FROM_IP</td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>See description of MSR_LASTBRANCH_0 at 680H.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 68EH, 1678</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors  (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 68FH, 1679</td>
<td>MSR_LASTBRANCH_15_FROM_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6COH, 1728</td>
<td>MSR_LASTBRANCH_0_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 6C1H, 1729</td>
<td>MSR_LASTBRANCH_1_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C2H, 1730</td>
<td>MSR_LASTBRANCH_2_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C3H, 1731</td>
<td>MSR_LASTBRANCH_3_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C4H, 1732</td>
<td>MSR_LASTBRANCH_4_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C5H, 1733</td>
<td>MSR_LASTBRANCH_5_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C6H, 1734</td>
<td>MSR_LASTBRANCH_6_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C7H, 1735</td>
<td>MSR_LASTBRANCH_7_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C8H, 1736</td>
<td>MSR_LASTBRANCH_8_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6C9H, 1737</td>
<td>MSR_LASTBRANCH_9_TO_IP</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6CAH, 1738</td>
<td>MSR_LASTBRANCH_10_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*See description of MSR_LASTBRANCH_0 at 6C0H.*
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 10</td>
<td>See description of MSR_LASTBRANCH_0 at 6C0H.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6CBH, 1739</td>
<td>MSR_LASTBRANCH_11_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 11</td>
<td>See description of MSR_LASTBRANCH_0 at 6C0H.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6CCH, 1740</td>
<td>MSR_LASTBRANCH_12_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 12</td>
<td>See description of MSR_LASTBRANCH_0 at 6C0H.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6CDH, 1741</td>
<td>MSR_LASTBRANCH_13_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 13</td>
<td>See description of MSR_LASTBRANCH_0 at 6C0H.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6CEH, 1742</td>
<td>MSR_LASTBRANCH_14_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 14</td>
<td>See description of MSR_LASTBRANCH_0 at 6C0H.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 6CFH, 1743</td>
<td>MSR_LASTBRANCH_15_TO_IP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Branch Record 15</td>
<td>See description of MSR_LASTBRANCH_0 at 6C0H.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_EFER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Feature Enables</td>
<td>See Table 2-2.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0081H</td>
<td>IA32_STAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Call Target Address (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0082H</td>
<td>IA32_LSTAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA-32e Mode System Call Target Address (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: C000_0084H</td>
<td>IA32_FMASK</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>System Call Flag Mask (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0100H</td>
<td>IA32_FS_BASE</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Map of BASE Address of FS (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0101H</td>
<td>IA32_GS_BASE</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Map of BASE Address of GS (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0102H</td>
<td>IA32_KERNEL_GS_BASE</td>
<td></td>
<td>3, 4, 6</td>
<td>Unique</td>
</tr>
<tr>
<td>Swap Target of BASE Address of GS (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-58. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)
2.19.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-59 apply to Intel® Xeon® Processor MP with up to 8MB level three cache. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 3 or 4 (see CPUID instruction for more details).

Table 2-59. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache

<table>
<thead>
<tr>
<th>Register Address: Hex</th>
<th>Register Name</th>
<th>Model Availability</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>107CCH</td>
<td>MSR_IFSB_BUSQ0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFSB BUSQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td>3, 4 Shared</td>
<td></td>
</tr>
<tr>
<td>107CDH</td>
<td>MSR_IFSB_BUSQ1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFSB BUSQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107CEH</td>
<td>MSR_IFSB_SNPQ0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFSB SNPQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td>3, 4 Shared</td>
<td></td>
</tr>
<tr>
<td>107CFH</td>
<td>MSR_IFSB_SNPQ1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFSB SNPQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107D0H</td>
<td>MSR_EFSB_DRDY0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EFSB DRDY Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td>3, 4 Shared</td>
<td></td>
</tr>
<tr>
<td>107D1H</td>
<td>MSR_EFSB_DRDY1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EFSB DRDY Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107D2H</td>
<td>MSR_IFSB_CTL6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFSB Latency Event Control Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td>3, 4 Shared</td>
<td></td>
</tr>
<tr>
<td>107D3H</td>
<td>MSR_IFSB_CNTR7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IFSB Latency Event Counter Register (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td>3, 4 Shared</td>
<td></td>
</tr>
</tbody>
</table>

The MSRs listed in Table 2-60 apply to Intel® Xeon® Processor 7100 series. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the
presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 6 (See CPUID instruction for more details.). The performance monitoring MSRs listed in Table 2-60 are shared between logical processors in the same core, but are replicated for each core.

### Table 2-60. MSRs Unique to Intel® Xeon® Processor 7100 Series

<table>
<thead>
<tr>
<th>Register Address: Hex</th>
<th>Register Information</th>
<th>Model</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 107CCH</td>
<td>MSR_EMON_L3_CTR_CTL0</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>GBUSQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107CDH</td>
<td>MSR_EMON_L3_CTR_CTL1</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>GBUSQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107CEH</td>
<td>MSR_EMON_L3_CTR_CTL2</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>GSNPQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107CFH</td>
<td>MSR_EMON_L3_CTR_CTL3</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>GSNPQ Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107D0H</td>
<td>MSR_EMON_L3_CTR_CTL4</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>FSB Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.6, “Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107D1H</td>
<td>MSR_EMON_L3_CTR_CTL5</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>FSB Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107D2H</td>
<td>MSR_EMON_L3_CTR_CTL6</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>FSB Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 107D3H</td>
<td>MSR_EMON_L3_CTR_CTL7</td>
<td>6</td>
<td>Shared</td>
</tr>
<tr>
<td>FSB Event Control and Counter Register (R/W)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.20 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 2-61. The column “Shared/Unique” applies to Intel Core Duo processor. “Unique” means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. “Shared” means the MSR or the bit field in an MSR address governs the operation of both processor cores.

### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 0H, 0</td>
<td>PS_MC_ADDR</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors,” and Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>PS_MC_TYPE</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors,” and Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/ Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 6H, 6</td>
<td>IA32_MONITOR_FILTER_SIZE</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Section 9.10.5, &quot;Monitor/Mwait Address Range Determination,&quot; and Table 2-2.</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Section 18.17, &quot;Time-Stamp Counter,&quot; and Table 2-2.</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>IA32_PLATFORM_ID</td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Platform ID (R)</td>
<td>See Table 2-2. The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1BH, 27</td>
<td>IA32_APIC_BASE</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Section 11.4.4, &quot;Local APIC Status and Location,&quot; and Table 2-2.</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 2AH, 42</td>
<td>MSR_EBL_CR_POWERON</td>
<td></td>
<td>Shared</td>
</tr>
</tbody>
</table>

**Processor Hard Power-On Configuration (R/W)**

Enables and disables processor features; (R) indicates current processor configuration.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bit Description</th>
<th>Shared/ Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data Error Checking Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.</td>
</tr>
<tr>
<td>2</td>
<td>Response Error Checking Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.</td>
</tr>
<tr>
<td>3</td>
<td>MCERR# Drive Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.</td>
</tr>
<tr>
<td>4</td>
<td>Address Parity Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.</td>
</tr>
<tr>
<td>6: 5</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BINIT# Driver Enable (R/W)</td>
<td>1 = Enabled; 0 = Disabled. Note: Not all processor implements R/W.</td>
</tr>
<tr>
<td>8</td>
<td>Output Tri-state Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
</tr>
<tr>
<td>9</td>
<td>Execute BIST (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
</tr>
<tr>
<td>10</td>
<td>MCERR# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
</tr>
<tr>
<td>11</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BINIT# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
</tr>
</tbody>
</table>
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 MByte Power on Reset Vector (R/O)</td>
<td>$1 = 1$ MByte; $0 = 4$ GBytes</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:16</td>
<td>APIC Cluster ID (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>System Bus Frequency (R/O)</td>
<td>$0 = 100$ MHz. $1 = Reserved.$</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Symmetric Arbitration ID (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26:22</td>
<td>Clock Frequency Ratio (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 3AH, 58</td>
<td>IA32_FEATURE_CONTROL</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Control Features in IA-32 Processor (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40H, 64</td>
<td>MSR_LASTBRANCH_0</td>
<td>Last Branch Record 0 (R/W) One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the ‘from’ address and bits 63-32 hold the ‘to’ address. See also: $•$ Last Branch Record Stack TOS at 1C9H. $•$ Section 18.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 41H, 65</td>
<td>MSR_LASTBRANCH_1</td>
<td>Last Branch Record 1 (R/W) See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 42H, 66</td>
<td>MSR_LASTBRANCH_2</td>
<td>Last Branch Record 2 (R/W) See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 43H, 67</td>
<td>MSR_LASTBRANCH_3</td>
<td>Last Branch Record 3 (R/W) See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 44H, 68</td>
<td>MSR_LASTBRANCH_4</td>
<td>Last Branch Record 4 (R/W) See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 45H, 69</td>
<td>MSR_LASTBRANCH_5</td>
<td>Last Branch Record 5 (R/W) See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 46H, 70</td>
<td>MSR_LASTBRANCH_6</td>
<td>Last Branch Record 6 (R/W) See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 47H, 71</td>
<td>MSR_LASTBRANCH_7</td>
<td></td>
<td>Unique</td>
</tr>
</tbody>
</table>
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record 7 (R/W)</td>
<td>See description of MSR_LASTBRANCH_0.</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 79H, 121</td>
<td>IA32_BIOS_UPDT_TRIG</td>
<td>Unique</td>
</tr>
<tr>
<td>BIOS Update Trigger Register (W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 8BH, 139</td>
<td>IA32_BIOS_SIGN_ID</td>
<td>Unique</td>
</tr>
<tr>
<td>BIOS Update Signature ID (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C1H, 193</td>
<td>IA32_PMC0</td>
<td>Unique</td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: C2H, 194</td>
<td>IA32_PMC1</td>
<td>Unique</td>
</tr>
<tr>
<td>Performance Counter Register</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: CDH, 205</td>
<td>MSR_FSB_FREQ</td>
<td></td>
</tr>
<tr>
<td>Scaleable Bus Speed (R/O)</td>
<td>This field indicates the scalable bus clock speed.</td>
<td>Shared</td>
</tr>
<tr>
<td>2:0</td>
<td>• 101B: 100 MHz (FSB 400)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 001B: 133 MHz (FSB 533)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 011B: 167 MHz (FSB 667)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.</td>
<td></td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: E7H, 231</td>
<td>IA32_MPERF</td>
<td>Unique</td>
</tr>
<tr>
<td>Maximum Performance Frequency Clock Count (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: E8H, 232</td>
<td>IA32_APERF</td>
<td>Unique</td>
</tr>
<tr>
<td>Actual Performance Frequency Clock Count (R/W)</td>
<td>See Table 2-2.</td>
<td></td>
</tr>
<tr>
<td>Register Address: FEH, 254</td>
<td>IA32_MTRRCAP</td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 11EH, 281</td>
<td>MSR_BBL_CR_CTL3</td>
<td></td>
</tr>
<tr>
<td>Control Register 3</td>
<td>Used to configure the L2 Cache.</td>
<td>Shared</td>
</tr>
<tr>
<td>0</td>
<td>L2 Hardware Enabled (R/O)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>If the L2 is hardware-enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Indicates if the L2 is hardware-disabled.</td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7:1</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L2 Enabled (R/W)</td>
<td>1 = L2 cache has been initialized. 0 = Disabled (default). Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.</td>
</tr>
<tr>
<td>22:9</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>L2 Not Present (R/O)</td>
<td>0 = L2 Present. 1 = L2 Not Present.</td>
</tr>
<tr>
<td>63:24</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>IA32_SYSENTER_CS</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>IA32_SYSENTER_ESP</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>IA32_SYSENTER_EIP</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Global Machine Check Status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>RIPV</td>
<td>When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.</td>
</tr>
<tr>
<td>1</td>
<td>EIPV</td>
<td>When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.</td>
</tr>
<tr>
<td>2</td>
<td>MCIP</td>
<td>When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.</td>
</tr>
<tr>
<td>63:3</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Register Address: 186H, 390</td>
<td>IA32_PERFEVTSEL0</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 187H, 391</td>
<td>IA32_PERFEVTSEL1</td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Shared</td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 19AH, 410</td>
<td>IA32_CLOCK_MODULATION</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Clock Modulation (R/W)</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 198H, 411</td>
<td>IA32_THERM_INTERRUPT</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Thermal Interrupt Control (R/W)</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2 and Section 15.8.2, “Thermal Monitor.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Thermal Monitor Status (R/W)</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2 and Section 15.8.2, “Thermal Monitor.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19DH, 413</td>
<td>MSR_THERM2_CTL</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Thermal Monitor 2 Control</td>
<td></td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>15:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TM_SELECT (R/W)</td>
<td>Mode of automatic thermal monitor:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Thermal Monitor 2 (thermally-initiated frequency transitions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
<td>Enable Miscellaneous Processor Features (R/W)</td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
</tr>
<tr>
<td>Enable Miscellaneous Processor Features (R/W)</td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Reserved.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>3</td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td>See Table 2-2.</td>
<td>Unique</td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
<tr>
<td>9:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>FERR# Multiplexing Enable (R/W)</td>
<td>1 = FERR# asserted by the processor to indicate a pending break event within the processor</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>0 = Indicates compatible FERR# signaling behavior</td>
<td>This bit must be set to 1 to support XAPIC interrupt model usage.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Branch Trace Storage Unavailable (R/O)</td>
<td>See Table 2-2.</td>
<td>Shared</td>
</tr>
</tbody>
</table>
When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.

When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state.

If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.

<table>
<thead>
<tr>
<th>Register Address: 1C9H, 457</th>
<th>MSR_LASTBRANCH_TOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Branch Record Stack TOS (R/W)</td>
<td></td>
</tr>
<tr>
<td>Contains an index (bits 0-3) that points to the MSR containing the most recent branch record.</td>
<td></td>
</tr>
<tr>
<td>See MSR_LASTBRANCH_0_FROM_IP (at 40H).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 1D9H, 473</th>
<th>IA32_DEBUGCTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug Control (R/W)</td>
<td></td>
</tr>
<tr>
<td>Controls how several debug features are used. Bit definitions are discussed in Table 2-2.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 1DDH, 477</th>
<th>MSR_LER_FROM_LIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Exception Record From Linear IP (R)</td>
<td></td>
</tr>
<tr>
<td>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 1DEH, 478</th>
<th>MSR_LER_TO_LIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Exception Record To Linear IP (R)</td>
<td></td>
</tr>
<tr>
<td>This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Address: 200H, 512</th>
<th>MTRRphysBase0</th>
</tr>
</thead>
</table>

**Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)**
Table 2-61.  MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask0</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 201H, 513</td>
<td>MTRRphysBase1</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask1</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 203H, 515</td>
<td>MTRRphysBase2</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask2</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 205H, 517</td>
<td>MTRRphysBase3</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask3</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 207H, 519</td>
<td>MTRRphysBase4</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask4</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 209H, 521</td>
<td>MTRRphysBase5</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask5</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 20BH, 523</td>
<td>MTRRphysBase6</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask6</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 20DH, 525</td>
<td>MTRRphysBase7</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRphysMask7</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 20EH, 526</td>
<td>MTRRfix64K_00000</td>
<td>Unique</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>MTRRfix16K_80000</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 25BH, 600</td>
<td>MTRRfix16K_A0000</td>
<td>Unique</td>
</tr>
</tbody>
</table>
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td>Bit Description</td>
<td></td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>MTRRfix4K_C0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>MTRRfix4K_C8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 26AH, 618</td>
<td>MTRRfix4K_D0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 26BH, 619</td>
<td>MTRRfix4K_D8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 26CH, 620</td>
<td>MTRRfix4K_E0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 26DH, 621</td>
<td>MTRRfix4K_E8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 26EH, 622</td>
<td>MTRRfix4K_F0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 26FH, 623</td>
<td>MTRRfix4K_F8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td></td>
</tr>
<tr>
<td>Default Memory Types (R/W)</td>
<td>See Table 2-2 and Section 12.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”</td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>IA32_MC1_CTL</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.1, “IA32_MCi_CTL MSRs.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 405H, 1029</td>
<td>IA32_MC1_STATUS</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.2, “IA32_MCi_STATUS MSRS.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>Register Address: 406H, 1030</td>
<td>IA32_MC1_ADDR</td>
<td></td>
</tr>
<tr>
<td>See Section 16.3.2.3, “IA32_MCi_ADDR MSRs.”</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
</tr>
</tbody>
</table>

2-488  Vol. 4
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs.&quot;</td>
<td>IA32_MC2_STATUS</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 409H, 1033</td>
<td>IA32_MC2_ADDR</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.2, &quot;IA32_MCi_STATUS MSRs.&quot;</td>
<td>MSR_MC4_CTL</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 40CH, 1036</td>
<td>MSR_MC4_STATUS</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.2, &quot;IA32_MCi_STATUS MSRS.&quot;</td>
<td>MSR_MC4_ADDR</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 40EH, 1038</td>
<td>MSR_MC3_MISC</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Section 16.3.2.3, &quot;IA32_MC_ADDR MSRs.&quot;</td>
<td>MSR_MC3_ADDR</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Register Address: 410H, 1040</td>
<td>MSR_MC3_CTL</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC3_CTL</td>
<td>See Section 16.3.2.1, &quot;IA32_MC_CTL MSRs.&quot;</td>
<td>MSR_MC3_STATUS</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC3_STATUS</td>
<td>See Section 16.3.2.2, &quot;IA32_MCi_STATUS MSRS.&quot;</td>
<td>MSR_MC3_ADDR</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_CTL</td>
<td>Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.</td>
<td>MSR_MC3_MISC</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_STATUS</td>
<td>Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).</td>
<td>MSR_MC5_CTL</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_ADDR</td>
<td>Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.</td>
<td>MSR_MC5_STATUS</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_MISC</td>
<td>Machine Check Error Reporting Register - contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set.</td>
<td>MSR_MC5_ADDR</td>
<td>Unique</td>
<td></td>
</tr>
<tr>
<td>IA32_MC5_ADDR</td>
<td>Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.</td>
<td>MSR_MC5_MISC</td>
<td>Unique</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-61. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>Register Address: 480H, 1152</td>
<td>IA32_VMX_BASIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reporting Register of Basic VMX Capabilities (R/O)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2 and Appendix A.1, “Basic VMX Information.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 481H, 1153</td>
<td>IA32_VMX_PINBASED_CTLS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of Pin-Based VM-Execution Controls (R/O)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Appendix A.3, “VM-Execution Controls.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 482H, 1154</td>
<td>IA32_VMX_PROCBASED_CTLS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of Primary Processor-Based VM-Execution Controls (R/O)</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Appendix A.3, “VM-Execution Controls.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 483H, 1155</td>
<td>IA32_VMX_EXIT_CTLS</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of VM-Exit Controls (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.4, “VM-Exit Controls.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 484H, 1156</td>
<td>IA32_VMX_ENTRY_CTLS</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of VM-Entry Controls (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.5, “VM-Entry Controls.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 485H, 1157</td>
<td>IA32_VMX_MISC</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Reporting Register of Miscellaneous VMX Capabilities (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.6, “Miscellaneous Data.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 486H, 1158</td>
<td>IA32_VMX_CRO_FIXED0</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.7, “VMX-Fixed Bits in CR0.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 487H, 1159</td>
<td>IA32_VMX_CRO_FIXED1</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.7, “VMX-Fixed Bits in CR0.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 488H, 1160</td>
<td>IA32_VMX_CR4_FIXED0</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.8, “VMX-Fixed Bits in CR4.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 489H, 1161</td>
<td>IA32_VMX_CR4_FIXED1</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.8, “VMX-Fixed Bits in CR4.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 48AH, 1162</td>
<td>IA32_VMX_VMCS_ENUM</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of VMCS Field Enumeration (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.9, “VMCS Enumeration.” (If CPUID.01H:ECX.[bit 5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 48BH, 1163</td>
<td>IA32_VMX_PROCBASED_CTLS2</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td></td>
<td>Capability Reporting Register of Secondary Processor-Based VM-Execution Controls (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Appendix A.3, “VM-Execution Controls.” (If CPUID.01H:ECX.[bit 5] and IA32_VMX_PROCBASED_CTLS[bit 63])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 600H, 1536</td>
<td>IA32_DS_AREA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.21 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.22 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS Save Area (R/W)</td>
<td>See Table 2-2 and Section 20.6.3.4, “Debug Store (DS) Mechanism.”</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>31:0</td>
<td>DS Buffer Management Area</td>
<td>Linear address of the first byte of the DS buffer management area.</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: C000_0080H</td>
<td>IA32_EFER</td>
<td></td>
<td>Unique</td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Execute Disable Bit Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:12</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-62. MSRs in Pentium M Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Shared/Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 0H, 0</td>
<td>P5_MC_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>P5_MC_TYPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 2.23, “MSRs in Pentium Processors.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>IA32_TIME_STAMP_COUNTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Section 18.17, “Time-Stamp Counter,” and see Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>IA32_PLATFORM_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform ID (R)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2AH, 42</td>
<td>MSR_EBL_CR_POWERON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Hard Power-On Configuration (R/W) Enables and disables processor features. (R) Indicates current processor configuration.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data Error Checking Enable (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 = Disabled.</td>
<td>Always 0 on the Pentium M processor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: Hex, Decimal</td>
<td>Register Name</td>
<td>Bit Description</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------</td>
<td>------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Response Error Checking Enable (R)</td>
<td>0 = Disabled. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MCERR# Drive Enable (R)</td>
<td>0 = Disabled. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Address Parity Enable (R)</td>
<td>0 = Disabled. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>6:5</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BINIT# Driver Enable (R)</td>
<td>1 = Enabled; 0 = Disabled. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Output Tri-state Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Execute BIST (R/O)</td>
<td>1 = Enabled; 0 = Disabled.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MCERR# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BINIT# Observation Enabled (R/O)</td>
<td>1 = Enabled; 0 = Disabled. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 MByte Power on Reset Vector (R/O)</td>
<td>1 = 1 MByte; 0 = 4 GBytes. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:16</td>
<td>APIC Cluster ID (R/O)</td>
<td>Always 00B on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>System Bus Frequency (R/O)</td>
<td>0 = 100 MHz. 1 = Reserved. Always 0 on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21:20</td>
<td>Symmetric Arbitration ID (R/O)</td>
<td>Always 00B on the Pentium M processor.</td>
<td></td>
</tr>
<tr>
<td>26:22</td>
<td>Clock Frequency Ratio (R/O)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 40H, 64</td>
<td>MSR_LASTBRANCH_0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-62. MSRs in Pentium M Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Last Branch Record 0 (R/W)</strong></td>
<td>MSR_LASTBRANCH_0</td>
</tr>
<tr>
<td>Register Address: 41H, 65</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 1 (R/W)</strong></td>
<td>MSR_LASTBRANCH_1</td>
</tr>
<tr>
<td>Register Address: 42H, 66</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 2 (R/W)</strong></td>
<td>MSR_LASTBRANCH_2</td>
</tr>
<tr>
<td>Register Address: 43H, 67</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 3 (R/W)</strong></td>
<td>MSR_LASTBRANCH_3</td>
</tr>
<tr>
<td>Register Address: 44H, 68</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 4 (R/W)</strong></td>
<td>MSR_LASTBRANCH_4</td>
</tr>
<tr>
<td>Register Address: 45H, 69</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 5 (R/W)</strong></td>
<td>MSR_LASTBRANCH_5</td>
</tr>
<tr>
<td>Register Address: 46H, 70</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 6 (R/W)</strong></td>
<td>MSR_LASTBRANCH_6</td>
</tr>
<tr>
<td>Register Address: 47H, 71</td>
<td></td>
</tr>
<tr>
<td><strong>Last Branch Record 7 (R/W)</strong></td>
<td>MSR_LASTBRANCH_7</td>
</tr>
<tr>
<td>Register Address: 119H, 281</td>
<td>MSR_BBL_CR_CTL</td>
</tr>
</tbody>
</table>

**Control Register**

- **Register Address:** 11EH, 281
- **MSR_BBL_CR_CTL**

- **Register Address:** 11EH, 281
- **MSR_BBL_CR_CTL3**

**Control Register 3**

- **Register Address:** 119H, 281
- **MSR_BBL_CR_CTL3**

- **Register Address:** 119H, 281
- **MSR_BBL_CR_CTL4**
### Table 2-62. MSRs in Pentium M Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Address: Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>ECC Check Enable (R/O)</td>
<td>This bit enables ECC checking on the cache data bus. ECC is always generated on</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>write cycles.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Disabled (default).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = Enabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For the Pentium M processor, ECC checking on the cache data bus is always enabled.</td>
</tr>
<tr>
<td>7:6</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>L2 Enabled (R/W)</td>
<td>1 = L2 cache has been initialized.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Disabled (default).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.</td>
</tr>
<tr>
<td>22:9</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>L2 Not Present (R/O)</td>
<td>0 = L2 Present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = L2 Not Present.</td>
</tr>
<tr>
<td>63:24</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>IA32_MCG_CAP</td>
<td>Register Address:</td>
<td>Read-only register that provides information about the machine-check architecture of the processor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>179H, 377</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td>Count (R/O)</td>
<td>Indicates the number of hardware unit error reporting banks available in the processor.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>IA32_MCG_CTL Present (R/O)</td>
<td>Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Not supported.</td>
</tr>
<tr>
<td>63:9</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>IA32_MCG_STATUS</td>
<td>Global Machine Check Status</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>RIPV</td>
<td>When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>EIPV</td>
<td>When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>MCIP</td>
<td>When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.</td>
</tr>
<tr>
<td>63:3</td>
<td></td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>Register Address: 198H, 408</td>
<td>IA32_PERF_STATUS</td>
<td>Assembly Characters</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-62. MSRs in Pentium M Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 199H, 409</td>
<td>IA32_PERF_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>See Table 2-2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 19AH, 410</td>
<td>IA32_CLOCK_MODULATION</td>
<td></td>
<td>Clock Modulation (R/W).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Table 2-2 and Section 15.8.3, “Software Controlled Clock Modulation.”</td>
</tr>
<tr>
<td>Register Address: 19BH, 411</td>
<td>IA32_THERM_INTERRUPT</td>
<td></td>
<td>Thermal Interrupt Control (R/W).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Table 2-2 and Section 15.8.2, “Thermal Monitor.”</td>
</tr>
<tr>
<td>Register Address: 19CH, 412</td>
<td>IA32_THERM_STATUS</td>
<td></td>
<td>Thermal Monitor Status (R/W)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Table 2-2 and Section 15.8.2, “Thermal Monitor.”</td>
</tr>
<tr>
<td>Register Address: 19DH, 413</td>
<td>MSR_THERM2_CTL</td>
<td></td>
<td>Thermal Monitor 2 Control</td>
</tr>
<tr>
<td>15:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TM_SELECT (R/W)</td>
<td></td>
<td>Mode of automatic thermal monitor:</td>
</tr>
<tr>
<td></td>
<td>0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Thermal Monitor 2 (thermally-initiated frequency transitions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1A0H, 416</td>
<td>IA32_MISC_ENABLE</td>
<td></td>
<td>Enable Miscellaneous Processor Features (R/W)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Allows a variety of processor functions to be enabled and disabled.</td>
</tr>
<tr>
<td>2:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Automatic Thermal Control Circuit Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows processor clocks to be automatically modulated based on the processor's thermal sensor operation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Disabled (default).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The automatic thermal control circuit enable bit determines if the thermal control circuit (TCC) will be activated when the processor’s internal thermal sensor determines the processor is about to exceed its maximum operating temperature.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>When the TCC is activated and TM1 is enabled, the processors clocks will be forced to a 50% duty cycle. BIOS must enable this feature.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The bit should not be confused with the on-demand thermal control circuit enable bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:4</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Performance Monitoring Available (R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Performance monitoring enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Performance monitoring disabled.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-62. MSRs in Pentium M Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:8</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
| 10                            | FERR# Multiplexing Enable (R/W) | 1 = FERR# asserted by the processor to indicate a pending break event within the processor.  
                                   | 0 = Indicates compatible FERR# signaling behavior. This bit must be set to 1 to support XAPIC interrupt model usage. |
| 12                            | Branch Trace Storage Unavailable (R/O) | 1 = Processor doesn't support branch trace storage (BTS)  
                                   | 0 = BTS is supported.                                                              |
| 15:13                         | Processor Event Based Sampling Unavailable (R/O) | 1 = Processor does not support processor event based sampling (PEBS);  
                                   | 0 = PEBS is supported. The Pentium M processor does not support PEBS.              |
| 16                            | Enhanced Intel SpeedStep Technology Enable (R/W) | 1 = Enhanced Intel SpeedStep Technology enabled.  
                                   | On the Pentium M processor, this bit may be configured to be read-only.            |
| 22:17                         | Reserved.              |                                                                                  |
| 23                            | xTPR Message Disable (R/W) | When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority. The default is processor specific. |
| 63:24                         | Reserved.              |                                                                                  |
| Register Address: 1C9H, 457   | MSR_LASTBRANCH_TOS    | Last Branch Record Stack TOS (R/W) Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also:  
                                   | • MSR_LASTBRANCH_0_FROM_IP (at 40H).  
<pre><code>                               | • Section 18.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).” |
</code></pre>
<p>| Register Address: 1D9H, 473   | MSR_DEBUGCTLB         | Debug Control (R/W) Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 18.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors).” |
| Register Address: 1DDH, 477   | MSR_LER_TO_LIP        | Last Exception Record To Linear IP (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 18.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors),” and Section 18.16.2, “Last Branch and Last Exception MSRs.” |
| Register Address: 1DEH, 478   | MSR_LER_FROM_LIP      |                                                                                  |</p>
<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2FFH, 767</td>
<td>IA32_MTRR_DEF_TYPE</td>
<td>Last Exception Record From Linear IP (R)</td>
<td>Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 18.15, “Last Branch, Interrupt, and Exception Recording (Pentium M Processors),” and Section 18.16.2, “Last Branch and Last Exception MSRs.”</td>
</tr>
<tr>
<td>400H, 1024</td>
<td>IA32_MC0_CTL</td>
<td>Default Memory Types (R/W)</td>
<td>Sets the memory type for the regions of physical memory that are not mapped by the MTRRs. See Section 12.11.2.1, “IA32_MTRR_DEF_TYPE MSR.”</td>
</tr>
<tr>
<td>401H, 1025</td>
<td>IA32_MC0_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>402H, 1026</td>
<td>IA32_MC0_ADDR</td>
<td>The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
</tr>
<tr>
<td>404H, 1028</td>
<td>IA32_MC1_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>405H, 1029</td>
<td>IA32_MC1_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>406H, 1030</td>
<td>IA32_MC1_ADDR</td>
<td>The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
</tr>
<tr>
<td>408H, 1032</td>
<td>IA32_MC2_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>409H, 1033</td>
<td>IA32_MC2_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40AH, 1034</td>
<td>IA32_MC2_ADDR</td>
<td>The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.</td>
<td></td>
</tr>
<tr>
<td>40CH, 1036</td>
<td>MSR_MC4_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40DH, 1037</td>
<td>MSR_MC4_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40EH, 1038</td>
<td>MSR_MC4_ADDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as “architectural” and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

### Table 2-63. MSRs in the P6 Family Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Bit Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 0H, 0</td>
<td>P5_MC_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>P5_MC_TYPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>TSC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>IA32_PLATFORM_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform ID (R)</td>
<td>The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.22 **MSRS IN THE P6 FAMILY PROCESSORS**

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as “architectural” and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

### Table 2-63. MSRs in the P6 Family Processors

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
<th>Bit Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 0H, 0</td>
<td>P5_MC_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1H, 1</td>
<td>P5_MC_TYPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 10H, 16</td>
<td>TSC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17H, 23</td>
<td>IA32_PLATFORM_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platform ID (R)</td>
<td>The operating system can use this MSR to determine “slot” information for the processor and the proper microcode update to load.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>52:50</td>
<td>Platform Id (R)</td>
<td></td>
<td>Contains information concerning the intended platform for the processor.</td>
</tr>
<tr>
<td></td>
<td>52 51 50</td>
<td>Processor Flag 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td></td>
<td>0 0 1</td>
<td>Processor Flag 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td></td>
<td>0 1 0</td>
<td>Processor Flag 2</td>
<td>0 1 0</td>
</tr>
<tr>
<td></td>
<td>0 1 1</td>
<td>Processor Flag 3</td>
<td>0 1 1</td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
<td>Processor Flag 4</td>
<td>1 0 0</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td>Processor Flag 5</td>
<td>1 0 1</td>
</tr>
<tr>
<td></td>
<td>1 1 0</td>
<td>Processor Flag 6</td>
<td>1 1 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td>Processor Flag 7</td>
<td>1 1 1</td>
</tr>
<tr>
<td>56:53</td>
<td>L2 Cache Latency Read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59:57</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>Clock Frequency Ratio Read.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:61</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 1BH, 27</td>
<td>APIC_BASE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section 11.4.4, “Local APIC Status and Location.”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Boot Strap Processor Indicator Bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = BSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:9</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>APIC Global Enable Bit - Permanent till reset</td>
<td></td>
<td>1 = Enabled.</td>
</tr>
<tr>
<td></td>
<td>0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31:12</td>
<td>APIC Base Address.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 2AH, 42</td>
<td>EBL_CR_POWERON</td>
<td></td>
<td>Processor Hard Power-On Configuration</td>
</tr>
<tr>
<td></td>
<td>(R/W) Enables and disables processor features, and (R) indicates current processor configuration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Reserved(^1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Data Error Checking Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Response Error Checking Enable FRCERR Observation Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AERR# Drive Enable (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BERR# Enable for Initiator Bus Requests (R/W)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Disabled.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) Reserved
### Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Reserved.</td>
<td></td>
</tr>
</tbody>
</table>
| 6                              | BERR# Driver Enable for Initiator Internal Errors (R/W) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 7                              | BINIT# Driver Enable (R/W) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 8                              | Output Tri-state Enabled (R) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 9                              | Execute BIST (R) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 10                             | AERR# Observation Enabled (R) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 11                             | Reserved.     |                 |
| 12                             | BINIT# Observation Enabled (R) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 13                             | In Order Queue Depth (R) | 1 = 1.  
|                                |               | 0 = 8.         |
| 14                             | 1-MByte Power on Reset Vector (R) | 1 = 1MByte.  
|                                |               | 0 = 4GBytes.   |
| 15                             | FRC Mode Enable (R) | 1 = Enabled.  
|                                |               | 0 = Disabled.   |
| 17:16                          | APIC Cluster ID (R) |                 |
| 19:18                          | System Bus Frequency (R) | 00 = 66MHz.  
|                                |               | 10 = 100Mhz.   |
|                                |               | 01 = 133MHz.   |
|                                |               | 11 = Reserved. |
| 21:20                          | Symmetric Arbitration ID (R) |                 |
| 25:22                          | Clock Frequency Ratio (R) |                 |
| 26                             | Low Power Mode Enable (R/W) |                 |
| 27                             | Clock Frequency Ratio |                 |
| 63:28                          | Reserved. 1 |                 |
| Register Address: 33H, 51      | MSR_TEST_CTRL |                 |
### Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Information / Bit Fields</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Test Control Register</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29:0</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Streaming Buffer Disable</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Disable LOCK#</td>
<td>Assertion for split locked access.</td>
</tr>
<tr>
<td><strong>Register Address: 79H, 121</strong></td>
<td>BIOS_UPDT_TRIG</td>
<td></td>
</tr>
<tr>
<td><strong>BIOS Update Trigger Register.</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 88H, 136</strong></td>
<td>BBL_CR_D0[63:0]</td>
<td>Chunk 0 data register D[63:0]; used to write to and read from the L2.</td>
</tr>
<tr>
<td><strong>Register Address: 89H, 137</strong></td>
<td>BBL_CR_D1</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 8AH, 138</strong></td>
<td>BBL_CR_D2</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 8BH, 139</strong></td>
<td>BIOS_SIGN/BBL_CR_D3</td>
<td>BIOS Update Signature Register or Chunk 3 data register D[63:0]. Used to write to and read from the L2 depending on the usage model.</td>
</tr>
<tr>
<td><strong>Register Address: C1H, 193</strong></td>
<td>PerfCtr0 (PERFCTR0)</td>
<td></td>
</tr>
<tr>
<td><strong>Performance Counter Register</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: C2H, 194</strong></td>
<td>PerfCtr1 (PERFCTR1)</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: FEH, 254</strong></td>
<td>MTRRcap</td>
<td></td>
</tr>
<tr>
<td><strong>Memory Type Range Registers</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 116H, 278</strong></td>
<td>BBL_CR_ADDR</td>
<td>Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses.</td>
</tr>
<tr>
<td>2:0</td>
<td>Reserved; set to 0.</td>
<td></td>
</tr>
<tr>
<td>31:3</td>
<td>Address bits [35:3].</td>
<td></td>
</tr>
<tr>
<td>63:32</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td><strong>Register Address: 118H, 280</strong></td>
<td>BBL_CR_DECC</td>
<td>Data ECC register D[7:0]: used to write ECC and read ECC to/from L2.</td>
</tr>
<tr>
<td><strong>Register Address: 119H, 281</strong></td>
<td>BBL_CR_CTL</td>
<td>Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response.</td>
</tr>
</tbody>
</table>
### Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:0</td>
<td>L2 Command:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01100 = Data Read w/ LRU update (RLU).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01110 = Tag Read w/ Data Read (TRR).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01111 = Tag Inquire (TI).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00010 = L2 Control Register Read (CR).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00011 = L2 Control Register Write (CW).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>010 + MESI encode = Tag Write w/ Data Read (TWR).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>111 + MESI encode = Tag Write w/ Data Write (TWW).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100 + MESI encode = Tag Write (Tw).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:5</td>
<td>7</td>
<td>State to L2</td>
<td></td>
</tr>
<tr>
<td>9:8</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:10</td>
<td>Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11</td>
<td>Way to L2</td>
<td></td>
</tr>
<tr>
<td>13:12</td>
<td>Modified - 11, Exclusive - 10, Shared - 01, Invalid - 00</td>
<td>Way from L2</td>
<td></td>
</tr>
<tr>
<td>15:14</td>
<td>State from L2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>L2 Hit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20:19</td>
<td>User supplied ECC.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Processor number: ²</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disable = 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enable = 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:22</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 11AH, 282</td>
<td>BBL_CR_TRIG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 11BH, 283</td>
<td>BBL_CR_BUSY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 11EH, 286</td>
<td>BBL_CR_CTL3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control register 3: used to configure the L2 Cache.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>L2 Configured (read/write).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:1</td>
<td>L2 Cache Latency (read/write).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ECC Check Enable (read/write).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Address Parity Check Enable (read/write).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CRTN Parity Check Enable (read/write).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L2 Enabled (read/write).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-63. MSRs in the P6 Family Processors  (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:9</td>
<td>L2 Associativity (read only):</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00 = Direct Mapped.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 = 2 Way.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10 = 4 Way.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11 = Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:11</td>
<td>Number of L2 banks (read only).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17:13</td>
<td>Cache size per bank (read/write):</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000001 = 256 KBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00010 = 512 KBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00100 = 1 MByte.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01000 = 2 MBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10000 = 4 MBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Cache State error checking enable (read/write).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22:20</td>
<td>L2 Physical Address Range support:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>111 = 64 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>110 = 32 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>101 = 16 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100 = 8 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>011 = 4 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>010 = 2 GBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>001 = 1 GByte.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>000 = 512 MBytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>L2 Hardware Disable (read only).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Cache bus fraction (read only).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63:26</td>
<td>Reserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 174H, 372</td>
<td>SYSENTER_CS_MSR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS register target for CPL 0 code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 175H, 373</td>
<td>SYSENTER_ESP_MSR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack pointer for CPL 0 stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 176H, 374</td>
<td>SYSENTER_EIP_MSR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPL 0 code entry point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 179H, 377</td>
<td>MCG_CAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Global Control Register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17AH, 378</td>
<td>MCG_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCI_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 17BH, 379</td>
<td>MCG_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 186H, 390</td>
<td>PerfEvtSel0 (EVNTSELO)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2-63. MSRs in the P6 Family Processors  (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Event Select Register 0 (R/W)</td>
<td></td>
<td>Event Select</td>
<td>Refer to Performance Counter section for a list of event encodings.</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>UMASK (Unit Mask)</td>
<td>Unit mask register set to 0 to enable all count options.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>USER</td>
<td>Controls the counting of events at Privilege levels of 1, 2, and 3.</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>OS</td>
<td>Controls the counting of events at Privilege level of 0.</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>E</td>
<td>Occurrence/Duration Mode Select: 1 = Occurrence. 0 = Duration.</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>PC</td>
<td>Enabled the signaling of performance counter overflow via BPO pin.</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>INT</td>
<td>Enables the signaling of counter overflow via input to APIC: 1 = Enable. 0 = Disable.</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>ENABLE</td>
<td>Enables the counting of performance events in both counters: 1 = Enable. 0 = Disable.</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>INV</td>
<td>Inverts the result of the CMASK condition: 1 = Inverted. 0 = Non-Inverted.</td>
</tr>
<tr>
<td>31:24</td>
<td></td>
<td>CMASK (Counter Mask)</td>
<td></td>
</tr>
<tr>
<td>Register Address: 187H, 391</td>
<td></td>
<td>PerfEvtSel1 (EVNTSEL1)</td>
<td></td>
</tr>
<tr>
<td>Performance Event Select for Counter 1 (R/W)</td>
<td></td>
<td>Event Select</td>
<td>Refer to Performance Counter section for a list of event encodings.</td>
</tr>
<tr>
<td>7:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td></td>
<td>UMASK (Unit Mask)</td>
<td>Unit mask register set to 0 to enable all count options.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>USER</td>
<td>Controls the counting of events at Privilege levels of 1, 2, and 3.</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>OS</td>
<td>Controls the counting of events at Privilege level of 0.</td>
</tr>
</tbody>
</table>
### Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
</table>
| 18 18                        | E             | Occurrence/Duration Mode Select:  
|                                |               | 1 = Occurrence.  
|                                |               | 0 = Duration.   |
| 19 19                        | PC            | Enabled the signaling of performance counter overflow via BP0 pin. |
| 20 20                        | INT           | Enables the signaling of counter overflow via input to APIC.  
|                                |               | 1 = Enable.  
|                                |               | 0 = Disable.   |
| 23 23                        | INV           | Inverts the result of the CMASK condition.  
|                                |               | 1 = Inverted.   
|                                |               | 0 = Non-Inverted.   |
| 31:24 CMASK (Counter Mask)   |               | DEBUGCTLMSR  
| Register Address: 1D9H, 473   |               | Enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages.  
|                                |               | This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode.   |
| 0 0                          | Enable/Disable Last Branch Records | 0: Enable/Disable Last Branch Records  
| 1 1                          | Branch Trap Flag | 1: Branch Trap Flag  
| 2 2                          | Performance Monitoring/Break Point Pins | 2: Performance Monitoring/Break Point Pins |
| 3 3                          | Performance Monitoring/Break Point Pins | 3: Performance Monitoring/Break Point Pins |
| 4 4                          | Performance Monitoring/Break Point Pins | 4: Performance Monitoring/Break Point Pins |
| 5 5                          | Performance Monitoring/Break Point Pins | 5: Performance Monitoring/Break Point Pins |
| 6 6                          | Enable/Disable Execution Trace Messages | 6: Enable/Disable Execution Trace Messages |
| 31:7 Reserved.               |               | 31:7 Reserved. |
| Register Address: 1D8H, 475   | LASTBRANCHFROMIP | 32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. |
| Register Address: 1DCH, 476   | LASTBRANCHTOIP | 32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. |
| Register Address: 1DDH, 477   | LASTINTFROMIP | 32-bit register for recording the instruction pointers for the last interrupt that the processor took prior to a debug exception being generated. |
| Register Address: 200H, 512   | MTRRphysBase0 | Memory Type Range Registers |
| Register Address: 201H, 513   | MTRRphysMask0 | Memory Type Range Registers |
Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 202H, 514</td>
<td>MTRRphysBase1</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 203H, 515</td>
<td>MTRRphysMask1</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 204H, 516</td>
<td>MTRRphysBase2</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 205H, 517</td>
<td>MTRRphysMask2</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 206H, 518</td>
<td>MTRRphysBase3</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 207H, 519</td>
<td>MTRRphysMask3</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 208H, 520</td>
<td>MTRRphysBase4</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 209H, 521</td>
<td>MTRRphysMask4</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20AH, 522</td>
<td>MTRRphysBase5</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20BH, 523</td>
<td>MTRRphysMask5</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20CH, 524</td>
<td>MTRRphysBase6</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20DH, 525</td>
<td>MTRRphysMask6</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20EH, 526</td>
<td>MTRRphysBase7</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 20FH, 527</td>
<td>MTRRphysMask7</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 250H, 592</td>
<td>MTRRfix64K_00000</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 258H, 600</td>
<td>MTRRfix16K_80000</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 259H, 601</td>
<td>MTRRfix16K_A0000</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 268H, 616</td>
<td>MTRRfix4K_C0000</td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td></td>
</tr>
<tr>
<td>Register Address: 269H, 617</td>
<td>MTRRfix4K_C8000</td>
</tr>
</tbody>
</table>
### Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information / Bit Fields</th>
<th>Register Name</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 26AH, 618</td>
<td>MTRRfix4K_D0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 26BH, 619</td>
<td>MTRRfix4K_D8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 26CH, 620</td>
<td>MTRRfix4K_E0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 26DH, 621</td>
<td>MTRRfix4K_E8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 26EH, 622</td>
<td>MTRRfix4K_F0000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 26FH, 623</td>
<td>MTRRfix4K_F8000</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Registers</td>
<td>Register Address: 2FFH, 767</td>
<td>MTRRdefType</td>
<td></td>
</tr>
<tr>
<td>2:0</td>
<td>Default memory type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Fixed MTRR enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MTRR Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 400H, 1024</td>
<td>MCO_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Error Reporting Register Address: 401H, 1025</td>
<td>MCO_STATUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Error Reporting Register Address: 402H, 1026</td>
<td>MC0_MISC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Check Error Reporting Register Address: 403H, 1027</td>
<td>MC0_ADDR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register Address: 404H, 1028</td>
<td>MC1_CTL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
- Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
MODEL-SPECIFIC REGISTERS (MSRS)

2.23 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5_MC_ADDR, P5_MC_TYPE, and TSC MSRs (named IA32_P5_MC_ADDR, IA32_P5_MC_TYPE, and IA32_TIME_STAMP_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 2.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.

NOTES
1. Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
2. The processor number feature may be disabled by setting bit 21 of the BBL_CR_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL_CR_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
3. The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

Table 2-63. MSRs in the P6 Family Processors (Contd.)

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Name</th>
<th>Register Information / Bit Fields</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Address: 405H, 1029</td>
<td>MC1_STATUS</td>
<td></td>
<td>Bit definitions same as MC0_STATUS.</td>
</tr>
<tr>
<td>Register Address: 406H, 1030</td>
<td>MC1_ADDR</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 407H, 1031</td>
<td>MC1_MISC</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 408H, 1032</td>
<td>MC2_CTL</td>
<td></td>
<td>Bit definitions same as MC0_STATUS.</td>
</tr>
<tr>
<td>Register Address: 409H, 1033</td>
<td>MC2_STATUS</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 40AH, 1034</td>
<td>MC2_ADDR</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 40BH, 1035</td>
<td>MC2_MISC</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 40CH, 1036</td>
<td>MC4_CTL</td>
<td></td>
<td>Bit definitions same as MC0_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.</td>
</tr>
<tr>
<td>Register Address: 40DH, 1037</td>
<td>MC4_STATUS</td>
<td></td>
<td>Defined in MCA architecture but not implemented in P6 Family processors.</td>
</tr>
<tr>
<td>Register Address: 40EH, 1038</td>
<td>MC4_ADDR</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 40FH, 1039</td>
<td>MC4_MISC</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 410H, 1040</td>
<td>MC3_CTL</td>
<td></td>
<td>Bit definitions same as MC0_STATUS.</td>
</tr>
<tr>
<td>Register Address: 411H, 1041</td>
<td>MC3_STATUS</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 412H, 1042</td>
<td>MC3_ADDR</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
<tr>
<td>Register Address: 413H, 1043</td>
<td>MC3_MISC</td>
<td></td>
<td>Defined in MCA architecture but not implemented in the P6 family processors.</td>
</tr>
</tbody>
</table>
Table 2-64. MSRs in the Pentium Processor

<table>
<thead>
<tr>
<th>Register Address: Hex, Decimal</th>
<th>Register Information</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0H, 0</td>
<td>P5_MC_ADDR</td>
<td></td>
</tr>
<tr>
<td>See Section 16.10.2, “Pentium Processor Machine-Check Exception Handling.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1H, 1</td>
<td>P5_MC_TYPE</td>
<td></td>
</tr>
<tr>
<td>See Section 16.10.2, “Pentium Processor Machine-Check Exception Handling.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10H, 16</td>
<td>TSC</td>
<td></td>
</tr>
<tr>
<td>See Section 18.17, “Time-Stamp Counter.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11H, 17</td>
<td>CESR</td>
<td></td>
</tr>
<tr>
<td>See Section 20.6.9.1, “Control and Event Select Register (CESR).”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12H, 18</td>
<td>CTR0</td>
<td></td>
</tr>
<tr>
<td>Section 20.6.9.3, “Events Counted.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13H, 19</td>
<td>CTR1</td>
<td></td>
</tr>
<tr>
<td>Section 20.6.9.3, “Events Counted.”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MODEL-SPECIFIC REGISTERS (MSRs)
INDEX
INDEX

A
A20M# signal, Vol.3-21-2, Vol.1-3-34, Vol.3-24-4
AAA instruction, Vol.1-7-9, Vol.2-3-19, Vol.2-3-21
AAD instruction, Vol.1-7-10, Vol.2-3-21
AAM instruction, Vol.1-7-10, Vol.2-3-23
AAS instruction, Vol.1-7-10, Vol.2-3-25
Abort
description of, Vol.3-6-5
restarting a program or task after, Vol.3-6-6
AC (alignment check) flag, EFLAGS register, Vol.1-3-17, Vol.3-2-11, Vol.3-6-50, Vol.1-23-6
Access rights
checking, Vol.3-2-24
checking caller privileges, Vol.3-5-26
description of, Vol.3-5-24
invalid values, Vol.1-23-19
Access rights, segment descriptor, Vol.1-6-8, Vol.1-6-13
ADDPD instruction, Vol.1-11-6, Vol.2-3-34
ADDP- Add Packed Single-Precision Floating-Point Values, Vol.2-3-37
ADDP- instruction, Vol.1-10-8
Address
size prefix, Vol.3-22-1
space, of task, Vol.3-8-16
Address size attribute
code segment, Vol.1-3-18
description of, Vol.1-3-18
of stack, Vol.1-6-3
Address sizes, Vol.1-3-9
Address space
64-bit mode, Vol.1-3-1, Vol.1-3-5
compatibility mode, Vol.1-3-1
overview of, Vol.1-3-2
physical, Vol.1-3-6
Address translation
in real-address mode, Vol.3-21-2
to logical linear, Vol.3-3-7
overview, Vol.3-3-6
Addressing methods
RIP-relative, Vol.2-2-12
Addressing modes
assembler, Vol.1-3-24
base plus displacement, Vol.1-3-23
base plus index plus displacement, Vol.1-3-23
base plus index time scale plus displacement, Vol.1-3-23, Vol.1-3-24
canonical address, Vol.1-3-10
effective address, Vol.1-3-23
immediate operands, Vol.1-3-20
index, Vol.1-3-22, Vol.1-3-24

ADDPS instruction
ADDPS- Add Packed Single-Precision Floating-Point Values

ADD instruction

ADD instruction

ADCPD instruction, Vol.1-1-2-1
Local APIC and I/O APICs, Vol.3-11-4

x87 FPU, Vol.1-8-1

See also: IA-32e mode, compatibility mode

ADDSD- Add Scalar Double-Precision Floating-Point Values
ADDSD instruction, Vol.1-11-6, Vol.2-3-40
ADDSS- Add Scalar Single-Precision Floating-Point Values
ADDSS instruction, Vol.1-10-8
ADEX- Signed Integer Addition of Two Operands with Overflow Flag, Vol.2-3-49
Advanced media boost, Vol.1-2-11
Advanced power management
C-state and Sub C-state, Vol.3-15-35
Mwait extensions, Vol.3-15-35
See also: thermal monitoring
Advanced programmable interrupt controller (see I/O APIC or Local APIC)
advanced smart cache

AESKEYGENASSIST - AES Round Key Generation Assist

AESDEC128KL- Perform Ten Rounds of AES Decryption Flow Using 128-Bit Key, Vol.2-3-53
AESDEC256KL- Perform 14 Rounds of AES Decryption Flow Using 256-Bit Key, Vol.2-3-55
AESDECLAST- Perform Last Round of an AES Decryption Flow, Vol.2-3-57
AESDEC- Perform One Round of an AES Decryption Flow, Vol.2-3-51
AESDECWIDE128KL- Perform Ten Rounds of AES Decryption Flow on 8 Blocks with 128-Bit Key, Vol.2-3-59
AESDECWIDE256KL- Perform 14 Rounds of AES Decryption Flow on 8 Blocks with 256-Bit Key, Vol.2-3-61
AESCENC128KL- Perform Ten Rounds of AES Encryption Flow Using 128-Bit Key, Vol.2-3-65
AESCENC256KL- Perform 14 Rounds of AES Encryption Flow Using 256-Bit Key, Vol.2-3-69
AESNENCLAST- Perform Last Round of an AES Encryption Flow, Vol.2-3-69
AESNENC- Perform One Round of an AES Encryption Flow, Vol.2-3-63
AESNENCWIDE128KL- Perform Ten Rounds of AES Encryption Flow on 8 Blocks with 128-Bit Key, Vol.2-3-71
AESNENCWIDE256KL- Perform 14 Rounds of AES Encryption Flow on 8 Blocks with 256-Bit Key, Vol.2-3-73
AESIMC- Perform the AES InvMixColumn Transformation, Vol.2-3-75
AESKEYGENASSIST - AES Round Key Generation Assist, Vol.2-3-76
AH register, Vol.1-3-12
AL register, Vol.1-3-12
Alignment
checking, Vol.3-5-27
words, doublewords, quadwords, Vol.1-4-2
AM (alignment mask) flag
CR0 control register, Vol.3-2-15, Vol.1-23-18
ANDNPD instruction, Vol.1-11-7
ANDNPD- Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values, Vol.2-3-90
ANDNPD instruction, Vol.1-10-9
ANDPD- Bitwise Logical AND of Packed Double Precision Floating-Point Values, Vol.2-3-81
ANDPD instruction, Vol.1-11-7, Vol.2-3-80
ANDPS- Bitwise Logical AND of Packed Single Precision Floating-Point Values, Vol.2-3-84
ANDPS instruction, Vol.1-10-9
APIC, Vol.3-11-40, Vol.3-11-41
APIC bus
arbitration mechanism and protocol, Vol.3-11-26, Vol.3-11-33
bus message format, Vol.3-11-34, Vol.3-11-47
diagram of, Vol.3-11-2, Vol.3-11-3
EOI message format, Vol.3-11-15, Vol.3-11-47

INDEX
nonfocused lowest priority message, Vol.3-11-49
short message format, Vol.3-11-48
SML message, Vol.4-32-2
status cycles, Vol.3-11-50
structure of, Vol.3-11-3
See also
local APIC
APIC flag, CPUID instruction, Vol.3-11-7
APIC (see I/O APIC or Local APIC)
Arctangent, x87 FPU operation
BEXTR—Bit Field Extract
BEXTR—Bit Field Extract, Vol.2-3-95
BD (debug register access detected) flag, DR6 register
B0-B3 (BP condition detected) flags
B (default stack size) flag
B (default size) flag, segment descriptor
B (busy) flag
Automatic bus locking, Vol.3-9-3
effects of a locked operation on internal processor caches, Vol.3-9-6
guaranteed, description of, Vol.3-9-2
overview of, Vol.3-9-1, Vol.3-9-3
software-controlled bus locking, Vol.3-9-4
At-retirement
counting, Vol.3-20-101, Vol.3-20-117
Vol.3-20-121
authenticated code execution mode, Vol.1-6-3
Auto HALT restartfield, SMM, Vol.4-32-14
SM, Vol.4-32-14
Automatic bus locking, Vol.3-9-3
Automatic thermal monitoring mechanism, Vol.3-15-36
AX register, Vol.1-3-12
B
B (busy) flag
TSS descriptor, Vol.3-8-5, Vol.3-8-11, Vol.3-8-16
B (default size) flag, segment descriptor, Vol.1-3-18
B (default stack size) flag
segment descriptor, Vol.3-22-1, Vol.1-23-33
BO-B3 (BP condition detected) flags
DR6 register, Vol.3-18-4
Backlink (see Previous task link)
Base address fields, segment descriptor, Vol.3-3-10
Basic execution environment, Vol.1-3-2
Basic programming environment, Vol.1-7-1
B-bit, x87 FPU status word, Vol.1-8-5
BCD integers
relationship to status flags, Vol.1-3-17
x87 FPU encoding, Vol.1-4-10
BD (debug register access detected) flag, DR6 register, Vol.3-18-11
BEXTR—Bit Field Extract, Vol.2-3-95
BH register, Vol.1-3-12
Bias value
numeric overflow, Vol.1-8-29
numeric underflow, Vol.1-8-30
Biased exponent, Vol.1-4-13
Biased constant, for floating-point numbers, Vol.1-4-6
Binary-coded decimal (see BCD)
BNIT# signal, Vol.3-2-26
Bit field, Vol.1-4-7
Bit order, Vol.1-1-6, Vol.2-1-5, Vol.3-1-7, Vol.4-1-5
BL register, Vol.1-3-12
BLENDPD — Blend Packed Double Precision Floating-Point Values,
Vol.2-3-96
BLENDPS — Blend Packed Single Precision Floating-Point Values,
Vol.2-3-98
BSL-Extract Lowest Set Isolated Bit, Vol.2-3-105
BSLMSK - Get Mask Up to Lowest Set Bit, Vol.2-3-106
BSLR — Reset Lowest Set Bit, Vol.2-3-107
BDNCL—Check Lower Bound, Vol.2-3-108
BDNU/BBDNC—Check Upper Bound, Vol.2-3-110
BNDLX—Load Extended Bounds Using Address Translation, Vol.2-3-112
BNMK—Make Bounds, Vol.2-3-115
BNMOV—Move Bounds, Vol.2-3-117
BNDTX—Store Extended Bounds Using Address Translation, Vol.2-3-120
BOUND instruction, Vol.1-6-18, Vol.1-7-18, Vol.1-7-23, Vol.2-3-123,
Vol.3-2-5, Vol.3-6-4, Vol.3-6-30
BOUND range exceeded exception (#BR), Vol.1-6-19, Vol.2-3-123, Vol.3-6-30
BOUND—Check Array Index Against Bounds, Vol.2-3-123
BP register, Vol.1-3-12
Branch
task-switch exception condition
BLDS command
general-detect exception condition
on x87 FPU condition codes, Vol.1-8-6, Vol.1-8-20
prediction, Vol.1-2-8
Branch hints, Vol.2-2-2
Branch record
branch trace message, Vol.3-18-15
IA-32e mode, Vol.3-18-23
saving as a branch trace message, Vol.3-18-15
structure, Vol.3-18-37
structure of in BTS buffer, Vol.3-18-21
Branch trace message (see BTM)
Branch trace store (see BTS)
Brand information, Vol.2-3-254
processor brand index, Vol.2-3-256
processor brand string, Vol.2-3-254
Breakpoint exception (#BP), Vol.3-6-4, Vol.3-6-28, Vol.3-18-11
Breakpoints
data breakpoint, Vol.3-18-6
data breakpoint exception conditions, Vol.3-18-10
description of, Vol.3-18-1
DRO-DR3 debug registers, Vol.3-18-4
event, Vol.3-18-6
exception, Vol.3-6-28
field recognition, Vol.3-18-6, Vol.3-18-7
general-detect exception condition, Vol.3-18-10
instruction breakpoint exception condition, Vol.3-18-6
instruction breakpoint exception condition, Vol.3-18-9
I/O breakpoint exception conditions, Vol.3-18-10
LEN0 - LEN3 (Length) fields
DR7 register, Vol.3-18-6
R/W0/R/w3 (read/write) fields
DR7 register, Vol.3-18-5
single-step exception condition, Vol.3-18-11
task-switch exception condition, Vol.3-18-11
BS (single step) flag, DR6 register, Vol.3-18-4
BSF instruction, Vol.1-7-14, Vol.2-3-125
BSF flag, IA32_APIC_BASE MSR, Vol.3-11-8
BSR instruction, Vol.1-7-14, Vol.2-3-127
BT (task switch) flag, DR6 register, Vol.3-18-4, Vol.3-18-11
Vol.2-3-610, Vol.3-9-4
BTF (single-step on branches) flag
DEBUGCTLMSR MSR, Vol.3-18-41
BTMs (branch trace messages)
description of, Vol.3-18-15

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 + Index -3
Canonical address, Vol.1-3-10
GETSEC, Vol.1-6-3
Capability MSRs
See VMX capability MSRs
Catastrophic shutdown detector
Thermal monitoring
catastrophic shutdown detector, Vol.3-15-37
CBW instruction, Vol.1-7-7, Vol.2-3-156
CCD and C1C (counter control) fields, CESR MSR (Pentium processor), Vol.3-20-137
CDQ instruction, Vol.1-7-7, Vol.2-3-316
CDQE instruction, Vol.1-7-7, Vol.2-3-156
Celeron processor
description of, Vol.1-2-2
CESR (control and event select) MSR (Pentium processor), Vol.3-20-136, Vol.3-20-137
CH register, Vol.1-3-12
CL register, Vol.1-3-12
CLC instruction, Vol.1-3-16, Vol.1-7-21, Vol.2-3-158
CLD instruction, Vol.1-3-17, Vol.1-7-21, Vol.2-3-159
CLFLUSH feature flag, CPUID instruction, Vol.3-10-8
CPUID flag, Vol.2-3-246
Clocks
counting processor clocks, Vol.3-20-138
Hyper-Threading Technology, Vol.3-20-138
nominal CPI, Vol.3-20-138
non-halted clockticks, Vol.3-20-138
non-halted CPI, Vol.3-20-138
non-sleep clockticks, Vol.3-20-138
time stamp counter, Vol.3-20-138
Cluster model, local APIC, Vol.3-11-24
CMC instruction, Vol.1-3-16, Vol.1-7-21, Vol.2-3-174
CMOVcc flag, Vol.2-3-246
CPUID flag, Vol.2-3-246
CM instruction, Vol.1-7-8, Vol.2-3-179
CMPPD-Compare Packed Double-Precision Floating-Point Values, Vol.2-3-181
CMPPPD instruction, Vol.1-11-7
CMPPPS-Compare Packed Single-Precision Floating-Point Values, Vol.2-3-188
CMPPS instruction, Vol.1-10-9
CMPS8b instruction, Vol.2-3-195
CMPSD-Compare Scalar Double-Precision Floating-Point Values, Vol.2-3-199
CMPSQ instruction, Vol.2-3-195
CMPSX instruction, Vol.2-3-203
CMPSW instruction, Vol.1-10-9
CMPSw instruction, Vol.2-3-195
CMPXCHG16B instruction, Vol.1-7-5, Vol.2-3-210
CPUID bit, Vol.2-3-244
CPUID flag, Vol.2-3-246
Code modules
16 bit vs. 32 bit, Vol.3-22-1
mixing 16-bit and 32-bit code, Vol.3-22-1
sharing data, mixed-size code segs, Vol.3-22-3
transferring control, mixed-size code segs, Vol.3-22-3
Code segment, Vol.1-3-14
Code segments
accessing data in, Vol.3-5-9
accessing through a call gate, Vol.3-5-15
description of, Vol.3-3-12
descriptor format, Vol.3-5-2
descriptor layout, Vol.3-5-2
direct calls or jumps to, Vol.3-5-10
paging of, Vol.3-2-6
pointer size, Vol.3-22-4
privilege level checks
transferring control between code segs, Vol.3-5-10
COMISD-Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS, Vol.2-3-213
COMISD instruction, Vol.1-11-7
COMISS-Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS, Vol.2-3-215
COMISS instruction, Vol.1-10-9
Compare
compare and exchange, Vol.1-7-4
integers, Vol.1-7-8
real numbers, x87 FPU, Vol.1-8-19
strings, Vol.1-7-18
Compatibility
IA-32 architecture, Vol.1-23-1
software, Vol.3-1-7, Vol.4-1-5
Compatibility mode
address space, Vol.1-3-1
branch functions, Vol.1-6-12
call gate descriptors, Vol.1-8-12
code segment descriptor, Vol.3-5-3
code segment descriptors, Vol.3-10-12
code segment selectors, Vol.3-2-13
CSL and CSO, Vol.3-10-12
direct calls or jumps to, Vol.3-5-4
descriptor layout, Vol.3-5-4
descriptor format, Vol.3-5-2
descriptor layout, Vol.3-5-2
descriptor format, Vol.3-5-2
descriptor layout, Vol.3-5-2
descriptor format, Vol.3-5-2
descriptor layout, Vol.3-5-2
memory management, Vol.3-2-6
memory models, Vol.1-3-9
MMX technology, Vol.1-9-2
operation, Vol.3-10-12
see 64-bit mode
gates, Vol.3-2-4
GDTR register, Vol.3-2-12, Vol.3-2-13
global and local descriptor tables, Vol.3-2-4
IDTR register, Vol.3-2-12
interrupt handling, Vol.3-2-5
L flag, Vol.3-3-12, Vol.3-5-4
memory management, Vol.3-2-6
memory models, Vol.1-3-9
MMX technology, Vol.1-9-2
operation, Vol.3-10-12
see 64-bit mode
segment loading instructions, Vol.3-3-9
segmentation, Vol.1-3-22
segments, Vol.3-3-5
SSE extensions, Vol.1-10-3
SSE2 extensions, Vol.1-11-3
SSE3 extensions, Vol.1-12-1
SSE3 extensions, Vol.1-12-1
summary table notation, Vol.2-3-9
switching to, Vol.3-10-12
SYSCALL and SYSEXIT, Vol.3-5-22
SYSEXENTER and SYSEXIT, Vol.3-5-21
system flags, Vol.3-2-11
system registers, Vol.3-2-7
task register, Vol.3-2-13
x87 FPU, Vol.1-8-1
See also: 64-bit mode, IA-32e mode
See also: IA-32e mode, 64-bit mode
Compatibility, software, Vol.1-1-6, Vol.2-1-6

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
INDEX
INDEX

Condition code flags, EFLAGS register, Vol.2-3-175
Condition code flags, x87 FPU status word
branching on, Vol.1-8-6
compatibility information, Vol.1-23-8
conditional moves on, Vol.1-8-6
description of, Vol.1-8-4
flags affected by instructions, Vol.2-3-14
interpretation of, Vol.1-8-5
use of, Vol.1-8-19
Conditional jump, Vol.2-3-548
Conditional moves, x87 FPU condition codes, Vol.1-8-6
Conforming code segment, Vol.2-3-582
Conforming code segments
accessing, Vol.3-5-12
C (conforming) flag, Vol.3-5-11
description of, Vol.3-3-13
Constants (floating point), Vol.1-8-17
Constants (floating point), loading, Vol.2-3-405
Context, task (see Task state)
Control registers
64-bit mode, Vol.1-3-5, Vol.3-2-13
CR0, Vol.3-2-13
CR1 (reserved), Vol.3-2-13
CR2, Vol.3-2-13
CR3 (PDBR), Vol.3-2-6, Vol.3-2-13
CR4, Vol.3-2-13
description of, Vol.3-2-13
introduction to, Vol.3-2-6
overview of, Vol.1-3-4
Control registers, moving values to and from, Vol.2-4-39
Coprocessor segment
overrun exception, Vol.3-6-35, Vol.1-23-12
Core Solo and Core Duo, Vol.1-2-4
Cosine, x87 FPU operation, Vol.1-8-20, Vol.2-3-381, Vol.2-3-433
Counter mask field
PerfEvtSel0 and PerfEvtSel1 MSRs (P6 family processors), Vol.3-20-5, Vol.3-20-135
CPL, Vol.2-3-166, Vol.2-5-152
description of, Vol.3-5-7
field, CS segment selector, Vol.3-5-2
CPUID instruction, Vol.2-3-217, Vol.2-3-246
36-bit page size extension, Vol.2-3-246
APIC on-chip, Vol.2-3-246
availability, Vol.1-23-5
basic CPUID information, Vol.2-3-218
cache and TLB characteristics, Vol.2-3-218
CLFLUSH flag, Vol.1-11-12, Vol.2-3-246
CLFLUSH instruction cache line size, Vol.2-3-242
CMOVc feature flag, Vol.1-7-3
CMPIXCHG16B flag, Vol.2-3-244
CMPIXCHGB flag, Vol.2-3-246
control register flags, Vol.3-2-20
core CPUID qualified debug store, Vol.2-3-243
debug extensions, CR4.DE, Vol.2-3-246
debug store supported, Vol.3-2-247
detecting features, Vol.1-23-2
determine support for, Vol.1-3-17
deterministic cache parameters leaf, Vol.2-3-218, Vol.2-3-221,
Vol.2-3-237
earlier processors, Vol.1-20-1
extended function information, Vol.2-3-238
feature information, Vol.2-3-245
FPU on-chip, Vol.2-3-246
FSAVE flag, Vol.2-3-247
FXSAVE-FXRSTOR flag, Vol.2-3-247
FXSAVE-FXRSTOR flag, Vol.1-10-14
IA-32e mode available, Vol.2-3-239
input limits for EAX, Vol.2-3-240
L1 Context ID, Vol.2-3-244
local APIC physical ID, Vol.2-3-242
machine check architecture, Vol.2-3-246
machine check exception, Vol.2-3-246
memory type range registers, Vol.2-3-246
MMX feature flag, Vol.1-9-8
MONITOR feature information, Vol.2-3-251
MONITOR/Mwait flag, Vol.2-3-243
MONITOR/Mwait leaf, Vol.2-3-219, Vol.2-3-220, Vol.2-3-224,
Mwait feature information, Vol.2-3-251
page attribute table, Vol.2-3-246
page size extension, Vol.2-3-246
performance monitoring features, Vol.2-3-252
physical address bits, Vol.2-3-240
physical address extension, Vol.2-3-246
power management, Vol.2-3-251, Vol.2-3-252, Vol.2-3-253,
Vol.2-3-254
processor brand index, Vol.2-3-242, Vol.2-3-254
processor brand string, Vol.2-3-239, Vol.2-3-254
processor identification, Vol.1-20-1
processor serial number, Vol.2-3-218, Vol.2-3-246
processor type field, Vol.2-3-241
RDMR flag, Vol.2-3-246
returned in EBX, Vol.2-3-242
returned in ECX & EDX, Vol.2-3-242
self snoop, Vol.2-3-247
serializing instructions, Vol.3-9-18
serializing use, Vol.1-19-5
SpeedStep technology, Vol.2-3-243
SS2 extensions flag, Vol.2-3-247
SSE extensions flag, Vol.2-3-247
SSE feature flag, Vol.1-10-1, Vol.1-10-6
SSE3 extensions flag, Vol.2-3-243
SSE3 feature flag, Vol.1-12-5
SSSE3 extensions flag, Vol.2-3-243
summary of, Vol.1-7-23
text format for data, Vol.3-1-9, Vol.4-1-7
SYSENTER flag, Vol.2-3-246
SYSEXIT flag, Vol.2-3-246
thermal management, Vol.2-3-251, Vol.2-3-252, Vol.2-3-253,
Vol.2-3-254
thermal monitor, Vol.2-3-243, Vol.2-3-247
time stamp counter, Vol.2-3-246
using CPUID, Vol.2-3-217
vendor ID string, Vol.2-3-240
version information, Vol.2-3-218, Vol.2-3-251
virtual 8086 Mode flag, Vol.2-3-246
virtual address bits, Vol.2-3-240
WMSR flag, Vol.2-3-246
CQ0 instruction, Vol.2-3-316
CR0 control register, Vol.2-4-647, Vol.1-23-8
description of, Vol.3-2-13
introduction to, Vol.3-2-6
state following processor reset, Vol.3-10-2
CR1 control register (reserved), Vol.3-2-13
CR2 control register
description of, Vol.3-2-13
introduction to, Vol.3-2-6
CR3 control register (PDBR)
associated with a task, Vol.3-8-1, Vol.3-8-3
description of, Vol.3-2-13
in TSS, Vol.3-8-4, Vol.3-8-17
introduction to, Vol.3-2-6
loading during initialization, Vol.3-10-10
memory management, Vol.3-2-6
page directory base address, Vol.3-2-6
page table base address, Vol.3-2-5
CR4 control register
INDEX

D

D (default operation size) flag
- segment descriptor, Vol.3-22-1, Vol.1-23-33
D (default operation size) flag, segment descriptor, Vol.2-4-397
D (default size) flag, segment descriptor, Vol.1-6-2, Vol.1-6-3
DAA instruction, Vol.1-7-9, Vol.2-3-317
DAS instruction, Vol.1-7-9, Vol.2-3-319
Data breakpoint exception conditions, Vol.3-18-10
Data movement instructions, Vol.1-7-2
Data pointer, x87 FPU, Vol.1-8-9
Data registers, x87 FPU, Vol.1-8-1
Data segment, Vol.1-3-14
Data segments
description of, Vol.3-3-12
descriptor layout, Vol.3-5-2
expand-down type, Vol.3-3-11
paging of, Vol.3-2-6
privilege level checking when accessing, Vol.3-5-8
Data types
128-bit packed SIMD, Vol.1-4-8
64-bit mode, Vol.1-7-2
64-bit packed SIMD, Vol.1-4-8
alignment, Vol.1-4-2
BCD integers, Vol.1-4-9, Vol.1-7-9
bit field, Vol.1-4-7
byte, Vol.1-4-1
doubleword, Vol.1-4-1
floating-point, Vol.1-4-4
fundamental, Vol.1-4-1
integers, Vol.1-4-3
numeric, Vol.1-4-2
operated on by GP instructions, Vol.1-7-1, Vol.1-7-2
operated on by MMX technology, Vol.1-9-3
operated on by SSE extensions, Vol.1-10-5
operated on by SSE2 extensions, Vol.1-11-3
operated on by x87 FPU, Vol.1-8-13
operated on in 64-bit mode, Vol.1-4-7
packed bytes, Vol.1-9-3
packed doublewords, Vol.1-9-3
packed SIMD, Vol.1-4-8
packed words, Vol.1-9-3
pointers, Vol.1-4-6
signed integers, Vol.1-4-4
strings, Vol.1-4-8
unsigned integers, Vol.1-4-3
word, Vol.1-4-1
DAZ (denormals-are-zeros) flag
MXCSR register, Vol.1-10-5
DE (debugging extensions) flag, CR4 control register, Vol.3-2-17,
DE (denormal operand exception) flag
MXCSR register, Vol.1-11-15
x87 FPU status word, Vol.1-8-5, Vol.1-8-28
Debug exception (NDB), Vol.3-6-7, Vol.3-6-25, Vol.3-8-5, Vol.3-18-7,
Vol.3-18-14, Vol.3-18-42
Debug registers
64-bit mode, Vol.1-3-5
legacy modes, Vol.1-3-4
Debug registers, moving value to and from, Vol.2-4-42
Debug store (see DS)

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 - Index -7
description of, Vol.1-3-15
flags affected by instructions, Vol.2-3-14
identifying 32-bit processors, Vol.1-23-6
instructions that operate on, Vol.1-7-21
introduction to, Vol.3-2-6
new flags, Vol.1-23-6
overview, Vol.1-3-11
part of basic programming environment, Vol.1-7-1
popping, Vol.2-4-401
popping on return from interrupt, Vol.2-3-539
pushing, Vol.2-4-521
pushing on interrupts, Vol.2-3-517
restoring from stack, Vol.1-6-7
saved in TSS, Vol.3-8-4
saving, Vol.2-4-590
saving on a procedure call, Vol.1-6-7
Vol.2-3-551, Vol.2-4-612, Vol.2-4-703
system flags, Vol.3-2-9
use with CMOVcc instructions, Vol.1-7-3
EIP register, Vol.2-3-140, Vol.2-3-517, Vol.2-3-539, Vol.2-3-554,
Vol.1-23-11
description of, Vol.1-3-18
overview, Vol.1-3-11
part of basic programming environment, Vol.1-7-1
relationship to CS register, Vol.1-3-14
saved in TSS, Vol.3-8-5
state following initialization, Vol.3-10-5
EM (emulation) flag
CR0 control register, Vol.3-2-16, Vol.3-6-32, Vol.3-10-6, Vol.3-10-7,
Vol.3-13-1, Vol.1-14-3
ENCODKEY128—Encode 128-Bit Key, Vol.2-3-342
ENCODKEY256—Encode 256-Bit Key, Vol.2-3-344
Encodings
See machine instructions, opcodes
ENDBR32—Terminate an Indirect Branch in 32-bit and Compatibility Mode
, Vol.2-3-346
Enhanced Intel Deeper Sleep, Vol.1-2-4
Enhanced Intel SpeedStep Technology
ACPI 3.0 specification, Vol.3-15-1
IA32_APERF MSR, Vol.3-15-2
IA32_MPERF MSR, Vol.3-15-2
IA32_PERF_CTL MSR, Vol.3-15-1
IA32_PERF_STATUS MSR, Vol.3-15-1
introduction, Vol.3-15-1
multiple processor cores, Vol.3-15-1
performance transitions, Vol.3-15-1
P-state coordination, Vol.3-15-1
See also: thermal monitoring
ENTRY instruction, Vol.1-6-20, Vol.1-7-21, Vol.2-3-354
EOI
End Of Interrupt register, Vol.3-11-38
Error code, Vol.3-13-7, Vol.3-17-7, Vol.3-17-10, Vol.3-17-13,
Vol.3-17-15
architectural MCA, Vol.3-17-1, Vol.3-17-3, Vol.3-17-7, Vol.3-17-10,
Vol.3-17-13, Vol.3-17-15
decoding IA32_MCI_STATUS, Vol.3-17-1, Vol.3-17-3, Vol.3-17-7,
Vol.3-17-10, Vol.3-17-13, Vol.3-17-15
exception, description of, Vol.3-6-17
external bus, Vol.3-17-1, Vol.3-17-3, Vol.3-17-7, Vol.3-17-10,
Vol.3-17-13, Vol.3-17-15
memory hierarchy, Vol.3-17-3, Vol.3-17-7, Vol.3-17-10, Vol.3-17-13,
Vol.3-17-15
pushing on stack, Vol.1-23-32
watchdog timer, Vol.3-17-1, Vol.3-17-3, Vol.3-17-7, Vol.3-17-10,
Vol.3-17-13, Vol.3-17-15
Error numbers
VM-instruction error field, Vol.3-31-31
Error signals, Vol.1-23-11
Error-reporting bank registers, Vol.3-16-2
ERROR# input, Vol.1-23-16
output, Vol.1-23-16
ES register, Vol.1-3-13, Vol.1-3-14, Vol.2-3-588, Vol.2-4-178,
Vol.2-4-604, Vol.2-4-665
ES (exception summary) flag
x87 FPU status word, Vol.1-8-31
ES0 and ES1 (event select) fields, CSER MSR (Pentium processor),
Vol.3-20-137
ESC instructions, x87 FPU, Vol.1-8-15
ESI register, Vol.1-3-11, Vol.1-3-12, Vol.2-3-195, Vol.2-3-612,
Vol.2-4-110, Vol.2-4-178, Vol.2-4-661
ESP register, Vol.1-3-12, Vol.2-3-140
ESP register (stack pointer), Vol.1-3-11, Vol.1-6-3
ESR
Error Status Register, Vol.3-11-39
ET (extension type) flag, CRO control register, Vol.3-2-15, Vol.1-23-8
Event select field, PerfEvtSel0 and PerfEvtSel1 MSRs (P6 family
processors), Vol.3-20-4, Vol.3-20-98, Vol.3-20-134
Events
at-retirement, Vol.3-20-117
at-retirement (Pentium 4 processor), Vol.3-20-106
non-retirement (Pentium 4 processor), Vol.3-20-106
ETEXR, Vol.2-3-5
Exception flags, x87 FPU status word, Vol.1-8-5
Exception handler
calling, Vol.3-6-11
defined, Vol.3-6-1
flag usage by handler procedure
machine-check exception handler, Vol.3-16-28
machine-check exceptions (#MC), Vol.3-16-28
machine-error logging utility, Vol.3-16-28
procedures, Vol.3-6-12
protection of handler procedures, Vol.3-6-16
task, Vol.3-6-17, Vol.3-8-2
Exception handlers
overview of, Vol.1-6-12
SIMD floating-point exceptions, Vol.1-D-1
SSE and SSE2 extensions, Vol.1-11-18
typical actions of a FP exception handler, Vol.1-4-24
x87 FPU, Vol.1-8-31
Exception priority, floating-point exceptions, Vol.1-4-23
Exception-flag masks, x87 FPU control word, Vol.1-6-7
Exceptions
64-bit mode, Vol.1-6-19
alignment check, Vol.1-23-11
BOUND range exceeded (#BR), Vol.2-3-123
classifications, Vol.3-6-5
compound error codes, Vol.3-16-21
conditions checked during a task switch, Vol.3-8-13
coprocessor segment overrun, Vol.1-23-12
description of, Vol.1-6-12, Vol.3-2-5, Vol.3-6-1
device not available, Vol.1-23-12
double fault, Vol.3-6-33
error code, Vol.3-6-17
execute-disable bit, Vol.3-5-32
floating-point error, Vol.1-23-12
general protection, Vol.1-23-12
handler, Vol.1-6-12
handler mechanism, Vol.3-6-12
handler procedures, Vol.3-6-12
handling, Vol.3-6-11
handling in real-address mode, Vol.3-21-4
handling in SMM, Vol.4-32-11
handling in virtual-8086 mode, Vol.3-21-11
handling through a task gate in virtual-8086 mode, Vol.3-21-14
handling through a trap or interrupt gate in virtual-8086 mode,
Vol.3-21-12
IA-32e mode, Vol.3-2-5
IDT, Vol.3-6-9
implicit call to handler, Vol.1-6-1
in real-address mode, Vol.1-6-18
initializing for protected-mode operation, Vol.3-10-10
invalid-opcode, Vol.1-23-5
masking debug exceptions, Vol.3-6-7
masking when switching stack segments, Vol.3-6-8
MCA error codes, Vol.3-16-20
MMX instructions, Vol.3-13-1
overflow exception (#OF), Vol.2-3-517
overview of, Vol.3-6-1
priority of, Vol.1-23-22
priority of, x87 FPU exceptions, Vol.1-23-10
reference information on all exceptions, Vol.3-6-23
reference information, 64-bit mode, Vol.3-6-19
restarting a task or program, Vol.3-6-5
returning from, Vol.2-3-539
segment not present, Vol.1-23-12
simple error codes, Vol.3-16-20
sources of, Vol.3-6-4
summary of, Vol.3-6-2
vectors, Vol.3-6-1
Executable, Vol.3-3-11
Execute-disable bit capability
conditional for, Vol.3-5-30
CPUID flag, Vol.3-5-30
detecting and enabling, Vol.3-5-30
exception handling, Vol.3-5-32
page-fault exceptions, Vol.3-6-44
protection matrix for IA-32e mode, Vol.3-5-31
protection matrix for legacy modes, Vol.3-5-31
reserved bit checking, Vol.3-5-31
GETSEC, Vol.1-6-3, Vol.1-6-5
Exit-reason numbers
VM entries & exits, Vol.1-C-1
Expand-down data segment type, Vol.3-3-11
Exponent, extracting from floating-point number, Vol.2-3-470
Exponent, floating-point number, Vol.1-4-11
Extended signature table, Vol.3-10-31
extended signature table, Vol.3-10-31
External bus errors, detected with machine-check architecture, Vol.3-16-27
Extract exponent and significand, x87 FPU operation, Vol.2-3-470
EXTRACTPS: Extract packed floating-point values, Vol.2-3-357
F
FABS instruction, Vol.1-8-17, Vol.2-3-361
FADD instruction, Vol.1-8-17, Vol.2-3-362
FADDP instruction, Vol.1-8-17, Vol.2-3-362
Family 06H, Vol.3-17-1
Family OFH, Vol.3-17-1
microcode update facilities, Vol.3-10-28
Far call
description of, Vol.1-6-4
operation, Vol.1-6-5
Far pointer
16-bit addressing, Vol.1-3-9
32-bit addressing, Vol.1-3-9
64-bit mode, Vol.1-4-7
description of, Vol.1-3-7, Vol.1-4-6
legacy modes, Vol.1-4-6
Far pointer, loading, Vol.2-3-588
Far return operation, Vol.1-6-5
Far return, RET instruction, Vol.2-4-558
Faults
description of, Vol.3-6-5
restarting a program or task after, Vol.3-6-5
FBLD instruction, Vol.1-8-16, Vol.2-3-365
FBSTP instruction, Vol.1-8-16, Vol.2-3-367
FCHS instruction, Vol.1-8-17, Vol.2-3-369
FCLEX instruction, Vol.2-3-371
FCLEX/FCNLEX instructions, Vol.1-8-5
FCOMPP instruction, Vol.1-8-6, Vol.1-8-18, Vol.2-3-375
FDECTSTP instruction, Vol.2-3-383
FDIVP instruction, Vol.1-8-17, Vol.2-3-384
FDIVR instruction, Vol.1-8-17, Vol.2-3-387
FDIVRP instruction, Vol.1-8-17, Vol.2-3-387
FE (fixed MTRRs enabled) flag, IA32_MTRR_DEF_TYPE MSR, Vol.3-12-23
Feature determination of, processor, Vol.1-23-2
information, processor, Vol.1-23-2
Feature determination of, processor, Vol.1-20-1
Feature information, processor, Vol.2-3-217
FFREE instruction, Vol.2-3-390
FIADD instruction, Vol.1-8-17, Vol.2-3-362
FICOM instruction, Vol.1-8-6, Vol.1-8-18, Vol.2-3-391
FICOMP instruction, Vol.1-8-6, Vol.1-8-18, Vol.2-3-391
FIDIV instruction, Vol.1-8-17, Vol.2-3-384
FIDIVR instruction, Vol.1-8-17, Vol.2-3-387
FILD instruction, Vol.1-8-16, Vol.2-3-393
FIMUL instruction, Vol.1-8-17, Vol.2-3-411
FINCSTP instruction, Vol.2-3-395
FINIT instruction, Vol.2-3-396
FIST instruction, Vol.1-8-16, Vol.2-3-396
FISTP instruction, Vol.1-8-16, Vol.2-3-396
FISPST instruction, Vol.1-8-16, Vol.2-3-396
FISTTP instruction, Vol.1-8-16, Vol.2-3-396
FISUB instruction, Vol.1-8-17, Vol.2-3-445
FISUBR instruction, Vol.1-8-17, Vol.2-3-448
FIX (fixed range registers supported) flag, IA32_MTRR_CAPS MSR, Vol.3-12-22
Fixed-range MTRRs
description of, Vol.3-12-23
Flags
Flat memory model, Vol.1-3-7, Vol.1-3-13
Flat segmentation model, Vol.3-3-3
FLD1 instruction, Vol.1-8-17, Vol.2-3-405
FLDSW instruction, Vol.1-8-23
FLD instruction, Vol.1-8-17, Vol.2-3-405
Floating point instructions
machine encodings, Vol.1-8-61
Floating-point data types
biasing constant, Vol.1-4-6
denormalized finite number, Vol.1-4-5
description of, Vol.1-4-4
double extended precision format, Vol.1-4-4, Vol.1-4-5
double precision format, Vol.1-4-4, Vol.1-4-5
ehalf precision format, Vol.1-4-5
infinities, Vol.1-4-5
normalized finite number, Vol.1-4-5
single precision format, Vol.1-4-4, Vol.1-4-5
SSE extensions, Vol.1-10-5
SSE2 extensions, Vol.1-11-3
storing in memory, Vol.1-4-6
x87 FPU, Vol.1-8-13
zonal, Vol.1-4-5
Floating-point error exception (#MF), Vol.1-23-12
Floating-point exception handlers
SSE and SSE2 extensions, Vol.1-11-18
typical actions, Vol.1-4-24
x87 FPU, Vol.1-8-31
Floating-point exceptions
denormal operand exception (#D), Vol.1-4-20, Vol.1-8-28,
Vol.1-1-21, Vol.1-1-21
exception conditions, Vol.1-4-20
inexact result (precision) exception (#P), Vol.1-4-22, Vol.1-8-30,
Vol.1-11-16, Vol.1-1-21
invalid operation (#I), Vol.1-23-14
invalid-operation exception (#IA), Vol.1-1-21
invalid-operation exception (#IS), Vol.1-1-21
invalid-operation exception (#I), Vol.1-1-21
numeric overflow exception (#O), Vol.1-4-21, Vol.1-8-28,
numeric overflow (#O), Vol.1-23-10
numeric underflow exception (#U), Vol.1-4-22, Vol.1-8-29,
Vol.1-11-16, Vol.1-1-21
numeric underflow (#U), Vol.1-23-10
saved CS and EIP values, Vol.1-23-11
SSE and SSE SIMD, Vol.2-3-16
summary of, Vol.1-4-19, Vol.1-1-21
typical handler actions, Vol.1-4-24
x87 FPU, Vol.2-3-16
Floating-point format
biased exponent, Vol.1-4-13
description of, Vol.1-8-13
description of, Vol.1-8-13
exponent, Vol.1-4-11
fraction, Vol.1-4-11
indefinite, Vol.1-4-5
QNaN floating-point indefinite, Vol.1-4-17
real number system, Vol.1-4-11
sign, Vol.1-4-11
significand, Vol.1-4-11
Floating-point numbers
defined, Vol.1-4-11
encoding, Vol.1-4-5
Flush
flushing, Vol.1-8-28
caches, Vol.1-2-3-352, Vol.1-6-3
TLB entry, Vol.1-2-3-354
Flush-to-zero
mode, Vol.1-10-4
FLUSH# pin, Vol.3-6-3
FMUL instruction, Vol.1-8-17, Vol.2-3-411
FMULP instruction, Vol.1-8-17, Vol.2-3-411
FNICLX instruction, Vol.2-3-371
FNIINX instruction, Vol.2-3-396
FNOP instruction, Vol.1-8-23, Vol.2-3-414
FNSAVE instruction, Vol.1-2-3-426, Vol.3-13-4
FNSTCW instruction, Vol.2-3-439
FNSTENV instruction, Vol.2-3-409, Vol.2-3-441
FNSTSW instruction, Vol.2-3-443
Focus processor, local APC, Vol.3-11-26
Fopcode compatibility mode, Vol.1-8-10
FORCPR# interrupt enable bit, Vol.3-15-44
FPATAN instruction, Vol.1-8-20, Vol.1-8-21, Vol.2-3-415,
Vol.1-23-13
Fraction, floating-point number, Vol.1-4-11
FRNDINT instruction, Vol.1-8-18, Vol.2-3-423
FS register, Vol.1-3-13, Vol.1-3-14, Vol.2-3-588
FSRT instruction, Vol.1-8-16, Vol.2-3-437
FSFW instruction, Vol.2-3-439
FSTENV instruction, Vol.2-3-441, Vol.3-13-3
FSTENV/FNSTENV instructions, Vol.1-8-4, Vol.1-8-9, Vol.1-8-11,
Vol.1-8-23, Vol.1-23-14
FSTP instruction, Vol.1-8-16, Vol.2-3-437
FSTSW instruction, Vol.2-3-443
FSUB instruction, Vol.1-8-17, Vol.2-3-445
FSUBP instruction, Vol.1-8-17, Vol.2-3-445
FSUBRP instruction, Vol.1-8-17, Vol.2-3-448
FTAN instruction, Vol.1-8-18, Vol.2-3-455
FXCH instruction, Vol.1-8-16, Vol.2-3-457
Vol.2-3-458, Vol.3-2-18, Vol.3-2-19, Vol.3-10-8, Vol.3-13-3,
CPUID flag, Vol.2-3-247
Vol.1-6-63, Vol.1-6-66, Vol.3-2-18, Vol.3-2-19, Vol.3-10-8,
CPUID flag, Vol.2-3-247
FXSR feature flag, CPUID instruction, Vol.3-10-8
Vol.1-23-14
FYL2X instruction, Vol.1-8-21, Vol.2-3-472
FYL2X instruction, Vol.1-8-21, Vol.2-3-474
G
G (global) flag
page-directory entries, Vol.3-12-13
page-table entries, Vol.3-12-13
G (granularity) flag
segment descriptor, Vol.3-3-10, Vol.3-3-11, Vol.3-5-2, Vol.3-5-4
GO-3 (global breakpoint enable) flags
DR7 register, Vol.3-18-5
Gate descriptors
call gates, Vol.3-5-13
description of, Vol.3-5-13
IA-32e mode, Vol.3-5-14
Gates, Vol.3-2-4
IA-32e mode, Vol.3-2-4
GD (general detect enable) flag
DR7 register, Vol.3-18-5, Vol.3-18-10
GD
description of, Vol.3-2-3, Vol.3-3-15
IA-32e mode, Vol.3-2-4
index field of segment selector, Vol.3-3-7
initializing, Vol.3-10-10

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 Index -11
INDEX

paging of, Vol.3-2-6
pointers to exception/interrupt handlers, Vol.3-6-12
segment descriptors in, Vol.3-3-9
selecting with Ti flag of segment selector, Vol.3-3-7
task switching, Vol.3-8-9
task-gate descriptor, Vol.3-8-8
TSS descriptors, Vol.3-8-5
use in address translation, Vol.3-3-6
GDT (global descriptor table), Vol.2-3-600, Vol.2-3-603
GDTR register, Vol.1-3-4, Vol.1-3-6
description of, Vol.3-2-3, Vol.3-2-6, Vol.3-2-12, Vol.3-3-15
IA-32e mode, Vol.3-2-4, Vol.3-2-12
limit, Vol.3-6-5
loading during initialization, Vol.3-10-10
storing, Vol.3-3-15
GDTR (global descriptor table register), Vol.2-3-600, Vol.2-4-617
GE (global exact breakpoint enable) flag
DR7 register, Vol.3-18-5, Vol.3-18-10
General purpose registers
64-bit mode, Vol.1-3-5, Vol.1-3-13
description of, Vol.1-3-11
overview of, Vol.1-3-2, Vol.1-3-5
parameter passing, Vol.1-6-7
part of basic programming environment, Vol.1-7-1
using REX prefix, Vol.1-3-13
General-detect exception condition, Vol.3-18-10
General-protection exception (#GP)
IA-32e mode, Vol.3-2-4, Vol.3-2-12
limit, Vol.3-6-5
General-purpose instructions
64-bit encodings, Vol.1-8-18
64-bit mode, Vol.1-7-1
basic programming environment, Vol.1-7-1
data types operated on, Vol.1-7-1, Vol.1-7-2
description of, Vol.1-7-1
non-64-bit encodings, Vol.1-8-7
origin of, Vol.1-7-1
using REX prefix, Vol.1-3-13
programming with, Vol.1-7-1
summary of, Vol.1-5-6, Vol.1-7-2
General-purpose registers
moving value to and from, Vol.2-4-36
popping all, Vol.2-4-397
pushing all, Vol.2-4-519
General-purpose registers, saved in TSS, Vol.3-8-4
Global control MSRs, Vol.3-16-2
Global descriptor table register (see GDTR)
Global descriptor table (see GDT)
GS register, Vol.1-3-13, Vol.1-3-14, Vol.2-3-588

H

HALT state
relationship to SMI interrupt, Vol.4-32-3, Vol.4-32-14
Hardware Lock Elision (HLE), Vol.1-16-2
Hardware reset
description of, Vol.3-10-1
processor state after reset, Vol.3-10-2
state of MTRRs following, Vol.3-12-20
value of SMBASE following, Vol.4-32-4
Hexadecimal numbers, Vol.1-1-7, Vol.2-1-7, Vol.3-1-8, Vol.4-4-1-6
HITMR line, Vol.3-12-6
HLT instruction, Vol.2-3-189, Vol.3-2-26, Vol.3-5-24, Vol.3-6-34, Vol.3-6-26, Vol.4-32-14
Horizontal processing model, Vol.1-12-1
HSUBPS instruction, Vol.1-5-23, Vol.1-12-4, Vol.2-3-495
HT Technology

first processor, Vol.1-2-3
implementing, Vol.1-2-17
introduction, Vol.1-2-16
Hyper-Threading Technology
architectural state of a logical processor, Vol.3-9-33
description of, Vol.3-9-27
architectures, Vol.3-9-31
executing multiple threads, Vol.3-9-27
execution-based timing loops, Vol.3-9-54
external signal compatibility, Vol.3-9-52
halting logical processors, Vol.3-9-53
handling interrupts, Vol.3-9-27
HLT instruction, Vol.3-9-48
IA32_MISC_ENABLE MSR, Vol.3-9-30, Vol.3-9-33
initializing IA-32 processors with, Vol.3-9-26
local a, Vol.3-9-28
local APIC
functionality in logical processor, Vol.3-9-29
logical processors, identifying, Vol.3-9-39
machine check architecture, Vol.3-9-29
managing idle and blocked conditions, Vol.3-9-48
mapping resources, Vol.3-9-34
memory ordering, Vol.3-9-30
microcode update resources, Vol.3-9-30, Vol.3-9-33, Vol.3-10-35
MP systems, Vol.3-9-27
MTRRs, Vol.3-9-28, Vol.3-9-33
multi-threading feature flag, Vol.3-9-25
multi-threading support, Vol.3-9-29
PAT, Vol.3-9-29
PAUSE instruction, Vol.3-9-48, Vol.3-9-49
performance monitoring, Vol.3-20-120, Vol.3-20-125
performance monitoring counters, Vol.3-9-30, Vol.3-9-33
placement of locks and semaphores, Vol.3-9-54
required operating system support, Vol.3-9-51
scheduling multiple threads, Vol.3-9-54
self modifying code, Vol.3-9-31
serializing instructions, Vol.3-9-30
spin-wait loops
PAUSE instructions in, Vol.3-9-51, Vol.3-9-53
thermal monitor, Vol.3-9-32
TLBs, Vol.3-9-31

I

IA-32 architecture
history of, Vol.1-2-1
introduction to, Vol.1-2-1
IA-32 Intel architecture
compatibility, Vol.1-23-1
processors, Vol.1-23-1
IA32e mode
registers and mode changes, Vol.3-10-12
IA-32e mode
call gates, Vol.3-5-14
code segment descriptor, Vol.3-5-3
CPUID flag, Vol.2-3-239
D flag, Vol.3-5-4
data structures and initialization, Vol.3-10-11
debug registers, Vol.3-2-7
debug store area
descorators, Vol.3-2-4
DPL field, Vol.3-5-4
descriptors, Vol.3-10-12
data structure and initialization, Vol.3-2-7
gates, Vol.3-2-4
global and local descriptor tables, Vol.3-2-4
IA32_EFER MSR, Vol.3-2-7, Vol.3-5-30
initialization process, Vol.3-10-11
validation bit, Vol.3-15-43
IA32_TIME_STAMP_COUNTER MSR, Vol.4-2-453
ICR
Interrupt Command Register, Vol.3-11-38, Vol.3-11-41, Vol.3-11-47
ID (identification) flag
EFLAGS register, Vol.3-2-11, Vol.1-23-6
ID (identification) flag, EFLAGS register, Vol.1-3-17
IDIV instruction
Vol.3-10-9, Vol.3-10-10
IDT (interrupt descriptor table), Vol.2-3-518, Vol.2-3-600
IDTR register, Vol.1-3-4, Vol.1-3-6
description of, Vol.3-2-12, Vol.3-6-9
IA-32e mode, Vol.3-2-12
introduction to, Vol.3-2-5
limit, Vol.3-2-5
loading in real-address mode, Vol.3-21-5
storing, Vol.3-3-16
IDTR (interrupt descriptor table register), Vol.2-3-600, Vol.2-4-643
IE (invalid operation exception) flag
MXCSR register, Vol.1-11-14
IF (interrupt enable) flag
IF (interrupt enable) flag, EFLAGS register, Vol.2-3-166, Vol.2-4-662
IM (invalid operation exception) mask bit
MXCSR register, Vol.1-11-14
x87 FPU control word, Vol.1-8-7
Immediate operands, Vol.1-3-20, Vol.2-2-3
IMUL instruction, Vol.1-7-9, Vol.2-3-501
Indefinite
description of, Vol.1-4-17, Vol.1-4-18
floating-point format, Vol.1-4-5, Vol.1-4-13
integer, Vol.1-4-4, Vol.1-8-14
packed BCD integer, Vol.1-4-11
QNaN floating-point, Vol.1-11-7
Index field, segment selector, Vol.3-3-7
Inexact result (precision)exception (#P), overview, Vol.1-4-22
exception (#P), SSE-SSE2 extensions, Vol.1-11-16
exception (#P), x87 FPU, Vol.1-8-30
on floating-point operations, Vol.1-4-18
Infinity control flag, x87 FPU control word, Vol.1-8-8
Infinity, floating-point format, Vol.1-4-5, Vol.1-4-15
INIT interrupt, Vol.3-11-3
INIT pin, Vol.1-3-15
initial-count register, local APIC, Vol.3-11-16
Initialization
built-in self-test (BIST), Vol.3-10-1, Vol.3-10-5
CS register state following, Vol.3-10-5
EIP register state following, Vol.3-10-5
equipment, Vol.3-10-14
first instruction executed, Vol.3-10-5
hardware reset, Vol.3-10-1
IA-32e mode, Vol.3-10-11
IDT, protected mode, Vol.3-10-10
IDT, real-address mode, Vol.3-10-8
Intel486 SX processor and Intel 487 SX math coprocessor, Vol.1-23-16
location of software-initialization code, Vol.3-10-5
machine-check initialization, Vol.3-16-19
model and stepping information, Vol.3-10-5
multitasking environment, Vol.3-10-10, Vol.3-10-11
overview, Vol.3-10-1
paging, Vol.3-10-10
processor state after reset, Vol.3-10-2
protected mode, Vol.3-10-9
real-address mode, Vol.3-10-8
RESET# pin, Vol.3-10-1
setting up exception- and interrupt-handling facilities, Vol.3-10-10
x87 FPU, Vol.3-10-5
Initialization x87 FPU, Vol.2-3-396
INIT# pin, Vol.3-6-3, Vol.3-10-1
INIT# signal, Vol.3-2-26, Vol.3-24-4
Input/output (see I/O)
volatile storage, Vol.3-2-11, Vol.3-2-12, Vol.3-6-9
IA-32e mode, Vol.3-2-12
introduction to, Vol.3-2-5
limit, Vol.3-2-5
loading in real-address mode, Vol.3-21-5
storing, Vol.3-3-16
IDTR (interrupt descriptor table register), Vol.2-3-600, Vol.2-4-643
Index-14 Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
IRETD instruction, Vol.2-3-539, Vol.3-2-10, Vol.3-9-18

IRI
IRR (interrupt request register), local APIC, Vol.3-11-30

ISR
In Service Register, Vol.3-11-38, Vol.3-11-41, Vol.3-11-47

I/O
address space, Vol.1-19-1
breakpoint exception conditions, Vol.3-18-10
in virtual-8086 mode, Vol.3-21-10
instruction restart flag
SMM revision identifier field, Vol.4-32-15
instruction restart flag, SMM revision identifier field, Vol.4-32-15
IO_SM bit, Vol.4-32-12
I/O permission bit map, TSS, Vol.3-8-5
I/O privilege level (see IOPL)
map base, Vol.1-19-4
map base address field, TSS, Vol.3-8-5
permission bit map, Vol.1-19-4
restarting following SMI interrupt, Vol.4-32-15
saving I/O state
I/O state, Vol.4-32-12
sensitive instructions, Vol.1-19-3
SMM state save map, Vol.4-32-12

I/O APIC, Vol.3-11-26
bus arbitration, Vol.3-11-26
description of, Vol.3-11-1
external interrupts, Vol.3-6-3
information about, Vol.3-11-1
interrupt sources, Vol.3-11-2
local APIC and I/O APIC, Vol.3-11-2, Vol.3-11-3
overview of, Vol.3-11-1
valid interrupts, Vol.3-11-15
See also: local APIC

J
J-bit, Vol.1-4-11
Jump operation, Vol.2-3-553

K
KEN# pin, Vol.3-12-13, Vol.1-23-36

L
LD-L3 (local breakpoint enable) flags
DR7 register, Vol.3-18-5
L1 Context ID, Vol.2-3-244
L1 (level 1) cache
Vol.3-12-6
Vol.3-12-18
caching methods, Vol.3-12-6
CPUID feature flag, Vol.3-12-18
description of, Vol.3-12-4
effect of using write-through memory, Vol.3-12-9
introduction of, Vol.1-23-30
invalidating and flushing, Vol.3-12-9
MESI cache protocol, Vol.3-12-9
shared and adaptive mode, Vol.3-12-18
L2 (level 2) cache
Vol.3-12-6
caching methods, Vol.3-12-6
description of, Vol.3-12-4
validating and flushing, Vol.3-12-18
effect of using write-through memory, Vol.3-12-9
LOADWKEY—Load Internal Wrapping Key, Vol.2-3-607

LMSW instruction

LLDT instruction

LINT pins

Limit field, segment descriptor, Vol.3-5-2, Vol.3-5-4

Linear address, Vol.1-3-7
description of, Vol.3-3-6
IA-32e mode, Vol.3-3-7
introduction to, Vol.3-2-6
Linear address space, Vol.3-3-6
defined, Vol.1-3-7, Vol.3-3-1
maximum size, Vol.1-3-7
of task, Vol.3-8-17

Link (to previous task) field, TSS, Vol.3-6-17

Linking tasks

mechanism, Vol.3-8-15
modifying task linkages, Vol.3-8-16

LINT pins

function of, Vol.3-6-2

LLDT instruction, Vol.3-2-603, Vol.3-2-24, Vol.3-5-23, Vol.3-9-18


Load effective address operation, Vol.2-3-595

LOADIwK—Load Internal Wrapping Key, Vol.2-3-607

Local APIC, Vol.3-11-38
64-bit mode, Vol.3-11-32
APIC_ID value, Vol.3-9-34
arbitration over the APIC bus, Vol.3-11-26
arbitration over the system bus, Vol.3-11-26
block diagram, Vol.3-11-4
crash model, Vol.3-11-24
CRB usage, Vol.3-11-32
current-count register, Vol.3-11-16
description of, Vol.3-11-1
detecting with CPUID, Vol.3-11-7
DFR (destination format register), Vol.3-11-24
divide configuration register, Vol.3-11-17
enabling and disabling, Vol.3-11-7
external interrupts, Vol.3-6-2
features
    Pentium 4 and Intel Xeon, Vol.1-23-28
    Pentium and P6, Vol.1-23-28
focus processor, Vol.3-11-26
global enable flag, Vol.3-11-8
IA32_APIC_BASE MSR, Vol.3-11-8
initial-count register, Vol.3-11-16
internal error interrupts, Vol.3-11-1
interrupt command register (ICR), Vol.3-11-19
interrupt destination, Vol.3-11-26
interrupt distribution mechanism, Vol.3-11-25
interrupt sources, Vol.3-11-2
IRR (interrupt request register), Vol.3-11-30
I/O APIC, Vol.3-11-1
local APIC and 82489DX, Vol.1-23-28
local APIC and I/O APIC, Vol.3-11-2, Vol.3-11-3
local vector table (LVT), Vol.3-11-12
logical destination mode, Vol.3-11-23
LVT (local APIC version register), Vol.3-11-11
mapping of resources, Vol.3-9-34
MDA (message destination address), Vol.3-11-23
overview of, Vol.3-11-1
performance-monitoring counter, Vol.3-20-136
physical destination mode, Vol.3-11-23
receiving external interrupts, Vol.3-6-2
register address map, Vol.3-11-16, Vol.3-11-38
shared resources, Vol.3-9-34
SMM interrupt, Vol.4-32-2
spurious interrupt, Vol.3-11-32

spurious-interrupt vector register, Vol.3-11-8
state after a software (INIT) reset, Vol.3-11-10
state after INIT-deassert message, Vol.3-11-11
state after power-up reset, Vol.3-11-10
state of, Vol.3-11-33

SVR (spurious-interrupt vector register), Vol.3-11-8
timer, Vol.3-11-16
timer generated interrupts, Vol.3-11-1
tMR (trigger mode register), Vol.3-11-30
valid interrupts, Vol.3-11-15
version register, Vol.3-11-11

Local descriptor table register (see LDTR)
Local descriptor table (see LDT)
Local vector table (LVT)
description of, Vol.3-11-12
thermal entry, Vol.3-15-39
Local x2APIC, Vol.3-11-31, Vol.3-11-41, Vol.3-11-46
Local xAPIC ID, Vol.3-11-41

LOCK prefix, Vol.2-3-28, Vol.2-3-33, Vol.2-3-78, Vol.2-3-132,
Vol.2-3-134, Vol.2-3-136, Vol.2-3-208, Vol.2-3-321,
Vol.2-3-507, Vol.2-3-610, Vol.2-4-163, Vol.2-4-166,
Vol.2-4-168, Vol.2-4-602, Vol.2-4-674, Vol.1-6-23,
Vol.1-6-28, Vol.1-6-36, Vol.3-2-26, Vol.3-6-31, Vol.3-9-1,
LOCK signal, Vol.1-7-4

Locked (atomic) operations
automatic bus locking, Vol.3-9-3
bus locking, Vol.3-9-3
effects on caches, Vol.3-9-6
loading a segment descriptor, Vol.1-23-20
on IA-32 processors, Vol.1-23-35
overview of, Vol.3-9-1
software-controlled bus locking, Vol.3-9-4
Locking operation, Vol.2-3-610

LOCK# signal, Vol.3-2-26, Vol.3-9-1, Vol.3-9-3, Vol.3-9-4,
Vol.3-9-6
LODS instruction, Vol.1-3-17, Vol.1-7-18, Vol.2-3-612,
Vol.2-4-555

LODSB instruction, Vol.2-3-612

LODSQ instruction, Vol.2-3-612

LODSW instruction, Vol.2-3-612

Log epsilon, x87 FPU operation

Log (base 2), x87 FPU operation

Logical x2APIC ID, Vol.3-11-41
Logical processors
per physical package, Vol.3-9-25
Logical x2APIC ID, Vol.3-11-46

LOOP instructions

LOOP instruction, Vol.1-7-16, Vol.2-3-615

LOOPcc instructions, Vol.1-3-17, Vol.1-7-16, Vol.2-3-615

low-temperature interrupt enable bit


LSL instruction

Logical x2APIC ID, Vol.3-11-41
Logical processors
per physical package, Vol.3-9-25
Logical x2APIC ID, Vol.3-11-46

LOOP instructions

LOOP instruction, Vol.1-7-16, Vol.2-3-615

LOOPcc instructions, Vol.1-3-17, Vol.1-7-16, Vol.2-3-615

low-temperature interrupt enable bit


LSL instruction, Vol.2-3-618, Vol.3-2-25, Vol.3-5-25

LSS instruction, Vol.1-7-23, Vol.2-3-588, Vol.3-3-8, Vol.3-5-8

LTR instruction, Vol.2-3-621, Vol.2-3-24, Vol.3-5-24, Vol.3-8-7,
Vol.3-9-18, Vol.3-10-11

LVT (see Local vector table)

LZCNT - Count the Number of Leading Zero Bits
Vol.2-3-623

Machine check architecture
CPUID flag, Vol.2-3-246
description, Vol.2-3-246

Machine check registers, Vol.1-3-4

Machine instructions
64-bit mode, Vol.1-B-1
classification (tttn) field, Vol.1-B-6
direction bit (d) field, Vol.1-B-6
floating-point instruction encodings, Vol.1-B-61
general description, Vol.1-B-1
general-purpose encodings, Vol.1-B-7–Vol.1-B-37
legacy prefixes, Vol.1-B-1
opcode fields, Vol.1-B-2
operand size (w) bit, Vol.1-B-4
P6 family encodings, Vol.1-B-41
Pentium processor family encodings, Vol.1-B-37
reg (reg) field, Vol.1-B-3, Vol.1-B-4
RED prefixes, Vol.1-B-2
segment register (sreg) field, Vol.1-B-5
sign-extend (s) bit, Vol.1-B-5
SIMD 64-bit encodings, Vol.1-B-37
special 64-bit encodings, Vol.1-B-61
special fields, Vol.1-B-2
special-purpose register (see) field, Vol.1-B-5
SSE3 encodings, Vol.1-B-56–Vol.1-B-60
VMX encodings, Vol.1-B-112, Vol.1-B-113
See also: opcodes
Machine status word, CR0 register
Machine-check architecture
availability of MCA and exception, Vol.3-16-19
compatibility with Pentium processor, Vol.3-16-1
compound error codes, Vol.3-16-21
CPUID flags, Vol.3-16-19
error codes, Vol.3-16-20, Vol.3-16-21
error-reporting bank registers, Vol.3-16-22
error-reporting MSRs, Vol.3-16-5
extended machine check state MSRs, Vol.3-16-12
external bus errors, Vol.3-16-27
first introduced, Vol.1-23-22
global MSRs, Vol.3-16-2
initialization of, Vol.3-16-19
logging correctable errors, Vol.3-16-29, Vol.3-16-31, Vol.3-16-35
machine-check exception handler, Vol.3-16-28
machine-check exception (#MC), Vol.3-16-1
MSRs, Vol.3-16-2
overview of MCA, Vol.3-16-1
Pentium processor exception handling, Vol.3-16-29
Pentium processor style error reporting, Vol.3-16-13
simple error codes, Vol.3-16-20
writing machine-check software, Vol.3-16-27, Vol.3-16-28
Machine-check exception (#MC), Vol.3-6-62, Vol.3-16-1, Vol.3-16-19,
Mapping of shared resources, Vol.3-9-34
Maskable hardware interrupts
description of, Vol.3-6-3
handling with virtual interrupt mechanism, Vol.3-21-15
masking, Vol.3-2-10, Vol.3-6-7
Maskable interrupts, Vol.1-6-13
Masked responses
denormal operand exception (#D), Vol.1-4-20, Vol.1-8-28
divide by zero exception (#Z), Vol.1-4-21, Vol.1-8-28
inexact result (precision) exception (#P), Vol.1-4-23, Vol.1-8-30
invalid arithmetic operation (#IA), Vol.1-8-27
invalid operation exception (#I), Vol.1-4-20
numeric overflow exception (#O), Vol.1-4-21, Vol.1-8-29
numeric underflow exception (#U), Vol.1-4-22, Vol.1-8-29
stack overflow or underflow exception (#IS), Vol.1-8-27
Marks, exception-flags
MXCSR register, Vol.1-10-4
x87 FPU control word, Vol.1-8-7
MAXPD instruction, Vol.1-11-6
MAXPS- Maximum of Packed Double-Precision Floating-Point Values,
Vol.2-4-12
MAXPS instruction, Vol.1-10-8
MAXPS- Maximum of Packed Single-Precision Floating-Point Values,
Vol.2-4-15
MAXSD instruction, Vol.1-11-6
MAXSD- Return Maximum Scalar Double-Precision Floating-Point Value,
Vol.1-15-3, Vol.2-4-18
MAXSS instruction, Vol.1-10-9
MAXSS- Return Maximum Scalar Single-Precision Floating-Point Value,
Vol.2-4-20
MCA flag, CPUID instruction, Vol.3-16-19
MCE flag, CPUID instruction, Vol.3-16-19
MCE (machine-check enable) flag
MDA (message destination address)
local APIC, Vol.3-11-23
measured environment, Vol.1-6-1
Measured Launched Environment, Vol.1-6-1, Vol.1-6-25
Memory, Vol.3-12-1
flat memory model, Vol.3-1-3
management registers, Vol.1-3-4
memory type range registers (MTRRs), Vol.1-3-4
modes of operation, Vol.1-3-9
organization, Vol.1-3-6, Vol.1-3-7
physical, Vol.1-3-6
real address mode memory model, Vol.1-3-7, Vol.1-3-8
segmented memory model, Vol.1-3-7
virtual-8086 mode memory model, Vol.1-3-7, Vol.1-3-8
Memory management
introduction to, Vol.3-2-6
overview, Vol.3-3-1
paging, Vol.3-3-1, Vol.3-3-2
registers, Vol.3-2-11
segments, Vol.3-3-1, Vol.3-3-2, Vol.3-3-7
Memory operands
64-bit mode, Vol.1-3-21
legacy modes, Vol.1-3-21
Memory ordering
in IA-32 processors, Vol.1-23-34
overview, Vol.3-9-6
processor ordering, Vol.3-9-6
strengthening or weakening, Vol.3-9-16
write ordering, Vol.3-9-6
Memory type range registers (see MTRRs)
Memory types
caching methods, defined, Vol.3-12-6
choosing, Vol.3-12-8
MTRR types, Vol.3-12-21
selecting for Pentium III and Pentium 4 processors, Vol.3-12-15
selecting for Pentium Pro and Pentium II processors, Vol.3-12-14
UC (strong uncacheable), Vol.3-12-6
UC (uncacheable), Vol.3-12-6
WB (write back), Vol.3-12-7
WC (write combining), Vol.3-12-7
WP (write protected), Vol.3-12-7
writing values across pages with different memory types, Vol.3-12-16
wT (write through), Vol.3-12-7
MemTypeGet() function, Vol.3-12-29
MemTypeSet() function, Vol.3-12-31
MESI cache protocol, Vol.3-12-5, Vol.3-12-9
Message address register, Vol.3-11-34
Message data register format, Vol.3-11-35
Message signalled interrupts
message address register, Vol.3-11-34
message data register format, Vol.3-11-34
Vol.3-9-7, Vol.3-9-16, Vol.3-9-17, Vol.3-9-18
Microarchitecture
(see Intel NetBurst microarchitecture)
(see P6 family microarchitecture)
Microcode update facilities
authenticating an update, Vol.3-10-37
BIOS responsibilities, Vol.3-10-38

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 Index -19
INDEX

calling program responsibilities, Vol.3-10-39
checksum, Vol.3-10-33
extended signature table, Vol.3-10-31
family OFH processors, Vol.3-10-28
field definitions, Vol.3-10-28
format of update, Vol.3-10-28
function 0DH presence test, Vol.3-10-42
function 01H write microcode update data, Vol.3-10-43
function 02H microcode update control, Vol.3-10-46
function 03H read microcode update data, Vol.3-10-47
general description, Vol.3-10-28
HT Technology, Vol.3-10-35
INT 15H-based interface, Vol.3-10-42
overview, Vol.3-10-27
process description, Vol.3-10-28
processor identification, Vol.3-10-32
processor signature, Vol.3-10-32
return codes, Vol.3-10-48
update loader, Vol.3-10-34
update signature and verification, Vol.3-10-36
update specifications, Vol.3-10-37
VMX non-root operation, Vol.3-26-12
MINPD instruction, Vol.1-11-6
MINPD- Minimum of Packed Double-Precision Floating-Point Values, Vol.2-4-23
MINPS instruction, Vol.1-10-9
MINPS- Minimum of Packed Single-Precision Floating-Point Values, Vol.2-4-26
MINSD instruction, Vol.1-11-7
MINSD- Return Minimum Scalar Double-Precision Floating-Point Value, Vol.2-4-29
MINSS instruction, Vol.1-10-9
MINSS- Return Minimum Scalar Single-Precision Floating-Point Value, Vol.2-4-31
Mixing 16-bit and 32-bit code in IA-32 processors, Vol.1-23-33
overview, Vol.3-22-1
MLE, Vol.1-6-1
MMX instruction set
arithmetic instructions, Vol.1-9-6
comparison instructions, Vol.1-9-7
conversion instructions, Vol.1-9-8
data transfer instructions, Vol.1-9-6
EMMS instruction, Vol.1-9-8
logical instructions, Vol.1-9-7
overview, Vol.1-9-5
shift instructions, Vol.1-9-9
MMX instructions
CPUID flag for technology, Vol.2-3-247
encodings, Vol.1-8-38
MMX registers
description of, Vol.1-9-2
overview of, Vol.1-3-2
MMX technology
64-bit mode, Vol.1-9-2
64-bit packed SIMD data types, Vol.1-4-8
compatibility mode, Vol.1-9-2
compatibility with FPU architecture, Vol.1-9-8
data types, Vol.1-9-3
debugging MMX code, Vol.3-13-5
detecting MMX technology with CPUID instruction, Vol.1-9-8
effect of instruction prefixes on MMX instructions, Vol.1-9-9
effect of MMX instructions on pending x87 floating-point exceptions, Vol.3-13-5
emulation of the MMX instruction set, Vol.3-13-1
exception handling in MMX code, Vol.1-9-11
exceptions that can occur when executing MMX instructions, Vol.3-13-1
IA-32e mode, Vol.1-9-2
interfacing with MMX code, Vol.1-9-10
introduction of into the IA-32 architecture, Vol.1-23-2
introduction to, Vol.1-9-1
memory data formats, Vol.1-9-3
mixing MMX and floating-point instructions, Vol.1-9-10
MMX registers, Vol.1-9-2
programming environment (overview), Vol.1-9-1
register aliasing, Vol.1-9-1
register mapping, Vol.1-9-11
saturation arithmetic, Vol.1-9-4
SIMD execution environment, Vol.1-9-4
state, Vol.3-13-1
state, saving and restoring, Vol.3-13-3
system programming, Vol.3-13-1
task or context switches, Vol.3-13-4
transitions between x87 FPU - MMX code, Vol.1-9-9
updating MMX technology routines using 128-bit SIMD integer instructions, Vol.1-11-24
using MMX code in a multitasking operating system environment, Vol.1-9-10
using the EMMS instruction, Vol.1-9-9
using TS flag to control saving of MMX state, Vol.1-14-7
wraparound mode, Vol.1-9-4
Mod field, instruction format, Vol.2-2-3
Mode switching
element, Vol.3-10-14
real-address and protected mode, Vol.3-10-13
to SMM, Vol.4-32-2
Model and stepping information, following processor initialization or reset, Vol.3-10-5
Model & family information, Vol.2-3-251
Model-specific registers (see MSRs)
Modes of operation
64-bit mode, Vol.1-3-1
compatibility mode, Vol.1-3-1
memory models used with, Vol.1-3-9
overview, Vol.1-3-1, Vol.1-3-5
protected mode, Vol.1-3-1
real-address mode, Vol.1-3-1
Modes of operation (see Operating modes)
ModRM byte, Vol.2-2-3
16-bit addressing forms, Vol.2-2-5
32-bit addressing forms of, Vol.2-2-6
description of, Vol.2-2-3
MONITOR instruction, Vol.1-5-24, Vol.1-12-6, Vol.2-4-33, Vol.3-26-3
CPUID flag, Vol.2-3-243
feature data, Vol.2-3-251
Moore's law, Vol.1-2-20
MOV instruction (control registers), Vol.2-4-39, Vol.2-4-61, Vol.2-4-63
MOV instruction (debug registers), Vol.2-4-42, Vol.2-4-52
MOV (control registers) instructions, Vol.3-2-24, Vol.3-5-24, Vol.3-9-18, Vol.3-10-13
MOVAPD- Move Aligned Packed Double-Precision Floating-Point Values, Vol.2-4-44
MOVAPS- Move Aligned Packed Single-Precision Floating-Point Values, Vol.2-4-48
MOV instruction, Vol.1-9-6, Vol.2-4-52
MOVDDUP instruction, Vol.1-5-24, Vol.1-12-3
MOVDUP- Replicate Double FP Values, Vol.2-4-58
MOVDQ- Move Aligned Packed Integer Values, Vol.2-4-65
MOVDQ- Move Unaligned Packed Integer Values, Vol.2-4-70
MOVHPS- Move Packed Single-Precision Floating-Point Values High to Low, Vol.2-4-78
MOVHLP instruction, Vol.1-10-8
MOVHPD instruction, Vol.1-11-6

Index-20 Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
INDEX

MULPD - Multiply Packed Double-Precision Floating-Point Values, Vol.2-4-148
MULPS instruction, Vol.1-10-8
MULPS- Multiply Packed Single-Precision Floating-Point Values, Vol.2-4-151
MULSD instruction, Vol.1-11-6
MULSD- Multiply Scalar Double-Precision Floating-Point Values, Vol.2-4-154
MULSS instruction, Vol.1-10-8
MULSS- Multiply Scalar Single-Precision Floating-Point Values, Vol.2-4-156
Multi-byte no operation, Vol.2-4-163, Vol.2-4-165, Vol.1-10-13
Multi-core technology, Vol.1-2-18
See multi-threading support
Multiple-processor management
bus locking, Vol.3-3-9
guaranteed atomic operations, Vol.3-9-2
initialization
MP protocol, Vol.3-9-19
procedure, Vol.3-9-55
local APIC, Vol.3-11-1
memory ordering, Vol.3-9-6
MP protocol, Vol.3-9-19
overview of, Vol.3-9-1
SMM considerations, Vol.4-32-16
Multiple-processor system
local APIC and I/O APICs, Pentium 4, Vol.3-11-3
local APIC and I/O APIC, P6 family, Vol.3-11-3
Multisegment model, Vol.3-3-4
Multitasking
initialization for, Vol.3-10-10, Vol.3-10-11
initializing IA-32e mode, Vol.3-10-11
linking tasks, Vol.3-8-15
mechanism, description of, Vol.3-8-2
overview, Vol.3-8-1
setting up TSS, Vol.3-10-10
setting up TSS descriptor, Vol.3-10-10
Multi-threading capability, Vol.1-2-18
Multi-threading support
executing multiple threads, Vol.3-9-27
handling interrupts, Vol.3-9-27
logical processors per package, Vol.3-9-25
mapping resources, Vol.3-9-34
microcode updates, Vol.3-9-33
performance monitoring counters, Vol.3-9-33
programmable considerations, Vol.3-9-34
See also: Hyper-Threading Technology and dual-core technology
MULX - Unsigned Multiply Without Affecting Flags, Vol.2-4-158
MVMV, Vol.1-6-1, Vol.1-6-5, Vol.1-6-37
CPUID flag, Vol.2-3-243
feature data, Vol.2-3-251
power management extensions, Vol.3-15-35
description, Vol.1-10-3
flush-to-zero flag (FTZ), Vol.1-10-4
FXSAVE and FXRSTOR instructions, Vol.1-11-23
LDMXCSR instruction, Vol.1-11-24
load and store instructions, Vol.1-10-12
RC field, Vol.1-4-18
saving on a procedure or function call, Vol.1-11-23
SIMD floating-point mask and flag bits, Vol.1-10-4
SIMD floating-point rounding control field, Vol.1-10-4
state management instructions, Vol.1-5-19, Vol.1-10-12
STMXCSR instruction, Vol.1-11-24
writing to while preventing general-protection exceptions (#GP), Vol.1-11-21

N
NaNs
description of, Vol.1-4-13, Vol.1-4-15

encoding of, Vol.1-4-5, Vol.1-4-14
SNANs vs. QNANs, Vol.1-4-15
NaN compatibility, IA-32 processors, Vol.1-23-9
NaN. testing for, Vol.2-3-451
NE (numeric error) flag
CR0 control register, Vol.3-2-15, Vol.3-6-48, Vol.3-10-6, Vol.3-10-7,
Near
return, RET instruction, Vol.2-4-558
Near call
description of, Vol.1-6-4
operation, Vol.1-6-4
Near pointer
64-bit mode, Vol.1-4-7
legacy modes, Vol.1-4-6
Near return operation, Vol.1-6-4
NEG instruction, Vol.1-7-8, Vol.2-3-610, Vol.2-4-163, Vol.3-9-4
NetBurst microarchitecture (see Intel NetBurst microarchitecture)
NMI interrupt, Vol.3-2-26, Vol.3-11-3
description of, Vol.3-6-2
handling during initialization, Vol.3-10-9
handling in SMM, Vol.4-32-11
handling multiple NMI s, Vol.3-6-6
masking, Vol.1-23-27
recieving when processor is shutdown, Vol.3-6-34
reference information, Vol.3-6-27
vector, Vol.3-6-2
NMI# pin, Vol.3-6-2, Vol.3-6-27
No operation, Vol.2-4-163, Vol.2-4-165, Vol.1-18-12
Nomenclature, used in instruction reference pages
NMI
description of, Vol.3-10-13
Non-conforming code segments
accessing, Vol.3-5-11
C (conforming) flag, Vol.3-5-11
description of, Vol.3-3-13
Non-conforming data
setting up counters, Vol.3-20-139
Non-Halted CPI method
description of, Vol.3-10-9
Non-halted clockticks, Vol.3-20-139
Non-halted clockticks
setting up counters, Vol.3-20-139
Non-Halted CPI method, Vol.3-20-139
Non-maskable interrupt (see NMI)
Non-number encodings, floating-point format, Vol.1-4-13
Non-precise event-based sampling
defined, Vol.3-20-107
used for at-retirement counting, Vol.3-20-117
writing an interrupt service routine for, Vol.3-18-26
Non-retirement events, Vol.3-20-106
Non-sleep clockticks, Vol.3-20-139
Non-temporal data
accessing, Vol.1-10-12
description, Vol.1-10-12
temporal vs. non-temporal data, Vol.1-10-12
Non-waiting instructions, x87 FPU, Vol.1-8-25
NOP instruction, Vol.1-7-23, Vol.2-4-165
Normalized finite number, Vol.1-4-5, Vol.1-4-13, Vol.1-4-14
NOT instruction, Vol.1-7-10, Vol.2-3-610, Vol.2-4-166, Vol.3-9-4
Notation
bit and byte order, Vol.1-1-6, Vol.2-1-5, Vol.3-1-7, Vol.4-1-5
conventions, Vol.3-1-7, Vol.4-1-5
hexadecimal and binary numbers, Vol.1-1-7, Vol.2-1-7, Vol.3-1-8,
Vol.4-1-6
instruction operands, Vol.1-1-7, Vol.2-1-6
Instructions
operands, Vol.3-1-8, Vol.4-1-6
notational conventions, Vol.1-1-6
reserved bits, Vol.1-1-6, Vol.2-1-6, Vol.3-1-7, Vol.4-1-5
Notational conventions, Vol.2-1-5
NT (nested task) flag
EFLAGS register, Vol.3-2-10, Vol.3-8-10, Vol.3-8-11, Vol.3-8-15
NT (nested task) flag, EFLAGS register, Vol.1-3-17, Vol.1-A-1,
Vol.2-3-539

Index-22 Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
INDEX

Null segment selector, checking for, Vol.3-5-6
Numeric overflow exception (#O), Vol.1-23-10
overview, Vol.1-4-21
SSE and SSE2 extensions, Vol.1-11-15
x87 FPU, Vol.1-8-4, Vol.1-8-28
Numeric underflow exception (#U), Vol.1-23-10
overview, Vol.1-4-22
SSE and SSE2 extensions, Vol.1-11-16
x87 FPU, Vol.1-8-4, Vol.1-8-29
NV (invert) flag, PerfEvtSel0 MSR (P6 family processors), Vol.3-30-5, Vol.3-20-135
NW (not write-through) flag
NXE bit, Vol.3-5-30

O

OE (numeric overflow exception) flag
MXCSR register, Vol.1-11-16
x87 FPU status word, Vol.1-8-5, Vol.1-8-29
OF flag, EFLAGS register, Vol.3-6-29
OF (carry) flag, EFLAGS register, Vol.2-3-502
OF (overflow) flag
EFLAGS register, Vol.1-3-16, Vol.1-6-18
OF, Vol.2-4-674
Offset (operand addressing), 64-bit mode, Vol.1-3-24
Offset (operand addressing), Vol.1-3-22
OM (numeric overflow exception) mask bit
MXCSR register, Vol.1-11-16
x87 FPU control word, Vol.1-8-7, Vol.1-8-29
On die thermal sensor, Vol.3-15-42
relevant MSRs, Vol.3-15-42
sensor enumeration, Vol.3-15-42
On-Demand
clock modulation enable bits, Vol.3-15-40
On-demand
clock modulation duty cycle bits, Vol.3-15-40
On-die sensors, Vol.3-15-36
Opcode format, Vol.2-2-3
OpCodes
addressing method codes for, Vol.1-A-1
extensions, Vol.1-A-17
extensions tables, Vol.1-A-18
group numbers, Vol.1-A-17
integers
one-byte opcodes, Vol.1-A-7
two-byte opcodes, Vol.1-A-7
key to abbreviations, Vol.1-A-1
ModRM byte, Vol.1-A-17
opcode maps, Vol.1-A-1
operand type codes for, Vol.1-A-2
register codes for, Vol.1-A-3
superscripts in tables, Vol.1-A-6
underlined, Vol.1-23-5
VMX instructions, Vol.1-B-112, Vol.1-B-113
x87 ESC instruction opcodes, Vol.1-A-20
Operand
addressing, modes, Vol.1-3-19
instruction, Vol.1-1-7
size attribute, Vol.1-3-18
sizes, Vol.1-3-9, Vol.1-3-19
x87 FPU instructions, Vol.1-8-15
Operands, Vol.2-1-6
instruction, Vol.3-1-8, Vol.4-1-6
operand-size prefix, Vol.3-22-1

P

P (present) flag
page-directory entry, Vol.3-6-44
guest-page table entry, Vol.3-6-44
segment descriptor, Vol.3-3-11

P6 family microarchitecture
description of, Vol.1-2-7
history of, Vol.1-2-2
P6 family processors
compatibility with FP software, Vol.1-23-7
description of, Vol.1-1-1, Vol.2-1-1, Vol.3-1-1, Vol.4-1-1
history of, Vol.1-2-2
last branch, interrupt, and exception recording, Vol.3-18-40
machine encodings, Vol.1-23-5
MSR supported by, Vol.4-2-498
P6 family microarchitecture, Vol.1-2-7
PABSW instruction, Vol.1-2-7
PABSD instruction, Vol.1-2-7
PABSB instruction, Vol.1-2-7
PABSW instruction, Vol.1-2-7

Packed
BCD integer indefinite, Vol.1-4-11

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
INDEX

BCD integers, Vol.1-4-10
bytes, Vol.1-9-3
doublewords, Vol.1-9-3
SIMD data types, Vol.1-4-8
SIMD floating-point values, Vol.1-4-8
SIMD integers, Vol.1-4-8
words, Vol.1-9-3
PACKSSdw instruction, Vol.2-4-188
PACKSSWb instruction, Vol.1-9-7, Vol.2-4-188
PAVB instruction, Vol.1-9-6
PADD/PADDW/PADDO/PADQQ - Add Packed Integers, Vol.2-4-206
PADD instruction, Vol.1-9-6
PADDQ instruction, Vol.1-11-11
PADDUSB instruction, Vol.1-9-7, Vol.2-4-217
PADDUSW instruction, Vol.1-9-7, Vol.2-4-217
PADDW instruction, Vol.1-9-6
PAE paging
feature flag, CR4 register, Vol.3-2-18, Vol.3-2-19
Page attribute table (PAT)
compatibility with earlier IA-32 processors, Vol.3-12-36
detecting support for, Vol.3-12-34
IA32_PAT MSR, Vol.3-12-34
introduction to, Vol.3-12-33
memory types that can be encoded with, Vol.3-12-34
MSR, Vol.3-12-13
precedence of cache controls, Vol.3-12-14
programming, Vol.3-12-35
selecting a memory type with, Vol.3-12-35
Page directories, Vol.3-2-6
Page directory
base address (PDBR), Vol.3-8-5
introduction to, Vol.3-2-6
overview, Vol.3-3-2
setting up during initialization, Vol.3-10-10
Page directory pointers, Vol.3-2-6
Page frame (see Page)
Page tables, Vol.3-2-6
introduction to, Vol.3-2-6
overview, Vol.3-3-2
setting up during initialization, Vol.3-10-10
Page-directory entries, Vol.3-12-5
Page-fault exception (#PF), Vol.3-4-54, Vol.3-6-44, Vol.1-23-21
Pages
disabling protection of, Vol.3-5-1
enabling protection of, Vol.3-5-1
introduction to, Vol.3-2-6
overview, Vol.3-3-2
PG flag, CR0 control register, Vol.3-5-1
split, Vol.1-23-15
Page-table entries, Vol.3-12-5, Vol.3-12-19
Paging
combining segmentation and page-level protection, Vol.3-5-29
combining with segmentation, Vol.3-3-5
defined, Vol.3-3-1
IA-32e mode, Vol.3-2-6
initializing, Vol.3-10-10
introduction to, Vol.3-2-6
large page size MTRR considerations, Vol.3-12-33
mapping segments to pages, Vol.3-4-54
page-fault exception, Vol.3-6-44, Vol.3-6-55, Vol.3-6-56
page-level protection, Vol.3-5-2, Vol.3-5-3, Vol.3-6-27
page-level protection flags, Vol.3-5-28
virtual-8086 tasks, Vol.3-21-7
PAIGNR instruction, Vol.1-5-26, Vol.1-12-8, Vol.2-4-221
PAND instruction, Vol.1-9-7, Vol.2-4-224
PANDN instruction, Vol.1-9-7, Vol.2-4-227
Parameter
passing, between 16- and 32-bit call gates, Vol.3-22-6
translation, between 16- and 32-bit code segments, Vol.3-22-6
Parameter passing
argument list, Vol.1-6-7
on stack, Vol.1-6-7
on the stack, Vol.1-6-7
through general-purpose registers, Vol.1-6-7
x87 FPU register stack, Vol.1-8-3
XMM registers, Vol.1-11-23
GETSEC, Vol.1-6-4
PAVGB instruction, Vol.1-10-11, Vol.2-4-231
PAVGw instruction, Vol.2-4-231
PB (performance monitoring/breakpoint pins) flags, DEBUGCTLMSR MSR,
Vol.3-18-39, Vol.3-18-41
PC (pin control) flag, PerfEvtSel0 and PerfEvtSel1 MSRs (P6 family processors),
Vol.3-20-24, Vol.3-20-135
PC (precision) field, x87 FPU control word, Vol.1-8-7
PC0 and PC1 (pin control) fields, CESR MSR (Pentium processor),
Vol.3-20-137
PCD pin (Pentium processor), Vol.3-12-13
PCD (page-level cache disable) flag
time-stamp counter
page-directory entries, Vol.3-10-7, Vol.3-12-13, Vol.3-12-33
page-table entries, Vol.3-10-7, Vol.3-12-13, Vol.3-12-33,
Vol.1-23-31
PE flag, CR4 register, Vol.2-4-542
PCE (performance monitoring counter enable) flag, CR4 control register,
Vol.3-2-18, Vol.3-5-24, Vol.3-20-109, Vol.3-20-135
PCE (performance-monitoring counter enable) flag, CR4 control register,
Vol.1-23-18
PCMEQBonstruction, Vol.1-9-7, Vol.2-4-245
PCMEQD instruction, Vol.1-9-7, Vol.2-4-245
PCMPGTB instruction, Vol.1-9-7, Vol.2-4-258
PCMPGTW instruction, Vol.1-9-7, Vol.2-4-258
PDBR (see CR3 control register)
PDEP - Parallel Bits Deposit, Vol.2-4-277
PE (inexact result exception) flag, Vol.1-11-16
MXCSR register, Vol.1-4-18
PE (protection enable) flag, CR0 control register, Vol.3-2-17, Vol.3-5-1,
Vol.3-10-10, Vol.3-10-13, Vol.4-32-9
PE (protection enable) flag, CR0 register, Vol.2-3-605
PEBS records, Vol.3-18-23
PEBS (precise event-based sampling) facilities
availability of, Vol.3-20-119
description of, Vol.3-20-107, Vol.3-20-119
DS save area, Vol.3-18-19
IA-32e mode, Vol.3-18-23
PEBS buffer, Vol.3-18-20, Vol.3-20-119
PEBS records, Vol.3-18-19, Vol.3-18-22
writing a PEBS interrupt service routine, Vol.3-20-120
writing interrupt service routine, Vol.3-18-26
PEBS, UNAVAILABLE flag
IA32_MISC_ENABLE MSR, Vol.3-18-20, Vol.4-2-462
Pending break enable, Vol.2-3-247
compatibility with FP software, Vol.1-23-7
last branch, interrupt, and exception recording, Vol.3-18-35
MSRs supported, Vol.4-2-64, Vol.4-2-81, Vol.4-2-94, Vol.4-2-113,
Vol.4-2-115, Vol.4-2-141, Vol.4-2-146, Vol.4-2-453, Vol.4-2-479
time-stamp counter, Vol.3-18-42
Pentium 4 processor supporting Hyper-Threading Technology
description of, Vol.1-2-2
P6 family microarchitecture, Vol.1-2-7
Pentium II Xeon processor
description of, Vol.1-2-2
INDEX
INDEX

LOCK, Vol.2-2-1, Vol.3-2-610
Operand-size override prefix, Vol.2-2-2
REP or REP/REPZ, Vol.2-2-1
REPNE/REPNZ, Vol.2-2-1
REP/REPE/REPZ/REPNZ, Vol.2-4-554
REX prefix encodings, Vol.1-8-2
Segment override prefixes, Vol.2-2-2
Previous task link field, TSS, Vol.3-8-5, Vol.3-8-15, Vol.3-8-16
Privilege levels
  checking when accessing data segments, Vol.3-5-8
  checking, for call gates, Vol.3-5-15
  checking, when transferring program control between code segments, Vol.3-5-10
  description of, Vol.1-6-8, Vol.3-5-6
  inter-privilege level calls, Vol.1-6-7
  protection rings, Vol.1-6-8, Vol.3-5-8
  stack switching, Vol.1-6-14
Privileged instructions, Vol.3-5-23
Procedure calls
  description of, Vol.1-6-4
  far call, Vol.1-6-4
  for block-structured languages, Vol.1-6-20
  inter-privilege level call, Vol.1-6-8
  linking, Vol.1-6-3
  near call, Vol.1-6-4
  overview, Vol.1-6-1
  return instruction pointer (EIP register), Vol.1-6-3
  saving procedure state information, Vol.1-6-7
  stack, Vol.1-6-1
  stack switching, Vol.1-6-8
  to exception handler procedure, Vol.1-6-13
  to exception task, Vol.1-6-18
  to interrupt handler procedure, Vol.1-6-13
  to interrupt task, Vol.1-6-18
  to other privilege levels, Vol.1-6-7
  types of, Vol.1-6-1
Processor families
  ODH, Vol.3-17-1
  OFH, Vol.3-17-1
Processor identification
  earlier Intel architecture processors, Vol.1-20-1
  early processors, Vol.1-20-1
  notes on where to start, Vol.1-20-1
  using CPUID, Vol.1-20-1
  using CPUID instruction, Vol.1-20-1
Processor management
  initialization, Vol.3-10-1
  local APIC, Vol.3-11-1
  microcode update facilities, Vol.3-10-27
  overview of, Vol.3-9-1
  See also: multiple-processor management
Processor ordering, description of, Vol.3-9-6
Processor state information, saving, Vol.1-6-7
PROCHOT# or FORCEPR# event bit, Vol.3-15-42, Vol.3-15-46
Protected mode
  IDT initialization, Vol.3-10-10
  initialization for, Vol.3-10-9
  I/O, Vol.1-19-3
  memory models used, Vol.1-3-9
  mixing 16-bit and 32-bit code modules, Vol.3-22-1
  mode switching, Vol.3-10-13
  overview, Vol.1-3-1
  PE flag, CR0 register, Vol.3-5-1
  switching to, Vol.3-5-1, Vol.3-10-13
  system data structures required during initialization, Vol.3-10-9
Protection
  combining segment & page-level, Vol.3-5-29
  disabling, Vol.3-5-1
  enabling, Vol.3-5-1
  flags used for page-level protection, Vol.3-5-2, Vol.3-5-3
  flags used for segment-level protection, Vol.3-5-2
IA-32e mode, Vol.3-5-3
  of exception, interrupt-handler procedures, Vol.3-6-16
  overview of, Vol.3-5-1
  page level, overriding, Vol.3-5-29
  page-level protection flags, Vol.3-5-28
  segment level, Vol.3-5-1
  user/supervisor type, Vol.3-5-28
Protection rings, Vol.1-6-8, Vol.3-5-8
PSADDw instruction, Vol.1-10-12, Vol.2-4-412
PSE (page size extension) flag
PSE-36 page size extension, Vol.3-3-6
Pseudo-functions
  VMfail, Vol.3-31-2
  VMfailValid, Vol.3-31-2
  VMfailInvalid, Vol.3-31-2
  VMsucceed, Vol.3-31-2
Pseudo-infinity, Vol.1-23-9
Pseudo-NaN, Vol.1-23-9
Pseudo-zero, Vol.1-23-9
PSHUFB instruction, Vol.1-5-25, Vol.1-12-8, Vol.2-4-416
PSHUFDB instruction, Vol.1-7-8, Vol.2-4-420
PSHUFHW instruction, Vol.1-11-11, Vol.2-4-424
PSHUFVLW instruction, Vol.1-11-11, Vol.2-4-427
PSHUFW instruction, Vol.1-10-12, Vol.1-11-11, Vol.2-4-430
PSIGNB instruction, Vol.2-4-431
PSIGNB/W/D instruction, Vol.1-5-25, Vol.1-12-8
PSIGND instruction, Vol.2-4-431
PSIGNW instruction, Vol.2-4-431
PSLLD instruction, Vol.1-9-8, Vol.2-4-437
PSLLQ instruction, Vol.1-9-8, Vol.2-4-437
PSLLW instruction, Vol.1-9-8, Vol.2-4-437
PSRAD instruction, Vol.2-4-449
PSRAW instruction, Vol.2-4-449
PSRLD instruction, Vol.2-4-461
PSRLQ instruction, Vol.1-11-11, Vol.2-4-459
PSRLQ instruction, Vol.2-4-461
PSRLW instruction, Vol.2-4-461
P-state, Vol.3-15-1
PSUBB instruction, Vol.1-9-6, Vol.2-4-473
PSUBD instruction, Vol.1-9-6, Vol.2-4-473
PSUBQ instruction, Vol.1-11-11, Vol.2-4-481
PSUBSB instruction, Vol.1-9-7, Vol.2-4-484
PSUBSW instruction, Vol.1-9-7, Vol.2-4-484
PSUBUS instruction, Vol.1-9-7, Vol.2-4-488
PSUBUSW instruction, Vol.1-9-7, Vol.2-4-488
PSUBW instruction, Vol.1-9-6, Vol.2-4-473
PTEST- Packed Bit Test, Vol.2-3-576
PUNPCKLBW instruction, Vol.1-9-7, Vol.2-4-496
PUNPCKLBW instruction, Vol.1-9-7, Vol.2-4-496
PUNPCKLBW instruction, Vol.1-9-7, Vol.2-4-496
PUNPCKLWD instruction, Vol.1-10-12, Vol.2-4-496
PUNPCKLWD instruction, Vol.1-9-7, Vol.2-4-496
PUNPCKLQDQ instruction, Vol.1-9-7, Vol.2-4-496
PUNPCKLQDQ instruction, Vol.1-10-12, Vol.2-4-496
PUNPCKLQDQ instruction, Vol.1-11-11, Vol.2-4-506
PUNPCLWD instruction, Vol.1-9-7, Vol.2-4-506
PUSHAD instruction, Vol.1-11-11, Vol.2-4-519
PVI (protected-mode virtual interrupts) flag
  [protected-mode virtual interrupts] flag
PWT pin (Pentium processor), Vol.3-12-13
PWT (page-level write-through) flag
INDEX

REX prefixes, Vol.1-3-2, Vol.1-3-12, Vol.1-3-19
addressing modes, Vol.2-2-9
and INC/DEC, Vol.2-2-8
field names, Vol.2-2-9
ModR/M byte, Vol.2-2-8
overview, Vol.2-2-7
REXB, Vol.2-2-8
REXR, Vol.2-2-8
REXW, Vol.2-2-8
special encodings, Vol.2-2-11
RF (resume) flag
EFLAGS register, Vol.3-2-10, Vol.3-6-7
RF (resume) flag, EFLAGS register, Vol.1-3-17, Vol.1-1-A-1
RFLAGS, Vol.1-3-18
RFLAGS register, Vol.1-7-22
See EFLAGS register
RIP register, Vol.1-6-4
64-bit mode, Vol.1-7-2
description of, Vol.1-3-18
related to EIP, Vol.1-7-2
RIP-relative addressing, Vol.2-2-12
ROL instruction, Vol.1-7-13, Vol.2-4-526
ROR instruction, Vol.1-7-13, Vol.2-4-526
RORX - Rotate Right Logical Without Affecting Flags, Vol.2-4-571
Rounding
modes, floating-point operations, Vol.1-4-18, Vol.2-4-573
modes, x87 FPU, Vol.1-8-8
toward zero (truncation), Vol.1-4-18
Rounding control (RC) field
x87 FPU control word, Vol.1-4-18, Vol.1-8-8, Vol.2-4-573
Rounding, round to integer, x87 FPU operation, Vol.2-3-423
ROUNDPD- Round Packed Double-Precision Floating-Point Values, Vol.2-4-675
RPL
description of, Vol.3-3-8, Vol.3-5-8
field, segment selector, Vol.3-5-2
RPL field, Vol.2-3-93
RSI register, Vol.1-3-12
Vol.3-26-4, Vol.4-3-22, Vol.4-3-22, Vol.4-3-23, Vol.4-3-23-13,
Vol.4-3-23-15, Vol.4-3-23-18
RSR register, Vol.1-3-12, Vol.1-6-4
RSQRTPS instruction, Vol.1-10-8, Vol.2-4-583
RSQRTSS instruction, Vol.1-10-8, Vol.2-4-585
RsvdZ, Vol.3-11-40
R/m field, instruction format, Vol.2-2-3
R/S# pin, Vol.3-6-3
R/W (read/write) flag
directory entry, Vol.3-5-1, Vol.3-5-2, Vol.3-5-28
page-table entry, Vol.3-5-1, Vol.3-5-2, Vol.3-5-28
R/W0-R/W3 (read/write) fields
S
S (descriptor type) flag
segment descriptor, Vol.3-3-11, Vol.3-3-12, Vol.3-5-2, Vol.3-5-5
Safer Mode Extensions, Vol.1-6-1
SAL instruction, Vol.1-7-10, Vol.2-4-592
SAR instruction, Vol.1-7-11, Vol.2-4-592
Saturation arithmetic (MMX instructions), Vol.1-9-4
Scalar operations
scalar double-precision FP operands, Vol.1-11-5
scalar single-precision FP operands, Vol.1-10-7
Scale, x87 FPU operation, Vol.1-8-21, Vol.2-3-429
Scaling bias value, Vol.1-8-29, Vol.1-8-30
Scan string instructions, Vol.2-4-604
SCAS instruction, Vol.1-3-17, Vol.1-7-18, Vol.2-4-555, Vol.2-4-604
SCASB instruction, Vol.2-4-604
SCASD instruction, Vol.2-4-604
SCASW instruction, Vol.2-4-604
Segment
defined, Vol.1-3-7
descriptor, segment limit, Vol.2-3-618
limit, Vol.2-3-618
maximum number, Vol.1-3-7
registers, moving values to and from, Vol.2-4-36
selector, RPL field, Vol.2-3-93
Segment descriptors
access rights, Vol.3-5-24
access rights, invalid values, Vol.1-23-19
base address fields, Vol.3-3-10
code type, Vol.3-5-2
data type, Vol.3-5-2
description of, Vol.3-2-4, Vol.3-3-9
DPL (descriptor privilege level) field, Vol.3-3-11, Vol.3-5-2
d/B (default operation size/default stack pointer size and/or upper bound) flag, Vol.3-3-11, Vol.3-5-4
e (expansion direction) flag, Vol.3-5-2, Vol.3-5-4
g (granularity) flag, Vol.3-3-11, Vol.3-5-2, Vol.3-5-4
limit field, Vol.3-5-2, Vol.3-5-4
loading, Vol.1-23-20
P (segment-present) flag, Vol.3-3-11
S (descriptor type) flag
Vol.3-3-11
Vol.3-5-2
segment limit field, Vol.3-3-10
system type, Vol.3-5-2
tables, Vol.3-5-14
tSS descriptor, Vol.3-8-5, Vol.3-8-6
type field, Vol.3-3-10, Vol.3-3-12, Vol.3-5-2, Vol.3-5-5
type field, encoding, Vol.3-3-14
when P (segment-present) flag is clear, Vol.3-3-11
Segment limit
checking, Vol.3-2-24
field, segment descriptor, Vol.3-3-10
Segment not present exception (#NP), Vol.3-3-11
Segment override prefixes, Vol.1-3-21
Segment registers
default usage rules, Vol.1-3-21
IA-32e mode, Vol.3-3-9
part of basic programming environment, Vol.1-7-1
saved in TSS, Vol.3-8-4
Segment selector
description of, Vol.1-3-7, Vol.1-3-13
Segment override prefixes, Vol.1-3-21
specifying, Vol.1-3-21
Segment selectors
description of, Vol.3-3-7
index field, Vol.3-3-7
null, Vol.3-3-6
null in 64-bit mode, Vol.3-5-6
RPL field, Vol.3-3-8, Vol.3-5-2
tI (table indicator) flag, Vol.3-3-7
Segmented addressing, Vol.2-1-7, Vol.3-1-8, Vol.4-1-6
Segment-not-present exception (#NP), Vol.3-6-38
Segments
64-bit mode, Vol.3-3-5
basic flat model, Vol.3-3-3
code type, Vol.3-3-12
combining segment, page-level protection, Vol.3-5-29
combining with paging, Vol.3-3-5
compatibility mode, Vol.3-3-5
data type, Vol.3-3-12
defined, Vol.3-3-1
disabling protection of, Vol.3-5-1
enabling protection of, Vol.3-5-1
SSE2 extensions
SIMD floating-point exception (#XM), Vol.1-11-18
SIMD floating-point exceptions, Vol.1-11-13
SIMD floating-point mask and flag bits, Vol.1-10-4
SIMD floating-point rounding control field, Vol.1-10-4
SSE and SSE2 conversion instruction chart, Vol.1-11-9
SSE feature flag, CPUID instruction, Vol.1-11-20
SSE compatibility, Vol.1-10-5
system programming, Vol.1-13-23
unpack instructions, Vol.1-10-9
updating MMX technology routines
using 128-bit SIMD integer instructions, Vol.1-11-24
using TS flag to control saving of state, Vol.1-14-7
x87 FPU compatibility, Vol.1-10-5
XMM registers, Vol.1-10-3
SSE feature flag
CPUID instruction, Vol.1-14-2
SSE instructions
descriptions of, Vol.1-10-6
SIMD floating-point exception cross-reference, Vol.1-C-3
summary of, Vol.1-5-17
SSE2 extensions
128-bit packed single-precision
data type, Vol.1-11-3
128-bit packed single-precision data type, Vol.1-12-1
128-bit SIMD integer instruction extensions, Vol.1-11-11
64-bit and 128-bit SIMD integer instructions, Vol.1-11-11
64-bit mode, Vol.1-11-3
arithmetic instructions, Vol.1-11-6
branch hints, Vol.1-11-13
branching on arithmetic operations, Vol.1-11-24
cacheability control instructions, Vol.1-11-12
cacheability hint instructions, Vol.1-11-25
cacheability instruction encodings, Vol.1-B-56
caller-save requirement for procedure and function calls, Vol.1-11-24
checking for SSE and SSE2 support, Vol.1-11-19
checking for with CPUID, Vol.1-14-2
checking support for FXSAVE/FXRSTOR, Vol.1-14-2
comparison instructions, Vol.1-11-7
compatibility mode, Vol.1-11-3
compatibility of SIMD and x87 FPU floating-point data types,
Vol.1-11-22
conversion instructions, Vol.1-11-9
CPUID feature flag, Vol.3-10-8
CPUID flag, Vol.2-3-247
data movement instructions, Vol.1-11-5
data types, Vol.1-11-3, Vol.1-12-1
denormal operand exception (#D), Vol.1-11-15
denormals-are-zero mode, Vol.1-11-3
divide by zero exception (#Z), Vol.1-11-15
EM flag, Vol.3-2-16
emulation of, Vol.1-11-6
exceptions, Vol.1-11-13
facilities for automatic saving of state, Vol.1-14-6, Vol.1-14-7
floating-point encodings, Vol.1-B-48
floating-point format, Vol.1-4-11
generating SIMD floating-point exceptions, Vol.1-11-16
handling combinations of masked and unmasked exceptions,
Vol.1-11-18
handling masked exceptions, Vol.1-11-17
handling SIMD floating-point exceptions in software, Vol.1-11-18
handling unmasked exceptions, Vol.1-11-18
inexact result exception (#P), Vol.1-11-16
initialization, Vol.3-10-8
instruction prefixes, effect on SSE and SSE2 instructions, Vol.1-11-25
instruction set, Vol.1-5-19
integer instruction encodings, Vol.1-B-52
interaction of SIMD and x87 FPU floating-point exceptions,
Vol.1-11-18
interaction of SSE and SSE2 instructions with x87 FPU and MMX
instructions, Vol.1-11-22
interfacing with SSE and SSE2 procedures and functions, Vol.1-11-23
intermixing packed and scalar floating-point and 128-bit SIMD integer
instructions and data, Vol.1-11-23
introduction of into the IA-32 architecture, Vol.1-23-3
invalid operation exception (#N), Vol.1-11-14
logical exceptions, Vol.1-11-7
masked responses to invalid arithmetic operations, Vol.1-11-14
memory ordering instructions, Vol.1-11-12
MMX technology compatibility, Vol.1-11-3
numeric overflow exception (#O), Vol.1-11-15
numeric underflow exception (#U), Vol.1-11-16
packed 128-Bit SIMD data types, Vol.1-4-8
packed and scalar floating-point instructions, Vol.1-11-4
programming environment, Vol.1-11-2
providing operating system support for, Vol.1-14-1
QNaN floating-point indefinite, Vol.1-4-17
restoring SSE and SSE2 state, Vol.1-11-21
REX prefixes, Vol.1-11-3
saving and restoring state, Vol.1-14-6
saving SSE and SSE2 state, Vol.1-11-21
saving state on task, context switches, Vol.1-14-6
saving XMM register state on a procedure or function call, Vol.1-11-23
shuffle instructions, Vol.1-11-7
SIMD floating-point exception conditions, Vol.1-11-14
SIMD floating-point exception cross reference, Vol.1-C-5
SIMD Floating-point exception (#XM), Vol.3-6-53
SIMD floating-point exception (#XM), Vol.1-11-18
SIMD floating-point exceptions, Vol.1-11-13
SSE and SSE2 conversion instruction chart, Vol.1-11-9
SSE compatibility, Vol.1-11-3
SSE2 feature flag, CPUID instruction, Vol.1-11-20
system programming, Vol.1-13-23
unpack instructions, Vol.1-11-7
updating MMX technology routines using 128-bit SIMD integer
instructions, Vol.1-11-24
using TS flag to control saving state, Vol.1-14-7
x87 FPU compatibility, Vol.1-11-3
SSE2 feature flag
CPUID instruction, Vol.1-14-2
SSE2 instructions
SIMD floating-point exception cross-reference, Vol.1-C-5
summary of, Vol.1-5-19
SSE3
CPUID flag, Vol.2-3-243
SSE3 extensions
64-bit mode, Vol.1-12-1
asymmetric processing, Vol.1-12-1
checking for with CPUID, Vol.1-14-2
compatibility mode, Vol.1-12-1
CPUID feature flag, Vol.3-10-8
CPUID flag, Vol.2-3-243
DNA exceptions, Vol.1-12-9
EM flag, Vol.3-2-16
emulation, Vol.1-12-10
emulation of, Vol.1-14-6
enabling support in a system executive, Vol.1-12-5, Vol.1-12-18
event mgmt instruction encodings, Vol.1-B-57
event mgmt instructions, Vol.1-11-25
exceptions, Vol.1-12-9
facilities for automatic saving of state, Vol.1-14-6, Vol.1-14-7
floating-point instruction encodings, Vol.1-B-57
guideline for packed addition/subtraction instructions, Vol.1-12-6
horizontal addition/subtraction instructions, Vol.1-12-4
horizontal processing, Vol.1-12-1
initialization, Vol.3-10-8
instruction that addresses cache line splits, Vol.1-5-23
instruction that improves X87-FP integer conversion, Vol.1-5-23

Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 Index -31
INDEX

instructions for horizontal addition/subtraction, Vol.1-5-23
instructions for packed addition/subtraction, Vol.1-5-23
instructions that enhance LOAD/MOVE/DUPLICATE, Vol.1-5-24
instructions that improve synchronization between agents, Vol.1-5-24
integer instruction encodings, Vol.1-5-60
introduction into the IA-32e architecture, Vol.1-23-3
LOAD/MOVE/DUPLICATE enhancement instructions, Vol.1-12-3
MMX technology compatibility, Vol.1-12-1
numeric error flag and IGNNE#, Vol.1-12-9
packed addition/subtraction instructions, Vol.1-12-4
programming environment, Vol.1-12-1
providing operating system support for, Vol.1-14-1
RDI prefixes, Vol.1-12-1
saving and restoring state, Vol.1-14-6
saving state on task, context switches, Vol.1-14-6
SIMD floating-point exception cross reference, Vol.1-12-1
specialized 120-bit load instruction, Vol.1-12-3
SSE compatibility, Vol.1-12-1
SSE2 compatibility, Vol.1-12-1
system programming, Vol.1-13-23
using TS flag to control saving of state, Vol.1-14-7
x86 FPU compatibility, Vol.1-12-1

SSE3 feature flag
CPUID instruction, Vol.1-12-4
CPUID instruction

SSE3 instructions
descriptions of, Vol.1-12-2
SIMD floating-point exception
cross-reference, Vol.1-12-2
summary of, Vol.1-12-2

SSE3 extensions

64-bit mode, Vol.1-12-1
asymmetric processing, Vol.1-12-1
checking for support, Vol.1-12-9
compatibility mode, Vol.1-12-1
CPUID flag, Vol.2-3-374
data types, Vol.1-12-1
DNA exceptions, Vol.1-12-9
emulation, Vol.1-12-10
enabling support in a system executive, Vol.1-12-9
exceptions, Vol.1-12-9
horizontal add/subtract instructions, Vol.1-12-7
horizontal processing, Vol.1-12-1
multiply and add packed instructions, Vol.1-12-8
numeric error flag and IGNNE#, Vol.1-12-9
packed absolute value instructions, Vol.1-12-7
packed align instruction, Vol.1-12-8
packed multiply high instructions, Vol.1-12-8
packed shuffle instruction, Vol.1-12-8
programming environment, Vol.1-12-1

SSE3 instructions
descriptions of, Vol.1-12-6
summary of, Vol.1-12-6

Stack

64-bit mode, Vol.1-3-5, Vol.1-6-4
64-bit mode behavior, Vol.1-6-19
address-size attribute, Vol.1-5-3
alignment, Vol.1-5-2
alignment of stack pointer, Vol.1-6-2
current stack, Vol.1-6-1, Vol.1-6-3
description of, Vol.1-6-1
eip register (return instruction pointer), Vol.1-6-3
maximum size, Vol.1-6-1
number allowed, Vol.1-6-1
overview of, Vol.1-3-4
passing parameters on, Vol.1-6-7
popping values from, Vol.1-6-1
procedure linking information, Vol.1-6-3
pushing values on, Vol.1-6-1
return instruction pointer, Vol.1-6-3
SS register, Vol.1-6-1
stack segment, Vol.1-3-14, Vol.1-6-1
stack-frame base pointer, EBP register, Vol.1-6-3
switching
on calls to interrupt and exception handlers, Vol.1-6-14
on inter-privilege level calls, Vol.1-6-10, Vol.1-6-17
privilege levels, Vol.1-6-8
width, Vol.1-6-2
Stack fault exception (#SS), Vol.3-6-40
Stack pointers
privilege level 0, 1, and 2 stacks, Vol.3-8-5
size of, Vol.3-3-11
Stack segments
paging of, Vol.3-2-6
privilege level check when loading SS register, Vol.3-5-10
size of stack pointer, Vol.3-3-11
Stack switching
exceptions/interrupts when switching stacks, Vol.3-6-8
IA-32e mode, Vol.3-6-21
inter-privilege level calls, Vol.3-5-17
Stack-fault exception (#SS), Vol.1-23-34
Stacks
error code pushes, Vol.1-23-32
faults, Vol.3-6-40
for privilege levels 0, 1, and 2, Vol.3-5-17
interlevel RET/IRET
from a 16-bit interrupt or call gate, Vol.1-23-33
interrupt stack table, 64-bit mode, Vol.3-6-22
management of control transfers for 16- and 32-bit procedure calls, Vol.3-22-4
operation on pushes and pops, Vol.1-23-32
pointers to in TSS, Vol.3-8-5
stack switching, Vol.3-5-17, Vol.3-6-21
usage on call to exception
or interrupt handler, Vol.1-23-33
Stack, pushing values on, Vol.2-4-516
Stack, x87 FPU
stack fault, Vol.1-8-6
stack overflow and underflow exception (#IS), Vol.1-8-4, Vol.1-8-26
Status flags
Status flags, EFLAGS register, Vol.2-3-177, Vol.2-3-179, Vol.2-3-373,
STC instruction, Vol.1-3-16, Vol.1-7-21, Vol.2-4-660
STD instruction, Vol.1-3-17, Vol.1-7-21, Vol.2-4-661
Stepping information, Vol.2-3-251
Stepping information, following processor initialization or reset,
Vol.3-10-5
Sticky bits, Vol.1-8-5
Store buffer

caching terminology, Vol.3-12-5
characteristics of, Vol.3-12-4
description of, Vol.3-12-5, Vol.3-12-20
in IA-32 processors, Vol.1-23-34
location of, Vol.3-12-1
operation of, Vol.3-12-20
STOS instruction, Vol.1-3-17, Vol.1-7-19, Vol.2-4-555, Vol.2-4-665
STOSB instruction, Vol.2-4-665
STOSD instruction, Vol.2-4-665
STOSQ instruction, Vol.2-4-665
STOSW instruction, Vol.2-4-665
STR instruction, Vol.2-4-668, Vol.3-2-24, Vol.3-3-16, Vol.3-8-7
Streaming SIMD extensions 2 (see SSE2 extensions)
Streaming SIMD extensions (see SSE extensions)
String data type, Vol.1-4-8
Vol.2-4-178, Vol.2-4-604, Vol.2-4-665
Strong uncached (UC) memory type
description of, Vol.3-12-6
effect on memory ordering, Vol.3-9-17
use of, Vol.3-10-8, Vol.3-12-8
ST(0), top-of-stack register, Vol.1-8-3
Sub C-state, Vol.3-15-35
SUBPD- Subtract Packed Double Precision Floating-Point Values, Vol.2-4-675
SUBPD- Subtract Packed Double-Precision Floating-Point Values, Vol.2-4-675
SUBPS- Subtract Packed Single-Precision Floating-Point Values, Vol.2-4-681
SUBSS- Subtract Scalar Single-Precision Floating-Point Values, Vol.2-4-683
Superscalar microarchitecture
P6 family microarchitecture, Vol.1-2-2
P6 family processors, Vol.1-2-7
Pentium 4 processor, Vol.1-2-9
Pentium Pro processor, Vol.1-2-2
Pentium processor, Vol.1-2-2
Supervisor mode
description of, Vol.3-5-28
U/S (user/supervisor) flag, Vol.3-5-28
SVR (spurious-interrupt vector register), local APIC, Vol.3-11-8, Vol.1-23-28
SwapGS instruction, Vol.2-4-685, Vol.3-2-7
Syscall instruction, Vol.2-4-687, Vol.3-2-7, Vol.3-5-22
Sysenter instruction, Vol.2-4-690, Vol.3-3-8, Vol.3-5-10, Vol.3-5-20, Vol.3-5-21
CpuID flag, Vol.2-3-246
Sysenter/cs_msr, Vol.3-5-21
Sysenter/iep_msr, Vol.3-5-21
Sysenter/esp_msr, Vol.3-5-21
Sysexit instruction, Vol.2-4-693, Vol.3-3-8, Vol.3-5-10, Vol.3-5-20, Vol.3-5-21
CpuID flag, Vol.2-3-246
Sysret instruction, Vol.2-4-696, Vol.3-2-7, Vol.3-5-22
System architecture, Vol.3-2-1, Vol.3-2-2
data structures, Vol.3-2-2
instructions, Vol.3-2-7, Vol.3-2-23
registers in IA-32e mode, Vol.3-2-7
registers, introduction to, Vol.3-2-6
segment descriptor, layout of, Vol.3-5-2
segments, paging of, Vol.3-2-6
System management mode (see SMM)
System programming
Mmx technology, Vol.3-13-1
SSE/SSE2/SSE3 extensions, Vol.1-13-23
System-management mode (see SMM)

T
T (debug trap) flag, TSS, Vol.3-8-5
tangient, x87 FPU operation, Vol.1-8-20, Vol.2-3-421
Task gate, Vol.1-6-18
Task gates
descriptor, Vol.3-8-8
executing a task, Vol.3-8-2
handling a virtual-8086 mode interrupt or exception through, Vol.3-21-14
IA-32e mode, Vol.3-2-5
in idt, Vol.3-6-10
introduction for IA-32e, Vol.3-2-4
introduction to, Vol.3-2-4, Vol.3-2-5
layout of, Vol.3-6-10
referencing of TSS descriptor, Vol.3-6-17
Task management, Vol.3-8-1
data structures, Vol.3-8-3
mechanism, description of, Vol.3-8-2
Task register, Vol.1-3-4, Vol.3-3-16
description of, Vol.3-2-13, Vol.3-8-1, Vol.3-8-7
IA-32e mode, Vol.3-2-13
initializing, Vol.3-10-11
introduction to, Vol.3-2-6
loading, Vol.2-3-621
storing, Vol.2-4-688
Task state segment (see TSS)
Task switch
CALL instruction, Vol.2-3-139
return from nested task, iret instruction, Vol.2-3-539
Task switching
description of, Vol.3-8-3
exception condition, Vol.3-18-11
operation, Vol.3-8-10
preventing recursive task switching, Vol.3-8-16
saving MMX state on, Vol.3-13-4
saving SSE/SSE2/SSE3 state
on task or context switches, Vol.1-14-6
T (debug trap) flag, Vol.3-8-5
Tasks
address space, Vol.3-8-16
description of, Vol.3-8-1
exception handler, Vol.1-6-18
exception-handler task, Vol.3-6-11
executing, Vol.3-8-2
Intel 286 processor tasks, Vol.1-23-37
interrupt handler, Vol.1-6-18
interrupt-handler task, Vol.3-6-11
interruptions and exceptions, Vol.3-6-17
linking, Vol.3-8-15
logical address space, Vol.3-8-18
management, Vol.3-8-1
mapping linear and physical address space, Vol.3-8-17
restart following an exception or interrupt, Vol.3-6-5
state (context), Vol.3-8-2, Vol.3-8-3
structure, Vol.3-8-1
switching, Vol.3-8-3
task management data structures, Vol.3-8-3
Temporal data, Vol.1-10-12
TEST instruction, Vol.1-7-14, Vol.2-4-703, Vol.2-5-718
Thermal Monitor, Vol.1-2-4
CpuID flag, Vol.2-3-247
Thermal Monitor 2, Vol.2-3-243
CpuID flag, Vol.2-3-243
Thermal monitoring
advanced power management, Vol.3-15-35
automatic, Vol.3-15-37
automatic thermal monitoring, Vol.3-15-36
clock-modulation bits, Vol.3-15-40
C-state, Vol.3-15-35
detection of facilities, Vol.3-15-41
Enhanced Intel SpeedStep Technology, Vol.3-15-1
IA32_APERF MSR, Vol.3-15-2
IA32_MPERF MSR, Vol.3-15-1
IA32_THERM_INTERRUPT MSR, Vol.3-15-42
IA32_THERM_STATUS MSR, Vol.3-15-42
interrupt enable/disable flags, Vol.3-15-39
interrupt mechanisms, Vol.3-15-36
Mwait extensions for, Vol.3-15-35
overview of, Vol.3.15-1, Vol.3.15-36
performance state transitions, Vol.3.15-38
sensor interrupt, Vol.3.11-1
setting thermal thresholds, Vol.3.15-42
software controlled clock modulation, Vol.3.15-36, Vol.3.15-40
status flags, Vol.3.15-38
status information, Vol.3.15-38, Vol.3.15-39
stop clock mechanism, Vol.3-15-36
thermal monitor 1 (TM1), Vol.3-15-37
thermal monitor 2 (TM2), Vol.3-15-37
TM flag, CPUID instruction, Vol.3-15-41
thread timeout indicator, Vol.3-17-3, Vol.3-17-7, Vol.3-17-10, Vol.3-17-13, Vol.3-17-15
Threshold #1 interrupt enable bit, Vol.3-15-44, Vol.3-15-47
Threshold #1 value, Vol.3-15-44, Vol.3-15-47
TI (table indicator) flag, segment selector, Vol.3-3-7
Time Stamp Counter, Vol.2-3-248
Timer, local APIC, Vol.3-11-16
Time-stamp counter
counting clockticks, Vol.3-20-139
daught description of, Vol.3-18-42
IA32_TIME_STAMP_COUNTER MSR, Vol.3-18-42
RDTSI instruction, Vol.3-18-42
reading, Vol.3-2-26
software drivers for, Vol.3-20-135
TSC flag, Vol.3-18-42
TSD flag, Vol.3-18-42
Time-stamp counter, reading, Vol.2-4-550, Vol.2-4-552
TLB entry, invalidating (flushing), Vol.2-3-534
TLB entries
description of, Vol.3-12-1, Vol.3-12-5
flushing, Vol.3-12-19
invalidating (flushing), Vol.3-2-25
relationship to PGE flag, Vol.1-23-19
relationship to PSE flag, Vol.3-12-20
TM1 and TM2
See: thermal monitoring, Vol.3-15-37
TMR
TMR (Trigger Mode Register), local APIC, Vol.3-11-30
TOP (stack TOP) field
x87 FPU status word, Vol.1-8-2, Vol.1-9-9
TPR
Task Priority Register, Vol.3-11-38, Vol.3-11-41
TR register, Vol.1-3-6
TR (trace message enable) flag
Trace cache, Vol.1-2-9, Vol.3-12-4, Vol.3-12-5
Translation looksaside buffer (see TLB)
Trap gate, Vol.1-6-13
Trap gates
difference between interrupt and trap gates, Vol.3-6-17
for 16-bit and 32-bit code modules, Vol.3-22-1
handling a virtual-8086 mode interrupt or exception through, Vol.3-21-12
in IDT, Vol.3-6-10
introduction for IA-32e, Vol.3-2-4
introduction to, Vol.3-2-4, Vol.3-2-5
layout of, Vol.3-6-10
Traps
description of, Vol.3-6-5
restarting a program or task after, Vol.3-6-5
Truncation
description of, Vol.1-4-18
with SSE-SSE2 conversion instructions, Vol.1-4-18
Trusted Platform Module, Vol.1-6-5
TS (task switched) flag

TS (task switched) flag, CR0 register, Vol.2-3-170
TSD (time-stamp counter disable) flag
TSS
16-bit TSS, structure of, Vol.3-8-18
32-bit TSS, structure of, Vol.3-8-3
64-bit mode, Vol.3-8-19
CR3 control register (PDTR), Vol.3-8-4, Vol.3-8-17
description of, Vol.3-2-4, Vol.3-2-5, Vol.3-8-1, Vol.3-8-3
EFLAGS register, Vol.3-8-4
EFLAGS, NT, Vol.3-8-15
EIP, Vol.3-8-5
executing a task, Vol.3-8-2
floating-point save area, Vol.1-23-12
format in 64-bit mode, Vol.3-8-19
general-purpose registers, Vol.3-8-4
IA-32e mode, Vol.3-2-5
initialization for multitasking, Vol.3-10-10
interrupt stack table, Vol.3-8-19
invalid TSS exception, Vol.3-6-36
IRET instruction, Vol.3-8-15
I/O map base, Vol.1-19-4
I/O map base address field, Vol.3-8-5, Vol.1-23-29
I/O permission bit map, Vol.1-19-4, Vol.3-8-5, Vol.3-8-19
LDT segment selector field, Vol.3-8-5, Vol.3-8-16
link field, Vol.3-6-17
order of reads/writes to, Vol.1-23-29
pointed to by task-gate descriptor, Vol.3-8-8
previous task link field, Vol.3-8-5, Vol.3-8-15, Vol.3-8-16
privilege-level 0, 1, and 2 stacks, Vol.3-5-17
referenced by task gate, Vol.3-6-17
saving state of EFLAGS register, Vol.1-3-15
segment registers, Vol.3-8-4
T (debug trap) flag, Vol.3-8-5
task register, Vol.3-8-7
using 16-bit TSSs in a 32-bit environment, Vol.1-23-29
virtual-mode extensions, Vol.1-23-29
TSS descriptor
B (busy) flag, Vol.3-8-5
busy flag, Vol.3-8-16
initialization for multitasking, Vol.3-10-10
structure of, Vol.3-8-5, Vol.3-8-6
TSS segment selector
dob, task-gate descriptor, Vol.3-8-8
writes, Vol.1-23-29
TSS, relationship to task register, Vol.2-4-668
Type
checking, Vol.3-5-5
field, IA32_MTRR_DEF_TYPE MSR, Vol.3-12-22
field, IA32_MTRR_PHYSBASEEn MTRR, Vol.3-12-24, Vol.3-12-26
field, segment descriptor, Vol.3-3-10, Vol.3-3-12, Vol.3-3-14, Vol.3-5-2, Vol.3-5-5
of segment, Vol.3-5-6
TZCNT - Count the Number of Trailing Zero Bits, Vol.2-4-713

U
UC (uncacheable) memory type, Vol.3-12-6
UCOMISD - Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS, Vol.2-4-715
UCOMISD instruction, Vol.1-11-7, Vol.2-4-713
UCOMISS - Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS, Vol.2-4-717
UCOMISS instruction, Vol.1-10-9
UE (numeric underflow exception) flag
MXCSR register, Vol.1-11-16
x87 FPU status word, Vol.1-8-5, Vol.1-8-29
UM (numeric underflow exception) mask bit
emulating 8086 operating system calls, Vol.3-21-18
enabling, Vol.3-21-6
entering, Vol.3-21-8
exception and interrupt handling overview, Vol.3-21-11
exceptions and interrupts, handling through a task gate, Vol.3-21-14
exceptions and interrupts, handling through a trap or interrupt gate, Vol.3-21-12
handling exceptions and interrupts through a task gate, Vol.3-21-14
interrupts, Vol.3-21-6
introduction to, Vol.3-2-8
I/O-port-mapped I/O, Vol.3-21-11
leaving, Vol.3-21-9
memory mapped I/O, Vol.3-21-11
native 16-bit mode, Vol.3-22-1
overflow of, Vol.3-21-1
paging of virtual-8086 tasks, Vol.3-21-7
protection within a virtual-8086 task, Vol.3-21-8
special I/O buffers, Vol.3-21-11
structure of a virtual-8086 task, Vol.3-21-7
virtual I/O, Vol.3-21-10
VM flag, EFLAGS register, Vol.3-2-10
Virtual-8086 tasks
paging of, Vol.3-21-7
protection within, Vol.3-21-8
structure of, Vol.3-21-7
VM entries
basic VM-entry checks, Vol.3-27-2
checking guest state
control registers, Vol.3-27-8
debug registers, Vol.3-27-8
descriptor-table registers, Vol.3-27-12
MSRs, Vol.3-27-8
non-register state, Vol.3-27-13
RIP and RFLAGS, Vol.3-27-12
segment registers, Vol.3-27-10
checks on controls, host-state area, Vol.3-27-2
registers and MSRs, Vol.3-27-7
segment and descriptor-table registers, Vol.3-27-7
VMX control checks, Vol.3-27-2
exit-reason numbers, Vol.1-C-1
loading guest state, Vol.3-27-15
control and debug registers, MSRs, Vol.3-27-15
RIP, RSP, RFLAGS, Vol.3-27-17
segment & descriptor-table registers, Vol.3-27-17
loading MSRs, Vol.3-27-18
failure cases, Vol.3-27-18
VM-entry MSR-load area, Vol.3-27-18
overview of failure conditions, Vol.3-27-1
overview of steps, Vol.3-27-1
VMLAUNCH and VMRESUME, Vol.3-27-1
See also: VMCS, VMM, VM exits
VM exits
architectural state
existing before exit, Vol.3-28-1
updating state before exit, Vol.3-28-1
basic VM-exit information fields, Vol.3-28-4
basic exit reasons, Vol.3-28-4
exit qualification, Vol.3-28-4
exception bitmap, Vol.3-28-1
exceptions (faults, traps, and aborts), Vol.3-26-5
exit-reason numbers, Vol.1-C-1
external interrupts, Vol.3-26-6
IA-32 faults and VM exits, Vol.3-26-1
INITs, Vol.3-26-6
instructions that cause:
conditional exits, Vol.3-26-2
unconditional exits, Vol.3-26-2
interrupt-window exiting, Vol.3-26-7
non-maskable interrupts (NMs), Vol.3-26-6
page faults, Vol.3-26-6
start-up IPGs (SIPGs), Vol.3-26-6
INDEX

VME (virtual-8086 mode) flag, EFLAGS register, Vol.1-3-17, Vol.2-3-539
VM (virtual-8086 mode) flag
EFLAGS register, Vol.3-2-8, Vol.3-2-10
VMASKMOV—Conditional SIMD Packed Loads and Stores, Vol.2-5-397
VMAXPH—Return Maximum of Packed FP16 Values, Vol.2-5-400
VMAXSH—Return Maximum of Scalar FP16 Values, Vol.2-5-402
VMCS error numbers, Vol.3-31-1
field encodings, Vol.3-1-7, Vol.1-B-1
16-bit guest-state fields, Vol.1-B-1
16-bit host-state fields, Vol.1-B-2
32-bit guest-state fields, Vol.1-B-8
32-bit read-only data fields, Vol.1-B-8
64-bit control fields, Vol.1-B-2
natural-width control fields, Vol.1-B-9
natural-width guest-state fields, Vol.1-B-10
natural-width host-state fields, Vol.1-B-11
natural-width read-only data fields, Vol.1-B-10
format of VMCS region, Vol.3-25-2
guest-state area, Vol.3-25-3
guest non-register state, Vol.3-25-6
guest register state, Vol.3-25-4
host-state area, Vol.3-25-3, Vol.3-25-8
introduction, Vol.3-25-1
migrating between processors, Vol.3-25-29
software access to, Vol.3-25-29
VMCS data, Vol.3-25-2, Vol.3-26-21
VMCS pointer, Vol.3-25-1
VMCS region, Vol.3-25-1
VMCS revision identifier, Vol.3-25-2, Vol.3-26-21
VM-entry control fields, Vol.3-25-3, Vol.3-25-23
entry controls, Vol.3-25-23
VM-execution control fields, Vol.3-25-3, Vol.3-25-9
controls for CR8 accesses, Vol.3-25-15
CR3-target controls, Vol.3-25-14
exception bitmap, Vol.3-25-14
I/O bitmaps, Vol.3-25-14
masks & read shadows CR0 & CR4, Vol.3-25-14
pin-based controls, Vol.3-25-9
processor-based controls, Vol.3-25-10, Vol.3-25-21
time-stamp counter offset, Vol.3-25-14
VM-exit control fields, Vol.3-25-3, Vol.3-25-21
VM-exit information fields, Vol.3-25-3, Vol.3-25-25
basic exit information, Vol.3-25-26, Vol.3-25-26
basic VM-exit information, Vol.3-25-26
exits due to instruction execution, Vol.3-25-28
exits due to vectored events, Vol.3-25-27
VM-exit error field, Vol.3-25-29
VM-exit error field, Vol.3-25-1, Vol.3-31-31
VMREAD instruction
field encodings, Vol.3-1-7, Vol.1-B-1
VMWRITE instruction
field encodings, Vol.3-1-7, Vol.1-B-1
VMX abort indicator, Vol.3-25-2, Vol.3-26-21
See also: VM entries, VM exits, VMX
VM (virtual-8086 mode) flag
VM entries, Vol.3-28-1
See also: VMCS, VM entries, VM exits, VMX
VMOVSH—Move Scalar FP16 Value, Vol.2-5-408
VMOVW—Move Word, Vol.2-5-410
VMPR/LD instruction, Vol.1-5-38, Vol.3-31-1
VMULPH—Multiply Packed FP16 Values, Vol.2-5-411
VMULSH—Multiply Scalar FP16 Values, Vol.2-5-413
field encodings, Vol.1-B-1
VMX
A20M# signal, Vol.3-24-4
capability MSRs
Vol.4-2-78, Vol.4-2-92, Vol.4-2-102, Vol.4-2-163,
Vol.4-2-212, Vol.4-2-474, Vol.4-2-490
IA32_VMX_CRO_FIXED0 MSR, Vol.1-A-7, Vol.4-2-79,
Vol.4-2-102, Vol.4-2-164, Vol.4-2-212, Vol.4-2-475,
Vol.4-2-490
IA32_VMX_CRO_FIXED1 MSR, Vol.1-A-7, Vol.4-2-79,
Vol.4-2-103, Vol.4-2-164, Vol.4-2-212, Vol.4-2-475,
Vol.4-2-490
IA32_VMX_CR4_FIXED0 MSR, Vol.4-2-79,
Vol.4-2-103, Vol.4-2-164, Vol.4-2-212, Vol.4-2-475,
Vol.4-2-490
IA32_VMX_CR4_FIXED1 MSR, Vol.4-2-79,
Vol.4-2-103, Vol.4-2-164, Vol.4-2-212, Vol.4-2-475,
Vol.4-2-490
Vol.1-A-6, Vol.4-2-78, Vol.4-2-92, Vol.4-2-102,
Vol.4-2-164, Vol.4-2-212, Vol.4-2-474, Vol.4-2-490
Vol.4-2-78, Vol.4-2-92, Vol.4-2-102, Vol.4-2-163,
Vol.4-2-212, Vol.4-2-474, Vol.4-2-490
IA32_VMX_MISC MSR, Vol.3-25-6, Vol.3-27-3, Vol.3-27-13,
Vol.4-32-26, Vol.1-A-6, Vol.4-2-78, Vol.4-2-92,
Vol.4-2-102, Vol.4-2-164, Vol.4-2-212, Vol.4-2-475,
Vol.4-2-490
Vol.4-2-78, Vol.4-2-92, Vol.4-2-102, Vol.4-2-163,
Vol.4-2-212, Vol.4-2-474, Vol.4-2-490
IA32_VMX_PROCBASED_CTLS MSR, Vol.3-25-10, Vol.1-A-2,
Vol.4-2-78, Vol.4-2-79, Vol.4-2-92, Vol.4-2-102,
Vol.4-2-163, Vol.4-2-212, Vol.4-2-474, Vol.4-2-490
IA32_VMS_VMS_ENUM MSR, Vol.4-2-475
CR4 control register, Vol.3-24-3
CR4 fixed bits, Vol.1-A-7
entering operation, Vol.3-24-3
guest software, Vol.3-24-1
IA32_FEATURE_CONTROL MSR, Vol.3-24-3
INIT# signal, Vol.3-24-4
introduction, Vol.1-2-20, Vol.3-24-1
microcode update facilities, Vol.3-26-12
non-root operation, Vol.3-24-1
event blocking, Vol.3-26-13
instruction changes, Vol.3-26-7
overview, Vol.3-26-1
task switches not allowed, Vol.3-26-13
see VM exits
operation restrictions, Vol.3-24-3
root operation, Vol.3-24-1
SM
CR4/VME reserved, Vol.4-32-19
overview, Vol.4-32-2
RSM instruction, Vol.4-32-18
VMCS pointer, Vol.4-32-17
VMX-critical state, Vol.4-32-17
testing for support, Vol.3-24-2
Virtual machine monitor (VMM), Vol.1-2-20
virtualization, Vol.1-2-20
virtual-machine control structure (VMCS), Vol.3-24-2
virtual-machine monitor (VMM), Vol.3-24-1
VM entries and exits, Vol.3-24-1
VM exits, Vol.3-28-1
VMCS pointer, Vol.3-24-2
VM life cycle, Vol.3-24-2
VMXOFF instruction, Vol.3-24-3
VMXON instruction, Vol.3-24-3
VMXON pointer, Vol.3-24-3
VMXON region, Vol.3-24-3
See also:VM, VMCS, VM entries, VM exits
VMXOFF instruction, Vol.1-5-38, Vol.3-24-3, Vol.3-31-1
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between
DWORDS/QUADWORDS to a Pair of Mask Registers, Vol.2-5-414
VPBLENDB—Blend Packed Dwords, Vol.2-5-416
VPBLENDBM/VPBLENDBMW—Blend Byte/Word Vectors Using an OpMask
Control, Vol.2-5-418
VPBLENDBMD/VPBLENDBMQ—Blend Int32/Int64 Vectors Using an OpMask
Control, Vol.2-5-420
VPBROADCAST/W/D/Q—Load With Broadcast Integer Data From General
Purpose Register, Vol.2-5-423
VPBROADCAST—Load Integer and Broadcast, Vol.2-5-426
VPBROADCASTM—Broadcast Mask to Vector Register, Vol.2-5-435
VPMP/B/VPMPUB—Compare Packed Byte Values Into Mask, Vol.2-5-437
VPMPD/VPMPUD—Compare Packed Integer Values Into Mask, Vol.2-5-440
VPMPQ/VPMPUQ—Compare Packed Integer Values into Mask, Vol.2-5-443
VPMPW/VPMPWU—Compare Packed Word Values Into Mask, Vol.2-5-446
VPMPRESSB/VPMPRESSW—Store Sparse Packed Byte/Word Integer
Values Into Dense Memory/Register, Vol.2-5-449
VPMPRESSD—Store Sparse Packed Doubleword Integer Values Into
Dense Memory/Register, Vol.2-5-452
VPMPRESSQ—Store Sparse Packed Quadword Integer Values Into
Dense Memory/Register, Vol.2-5-454
VPCONFLICTD/Q—Detect Conflicts within a Vector of Packed
Dword/Qword Values Into Dense Memory/Register, Vol.2-5-456
VPDPBU5D—Multiply and Add Unsigned and Signed Bytes, Vol.2-5-459
VPDPBUS5D—Multiply and Add Unsigned and Signed Bytes With
Saturation, Vol.2-5-461
VPDPWSD—Multiply and Add Signed Word Integers, Vol.2-5-463
VPDPWSWD—Multiply and Add Signed Word Integers With Saturation,
Vol.2-5-465
VPERM2F128—Permute Floating-Point Values, Vol.2-5-467
VPERM2I128—Permute Integer Values, Vol.2-5-471
VPERM2F128—Permute Doubleword/Word Elements, Vol.2-5-473
VPERM2B—Full Permute of Bytes From Two Tables Overwriting the Index,
Vol.1-6-6
VPERM2B—Full Permute of Bytes From Two Tables Overwriting the Index,
Vol.2-5-476
VPERM2W/D/Q/P5/PD—Full Permute From Two Tables Overwriting the
Index, Vol.2-5-478
VPERMILPD—Permute In-Lane of Pairs of Double Precision Floating-Point
Values, Vol.2-5-484
VPERMILPS—Permute In-Lane of Quadruples of Single Precision
Floating-Point Values, Vol.2-5-489
VPERMILPD—Permute Double Precision Floating-Point Elements,
Vol.2-5-494
VPERMPS—Permute Single Precision Floating-Point Elements,
Vol.2-5-497
VPERMQ—Quads Element Permutation, Vol.2-5-500
VPERM2F128—Permute Floating-Point Values From Two Tables Overwriting a Table,
Vol.2-5-503
VPERM2W/D/Q/P5/PD—Full Permute From Two Tables Overwriting One Table,
Vol.2-5-505
VPMXPANDDB/VPMXPANDW—Expand Byte/Word Values, Vol.2-5-510
VPMXPANDD—Load Sparse Packed Doubleword Integer Values From
Dense Memory/Register, Vol.2-5-513
VPMXPANDQ—Load Sparse Packed Quadword Integer Values From Dense
Memory/Register, Vol.2-5-515
VPAGATHERDD/VPAGATHERDQ—Gather Packed Dword, Packed Qword With
Signed Dword Indices, Vol.2-5-521
VPAGATHERDD/VPAGATHERQ—Gather Packed Dword Values Using Signed
Dword/Qword Indices, Vol.2-5-517
VPAGATHERQ/VPAGATHERQ—Gather Packed Qword values Using Signed
Dword/Qword Indices, Vol.2-5-524
VPAGATHERDD/VPAGATHERQ—Gather Packed Dword, Packed Qword with
Signed Qword Indices, Vol.2-5-528
VPLZCNLD/VIQ—Count the Number of Leading Zero Bits for Packed Dword,
Packed Qword Values, Vol.2-5-531
VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Unsigned Integers
And Add High 52-Bit Products to 64-Bit Accumulators,
Vol.2-5-534
VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the
Low 52-Bit Products to Qword Accumulators, Vol.2-5-536
VPMSKMOV—Conditional SIMD Integer Packed Loads and Stores,
Vol.2-5-538
VPMOVVB2M/VPMOVVB2M/VPMOVVB2M/VPMOVVB2M—Convert a Vector
Register to a Mask, Vol.2-5-541
VPMOVDB/VPMOVUSB/VPMOVUSDB—Down Convert Dword to Byte, Vol.2-5-544
VPMOVDB/VPMOVUSB/VPMOVUSDB—Down Convert Dword to Byte,
Vol.2-5-544
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert Qword to Word, Vol.2-5-548
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert Qword to Word, Vol.2-5-548
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert Qword to Dword,
Vol.2-5-559
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert Qword to Dword,
Vol.2-5-559
VPMOVW/VPMOVSQW/VPMOVUSQW—Down Convert Qword to Dword,
Vol.2-5-563
VPMOVQV/FPMOVSQB/VPMOVUSQB—Down Convert Qword to Dword,
Vol.2-5-563
VPMOVQV/FPMOVSQB/VPMOVUSQB—Down Convert Qword to Dword,
Vol.2-5-566
VPMOVQW/VPMOVSQB/VPMOVUSQB—Down Convert Qword to Dword,
Vol.2-5-567
VPMULTISHFTQB—Select Packed Unaligned Bytes from Quadword Source
Vol.2-5-415
VPMULTISHFTQB—Select Packed Unaligned Bytes from Quadword
Sources, Vol.2-5-571
VPONRT/—Return the Count of Number of Bits Set to 1 in
BYTE/WORD/DWORD/QWORD, Vol.2-5-573
VPPLD/VPPLVD/VPPLQ/VPPLQ—Bit Rotate Left, Vol.2-5-576
VPOR/RDP/VPOR/RDP/VPOR/RDP—Bit Rotate Right, Vol.2-5-580
VPSCATTERQD/VPSCATTERQD/VPSCATTERQD/VPSCATTERQD/VPSCATTERQD—Select
Scattered Dword, Packed Qword with Signed Dword, Signed
Qword Indices, Vol.2-5-594
VPSHLD—Concateenate and Shift Packed Data Left Logical, Vol.2-5-588
VPSHLD—Concateenate and Shift Packed Data Left Logical, Vol.2-5-591
VPSHRD—Concateenate and Shift Packed Data Right Logical, Vol.2-5-594
VPSHFDV—Concateenate and Shift Packed Data Right Logical,
Vol.2-5-597
VPSHFBTOM—Shuffle Bits From Quadword Elements Using Byte
Indexes Into Mask, Vol.2-5-600
INDEX

VPSLLW/VPSLLD/VPSLLQ—Variable Bit Shift Left Logical, Vol.2-5-601
VPSRAW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic, Vol.2-5-606
VPSRLW/VPSRLD/VPSRLQ—Variable Bit Shift Right Logical, Vol.2-5-611
VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic, Vol.2-5-616
VPTESTMB/VPTESTMD/VPTESTMQ—Logical AND and Set Mask, Vol.2-5-619
VPTESTNMB/W/D/Q—Logical NAND and Set, Vol.2-5-622
VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values, Vol.2-5-625
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values, Vol.2-5-632
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values, Vol.2-5-635
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values, Vol.2-5-638
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values, Vol.2-5-642
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value, Vol.2-5-640
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value, Vol.2-5-640
VRCP28BS—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error, Vol.1-6-22
VRCP28BS—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error, Vol.1-6-26
VRCPBPH—Compute Reciprocals of Packed FP16 Values, Vol.2-5-646
VRCPSSH—Compute Reciprocal of Scalar FP16 Value, Vol.2-5-648
VREDECEPD—Perform Reduction Transformation on Packed Float64 Values, Vol.2-5-649
VREDECEPH—Perform Reduction Transformation on Packed FP16 Values, Vol.2-5-652
VREDECEPS—Perform Reduction Transformation on Packed Float32 Values, Vol.2-5-661
VREDUCESS—Perform Reduction Transformation on Scalar Float32 Value, Vol.2-5-659
VREDUCETPD—Round Packed Float64 Values to a Given Number of Fraction Bits, Vol.2-5-663
VREDUCETPH—Round Packed Float64 Values to Include a Given Number of Fraction Bits, Vol.2-5-665
VREDUCETPH—Round Packed Float64 Values to Include a Given Number of Fraction Bits, Vol.2-5-666
VREDUCETPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits, Vol.2-5-666
VREDUCESS—Round Scalar Float64 Value to Include a Given Number of Fraction Bits, Vol.2-5-672
VREDUCESS—Round Scalar Float64 Value to Include a Given Number of Fraction Bits, Vol.2-5-674
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values, Vol.2-5-678
VRSQRT14SD—Compute Approximate Reciprocals of Square Root of Scalar Float64 Value, Vol.2-5-680
VRSQRT14SS—Compute Approximate Reciprocals of Square Root of Scalar Float32 Value, Vol.2-5-684
VRSQRT28BS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error, Vol.1-6-32
VRSQRT28BS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error, Vol.1-6-30
VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error, Vol.1-6-30
VRSQRTT2BS—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error, Vol.1-6-30
VRSQRTPH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value, Vol.2-5-686
VRSQRTQPS—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value, Vol.2-5-686
VSCALEFPD—Scale Packed Float64 Values With Float64 Values, Vol.2-5-689
VSCALEFPH—Scale Packed FP16 Values with FP16 Values, Vol.2-5-692
VSCALEFSS—Scale Packed Float32 Values With Float32 Values, Vol.2-5-694
VSCALEFS—Scale Scalar Float64 Values With Float64 Values, Vol.2-5-697
VSCALEFSH—Scale Scalar FP16 Values with FP16 Values, Vol.2-5-699
VSCATTERFPD/VSCATTERFP10PS/VSCATTERFP110PD/VSCATTERFPBF—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices, Vol.2-5-703
VZEROALL—Zero XMM, YMM, and ZMM Registers, Vol.2-5-722
VZEROUPPER—Zero Upper Bits of YMM and ZMM Registers, Vol.2-5-723

W
Waiting instructions, x87 FPU, Vol.1-8-24
GETSEC, Vol.1-6-4
WB (write back) memory type, Vol.3-9-17, Vol.3-2-27, Vol.3-12-8
WB (write-back) pin (Pentium processor), Vol.3-12-13
WBINVD/INVD bit, Vol.2-3-219
W/B/W# pins, Vol.3-12-13
WC buffer (see Write combining (WC) buffer)
WC memory type, Vol.1-10-12
WC (write combining)
flag, IA32_MTRRCAP MSR, Vol.3-12-22
memory type, Vol.3-12-7, Vol.3-12-8
wide dynamic execution, Vol.1-2-4
Word, Vol.1-4-1
WP (write protected) memory type, Vol.3-12-7
WP (write protect) flag
Wraparound mode (MMX instructions), Vol.1-9-4
Write
hit, Vol.3-12-5
Write combining (WC) buffer, Vol.3-12-4, Vol.3-12-8
Write-back and invalidate caches, Vol.1-6-3
Write-back caching, Vol.3-12-6
CPUID flag, Vol.2-3-246
WT (write through) memory type, Vol.3-12-7, Vol.3-12-8
WT# (write-through) pin (Pentium processor), Vol.3-12-13
INDEX

XLAB instruction, Vol.1-6-34
XLAT instruction, Vol.1-6-34
XLAT/XLATB instruction, Vol.1-1-7-23
XMM registers
64-bit mode, Vol.1-3-5
description, Vol.1-10-3
FXSAVE and FXRSTOR instructions, Vol.1-11-23
overview of, Vol.1-3-2
parameters passing in, Vol.1-11-23
saving on a procedure or function call, Vol.1-11-23
XORPD- Bitwise Logical XOR of Packed Double Precision Floating-Point Values, Vol.1-6-38
XORPD instruction, Vol.1-11-7
XORPS- Bitwise Logical XOR of Packed Single Precision Floating-Point Values, Vol.2-5-723, Vol.1-6-41
XORPS instruction, Vol.1-10-9
Vol.1-14-9, Vol.1-14-10
XTEST - Test If In Transactional Execution, Vol.1-6-69

Z
ZE (divide by zero exception) flag
x87 FPU status word, Vol.1-8-5, Vol.1-8-28
ZE (divide by zero exception) flag bit
MXCSR register, Vol.1-11-15
Zero, floating-point format, Vol.1-4-5, Vol.1-4-14
ZF flag, EFLAGS register, Vol.3-5-25
Vol.2-3-582, Vol.2-3-615, Vol.2-3-618, Vol.2-4-555,
Vol.2-5-152
ZM (divide by zero exception) mask bit
MXCSR register, Vol.1-11-15
x87 FPU control word, Vol.1-8-7, Vol.1-8-28

Index-42 Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4